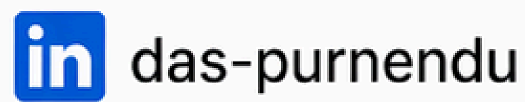


SQL WINDOW FUNCTIONS

YOU MUST KNOW

For Data Science & Data Engineering

Cover mostly asked interview questions

Ranking Functions

Partitioning

Running Calculations

LAG / LEAD

Real Interview Patterns

Save this for your next interview

das-purnendu

WHAT IS A WINDOW FUNCTION?

Window functions perform calculations across a set of table rows that are somehow related to the current row.

GROUP BY

Collapses rows into groups.
Returns one row per group.

Department	Total Salary
HR	250000
IT	550000
Finance	300000

Loses row-level details

VS

WINDOW FUNCTION

Keeps all rows.
Adds calculated values to each row.

Emp ID	Name	Department	Salary	Dept Total Salary
1	Amit	HR	120000	250000
2	Priya	HR	130000	250000
3	Ravi	IT	200000	550000
4	Neha	IT	350000	550000
5	Karan	Finance	300000	300000

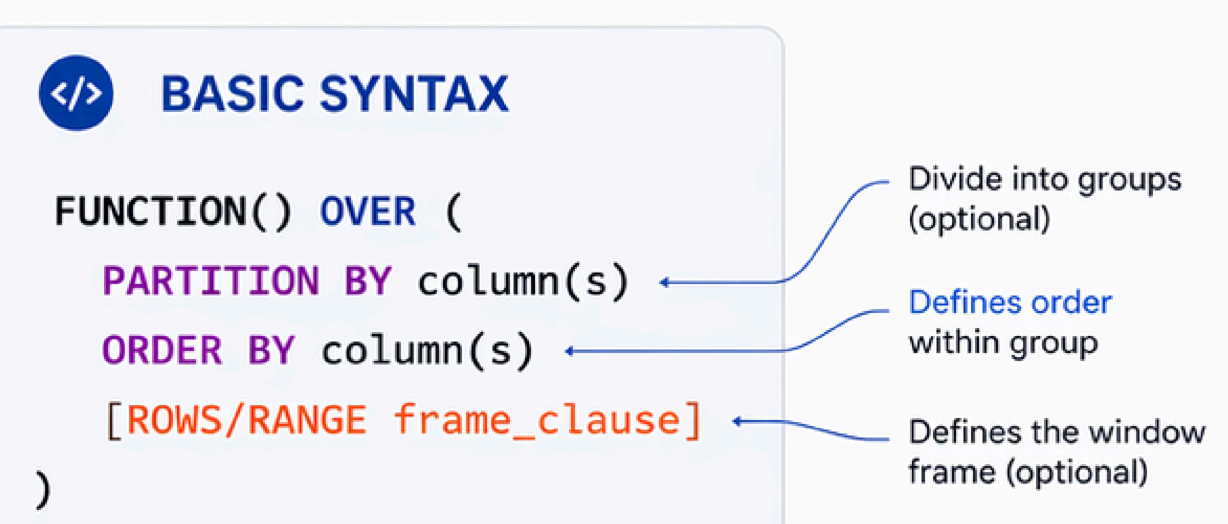
Preserves all rows + adds insights

KEY TAKEAWAYS

Do not reduce the number of rows

Perform calculations across related rows

Ideal for rankings, running totals, moving averages, comparisons, more



BASIC SYNTAX

```
FUNCTION() OVER (
  PARTITION BY column(s)
  ORDER BY column(s)
  [ROWS/RANGE frame_clause]
)
```

Save this slide for quick revision!

ROW_NUMBER()

Assigns a **unique sequential number** to rows within the result set.

EXAMPLE

Employee Salary Table

Emp ID	Name	Department	Salary
1	Amit	HR	120000
2	Priya	HR	130000
3	Ravi	IT	200000
4	Neha	IT	350000
5	Karan	Finance	300000

QUERY

```
SELECT Emp_ID, Name, Department, Salary,
  ROW_NUMBER() OVER (ORDER BY Salary DESC) AS rn
FROM Employees;
```

RESULT

Emp ID	Name	Department	Salary	rn
4	Neha	IT	350000	1
5	Karan	Finance	300000	2
3	Ravi	IT	200000	3
2	Priya	HR	130000	4
1	Amit	HR	120000	5

KEY POINTS

Unique sequential number.

Follows the ORDER BY clause.

Useful for pagination, deduplication, top N per group, etc.

WITH PARTITION BY

```
SELECT Emp_ID, Name, Department, Salary,
   ROW_NUMBER() OVER (PARTITION BY Department
   ORDER BY Salary DESC) AS rn
FROM Employees;
```

→ Resets numbering within each department.

Emp ID	Name	Department	Salary	rn
2	Priya	HR	130000	1
1	Amit	HR	120000	2
4	Neha	IT	350000	1
3	Ravi	IT	200000	2
5	Karan	Finance	300000	1

Save this slide for quick revision!

RANK() vs DENSE_RANK()

Assign rank to rows with gaps (RANK) or without gaps (DENSE_RANK)

EXAMPLE TABLE

Emp ID	Name	Department	Salary
1	Amit	HR	120000
2	Priya	HR	130000
3	Ravi	IT	200000
4	Neha	IT	200000
5	Karan	Finance	170000
6	Sneha	Finance	170000

KEY DIFFERENCE

RANK()

Same values get same rank, but next rank has gaps.

Example: 1, 2, 2, 4

DENSE_RANK()

Same values get same rank, but next rank has no gaps.

Example: 1, 2, 2, 3

A RANK() OVER (ORDER BY Salary DESC)

Emp ID	Name	Department	Salary	RANK()
3	Ravi	IT	200000	1
4	Neha	IT	200000	1
5	Karan	Finance	170000	3
6	Sneha	Finance	170000	3
2	Priya	HR	130000	5
1	Amit	HR	120000	6

B DENSE_RANK() OVER (ORDER BY Salary DESC)

Emp ID	Name	Department	Salary	DENSE_RANK()
3	Ravi	IT	200000	1
4	Neha	IT	200000	1
5	Karan	Finance	170000	2
6	Sneha	Finance	170000	2
2	Priya	HR	130000	3
1	Amit	HR	120000	4

USE CASES

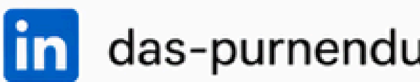
RANK()

- Competition rankings (1st, 2nd, 3rd)
- Leaderboards with gaps
- When rank position matters (skipping next ranks)

DENSE_RANK()

- Leaderboards without gaps
- Grouping or bucket numbering
- When you need continuous rank numbers

Save this slide for quick revision!



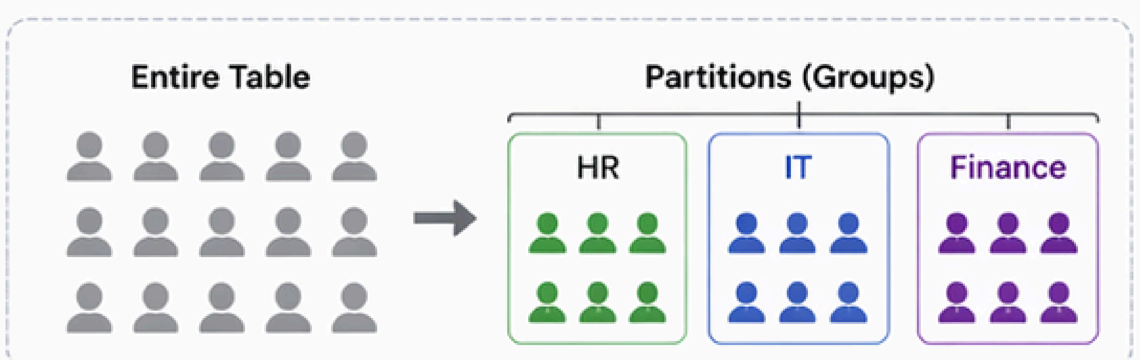
PARTITION BY

Splits data into partitions (groups) and applies the window function within each partition.

CONCEPT

Without PARTITION BY → calculation is over the entire result set.

With PARTITION BY → calculation resets for each group.



EXAMPLE TABLE

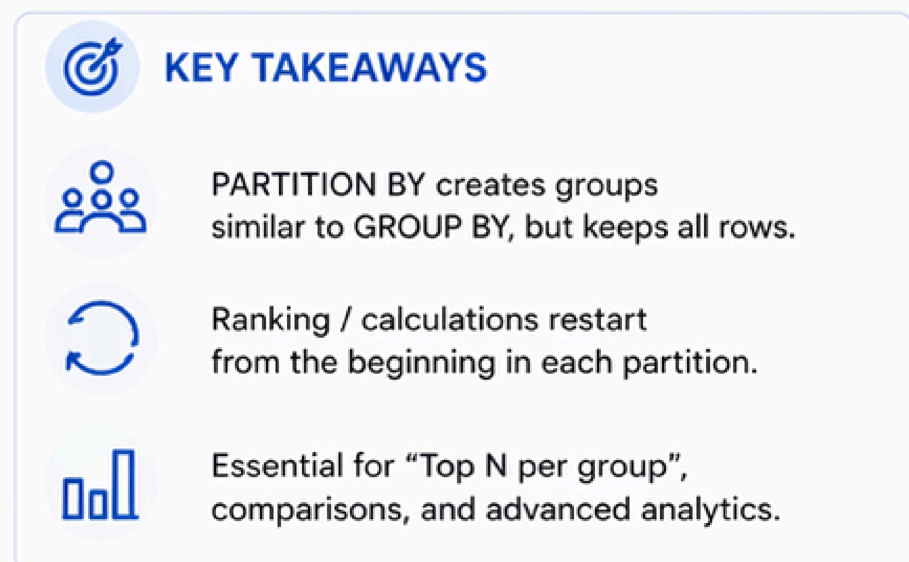
Emp ID	Name	Department	Salary
1	Amit	HR	120000
2	Priya	HR	130000
3	Ravi	IT	200000
4	Neha	IT	350000
5	Karan	Finance	170000
6	Sneha	Finance	170000

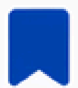
QUERY

```
SELECT Emp_ID, Name, Department, Salary,
   ROW_NUMBER() OVER (
     PARTITION BY Department
     ORDER BY Salary DESC
   ) AS dept_rank
FROM Employees;
```

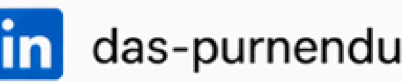
RESULT (Ranking resets in each department)

Emp ID	Name	Department	Salary	dept_rank
2	Priya	HR	130000	1
1	Amit	HR	120000	2
4	Neha	IT	350000	1
3	Ravi	IT	200000	2
5	Karan	Finance	170000	1
6	Sneha	Finance	170000	2





Save this slide for quick revision!



RUNNING TOTAL
(CUMULATIVE SUM)

Calculates **cumulative sum** up to the **current row.**

USE CASES

- Revenue / sales growth tracking
- Running balance, inventory tracking, etc.

EXAMPLE TABLE (Sales)

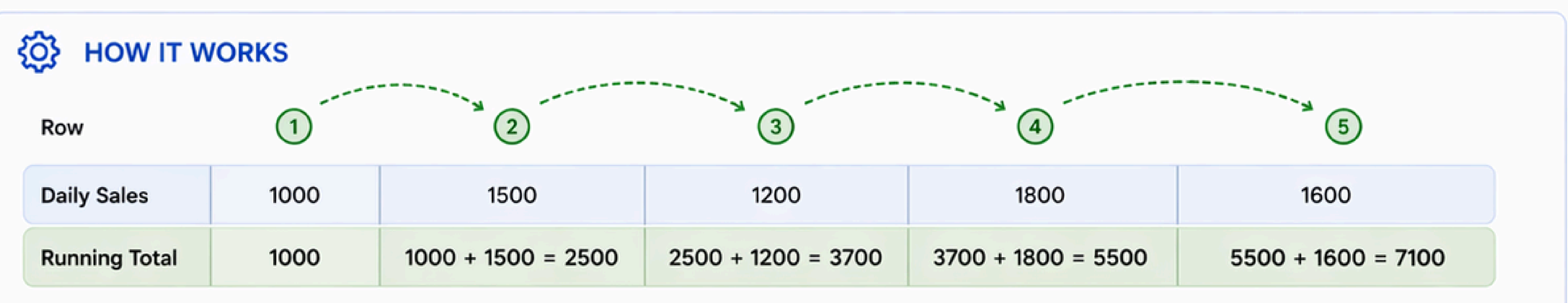
Date	Daily Sales
2024-01-01	1000
2024-01-02	1500
2024-01-03	1200
2024-01-04	1800
2024-01-05	1600

QUERY

```
SELECT
  date,
  daily_sales,
  SUM(daily_sales) OVER (ORDER BY date
  ROWS BETWEEN UNBOUNDED PRECEDING
  AND CURRENT ROW) AS running_total
FROM sales;
```

RESULT

Date	Daily Sales	Running Total
2024-01-01	1000	1000
2024-01-02	1500	2500
2024-01-03	1200	3700
2024-01-04	1800	5500
2024-01-05	1600	7100



HOW IT WORKS

Row	1	2	3	4	5
Daily Sales	1000	1500	1200	1800	1600
Running Total	1000	1000 + 1500 = 2500	2500 + 1200 = 3700	3700 + 1800 = 5500	5500 + 1600 = 7100

KEY POINTS

- Uses `SUM() OVER()` with `ORDER BY`.
- Default frame is `UNBOUNDED PRECEDING` to `CURRENT ROW`.
- Keeps all rows while adding cumulative insights.

VARIATION (Reverse Running Total)

```
SUM(daily_sales) OVER (ORDER BY date
  ROWS BETWEEN CURRENT  ROW AND UNBOUNDED FOLLOWING)
  AS reverse_running_total
```

Save this slide for quick revision!

MOVING AVERAGE

(ROLLING / WINDOWED AVERAGE)

Calculates the average over a **moving window** of rows.

USE CASES

- Smoothing out short-term fluctuations
- Trend analysis in time series data

EXAMPLE TABLE (Sales)

Date	Daily Sales
2024-01-01	1000
2024-01-02	1500
2024-01-03	1200
2024-01-04	1800
2024-01-05	1600
2024-01-06	2000
2024-01-07	1700

QUERY

```
SELECT
  date,
  daily_sales,
  AVG(daily_sales) OVER (
    ORDER BY date
    ROWS BETWEEN 2 PRECEDING
    AND CURRENT ROW
  ) AS moving_avg_3d
FROM sales;
```

RESULT (3-Day Moving Average)

Date	Daily Sales	Moving Avg (3-Day)
2024-01-01	1000	1000.00
2024-01-02	1500	1250.00
2024-01-03	1200	1233.33
2024-01-04	1800	1500.00
2024-01-05	1600	1533.33
2024-01-06	2000	1800.00
2024-01-07	1700	1766.67

HOW THE 3-DAY WINDOW WORKS

Window	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
Dates	01-01	01-01 01-02	01-01 01-02 01-03	01-02 01-03 01-04	01-03 01-04 01-05	01-04 01-05 01-06	01-05 01-06 01-07
Daily Sales	1000	1000 1500	1000 1500 1200	1500 1200 1800	1200 1800 1600	1800 1600 2000	1600 2000 1700
Average	1000.00	1250.00	1233.33	1500.00	1533.33	1800.00	1766.67

KEY POINTS

- Uses `AVG() OVER()` with `ORDER BY`.
- `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW` includes current row + 2 previous rows (total 3).
- Adjust the frame size to change the window.

VARIATIONS

7-Day Moving Average	`ROWS BETWEEN 6 PRECEDING AND CURRENT ROW`
Centered Moving Average	`ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING` (Excludes future rows if not available)

Save this slide for quick revision!

LAG() & LEAD()

Access data from the **previous row (LAG)** or **next row (LEAD)** without self join.

USE CASES

- Compare with previous/next value
- Calculate growth, difference, change
- Detect trends and anomalies

EXAMPLE TABLE (Revenue)

Date	Revenue
2024-01-01	1000
2024-01-02	1200
2024-01-03	1100
2024-01-04	1500
2024-01-05	1300

QUERY

```
SELECT
  date,
  revenue,
  LAG(revenue) OVER (ORDER BY date)
    AS prev_revenue,
  LEAD(revenue) OVER (ORDER BY date)
    AS next_revenue,
  revenue - LAG(revenue) OVER (ORDER
    BY date) AS change_from_prev
FROM sales;
```

RESULT

Date	Revenue	prev_revenue	next_revenue	change_from_prev
2024-01-01	1000	NULL	1200	NULL
2024-01-02	1200	1000	1100	200
2024-01-03	1100	1200	1500	-100
2024-01-04	1500	1100	1300	400
2024-01-05	1300	1500	NULL	-200

LAG()

- Returns data from the **previous** row.
- Default offset = **1** (one row back).
- Use **LAG(column, n, default_value)** for n rows back.

```
LAG(revenue, 1, 0) OVER (ORDER BY date)
```

LEAD()

- Returns data from the **next row.**
- Default offset = **1** (one row ahead).
- Use **LEAD(column, n, default_value)** for n rows ahead.

```
LEAD(revenue, 1, 0) OVER (ORDER BY date)
```

NULL HANDLING

- First row → prev value is NULL
- Last row → next value is NULL
- Provide default value to replace NULL if needed.

```
LAG(revenue, 1, 0)
  OVER (ORDER BY date)
```

REAL WORLD USE CASE

Calculate day-over-day revenue change %

```
SELECT date,
  revenue,
  LAG(revenue) OVER (ORDER BY date) AS prev_rev,
  ROUND( (revenue - LAG(revenue) OVER (ORDER BY date))
    / LAG(revenue) OVER (ORDER BY date) * 100, 2) AS change_pct
FROM sales;
```

Date	Revenue	prev_rev	change_pct (%)
2024-01-01	1000	NULL	NULL
2024-01-02	1200	1000	20.00
2024-01-03	1100	1200	-8.33
2024-01-04	1500	1100	36.36
2024-01-05	1300	1500	-13.33

Save this slide for quick revision!

NTH_VALUE(), FIRST_VALUE(), LAST_VALUE()

Access a specific row's value (NTH), or the first / last value within a window.

EXAMPLE TABLE (Scores)

Student	Subject	Exam Date	Score
Amit	Math	2024-01-01	78
Amit	Math	2024-02-01	85
Amit	Math	2024-03-01	92
Priya	Math	2024-01-01	88
Priya	Math	2024-02-01	91
Priya	Math	2024-03-01	95

QUERY

```
SELECT
  student,
  subject,
  exam_date,
  score,
 NTH_VALUE(score, 2) OVER (
   PARTITION BY student, subject
   ORDER BY exam_date
   ROWS BETWEEN UNBOUNDED PRECEDING
   AND UNBOUNDED FOLLOWING
 ) AS second_score,
 FIRST_VALUE(score) OVER (
   PARTITION BY student, subject
   ORDER BY exam_date
   ROWS BETWEEN UNBOUNDED PRECEDING
   AND UNBOUNDED FOLLOWING
 ) AS first_score,
 LARST_VALUE(score) OVER (
   PARTITION BY student, subject
   ORDER BY exam_date
   ROWS BETWEEN UNBOUNDED PRECEDING
   AND UNBOUNDED FOLLOWING
 ) AS last_score
FROM scores;
```

RESULT

Student	Subject	Exam Date	Score	second_score (N=2)	first_score	last_score
Amit	Math	2024-01-01	78	85	78	92
Amit	Math	2024-02-01	85	85	78	92
Amit	Math	2024-03-01	92	85	78	92
Priya	Math	2024-01-01	88	91	88	95
Priya	Math	2024-02-01	91	91	88	95
Priya	Math	2024-03-01	95	91	88	95

WHAT THEY DO

NTH_VALUE(expr, n)

- Returns the value from the Nth row in the window.
- If n is greater than the number of rows, returns **NULL**.
- Use cases: Median (n = middle), Top/Bottom N, comparison with Nth value.

```
NTH_VALUE(score, 2) OVER (
  PARTITION BY student, subject
  ORDER BY exam_date
  ROWS BETWEEN UNBOUNDED PRECEDING
  AND UNBOUNDED FOLLOWING
)
```

FIRST_VALUE(expr)

- Returns the value from the first row in the window.
- Helpful for baseline comparisons.

```
FIRST_VALUE(score) OVER (
  PARTITION BY student, subject
  ORDER BY exam_date
  ROWS BETWEEN UNBOUNDED PRECEDING
  AND UNBOUNDED FOLLOWING
)
```

LAST_VALUE(expr)

- Returns the value from the last row in the window.
- Helpful for latest value comparisons.

```
LAST_VALUE(score) OVER (
  PARTITION BY student, subject
  ORDER BY exam_date
  ROWS BETWEEN UNBOUNDED PRECEDING
  AND UNBOUNDED FOLLOWING
)
```

KEY NOTES

Functions operate within the defined window (partition + order + frame).

Specify the frame as UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING to consider all rows.

If the required Nth row does not exist, returns **NULL**.

Powerful for rankings, benchmarks, cohorts, and trend analysis.

Save this slide for quick revision!

das-purnendu

WINDOW FUNCTIONS – SUMMARY

Powerful functions that perform calculations across a set of rows related to the current row **without collapsing** the result set.

FUNCTION	WHAT IT DOES	SYNTAX (Simplified)	EXAMPLE USE CASE
PARTITION BY	Splits data into groups (partitions) and applies the window function within each group.	`... OVER (PARTITION BY column ORDER BY column)`	Top N employees per department, rank within category.
RUNNING TOTAL (CUMULATIVE SUM)	Calculates cumulative sum up to the current row.	`SUM(column) OVER (ORDER BY column ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)`	Revenue growth tracking, running balance.
MOVING AVERAGE (ROLLING AVERAGE)	Calculates average over a moving window of rows.	`AVG(column) OVER (ORDER BY column ROWS BETWEEN n PRECEDING AND CURRENT ROW)`	Smoothing short-term fluctuations, trend analysis.
LAG() / LEAD()	Access data from the previous row (LAG) or next row (LEAD) without self join.	`LAG(column, offset, default) OVER (ORDER BY column)` `LEAD(column, offset, default) OVER (ORDER BY column)`	Compare with previous/next value, calculate change, detect trends.
NTH_VALUE(), FIRST_VALUE(), LAST_VALUE()	Access a specific row's value (Nth), or the first / last value within a window.	`NTH_VALUE(column, n) OVER ( ... )` `FIRST_VALUE(column) OVER ( ... )` `LAST_VALUE(column) OVER ( ... )`	Baseline comparisons, best/worst values, benchmarks.

DEFAULT WINDOW FRAME

If ORDER BY is used:

```
RANGE BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW
```

If ORDER BY is not used:

```
RANGE BETWEEN UNBOUNDED PRECEDING
AND UNBOUNDED FOLLOWING
(entire partition)
```

TIPS

- Always use ORDER BY inside OVER() for meaningful results.
- Choose the right frame (ROWS / RANGE) based on your need.
- NULLs in LAG/LEAD: use default value to handle edge rows.
- Window functions run after WHERE, GROUP BY, HAVING.

QUICK REFERENCE – WINDOW FRAME EXAMPLES

FRAME	MEANING
`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`	From first row of partition till current row
`ROWS BETWEEN n PRECEDING AND CURRENT ROW`	n rows before current row (inclusive) till current row
`ROWS BETWEEN CURRENT ROW AND n FOLLOWING`	From current row till next n rows
`ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`	All rows in the partition

SAMPLE MINI OUTPUT (Concept)

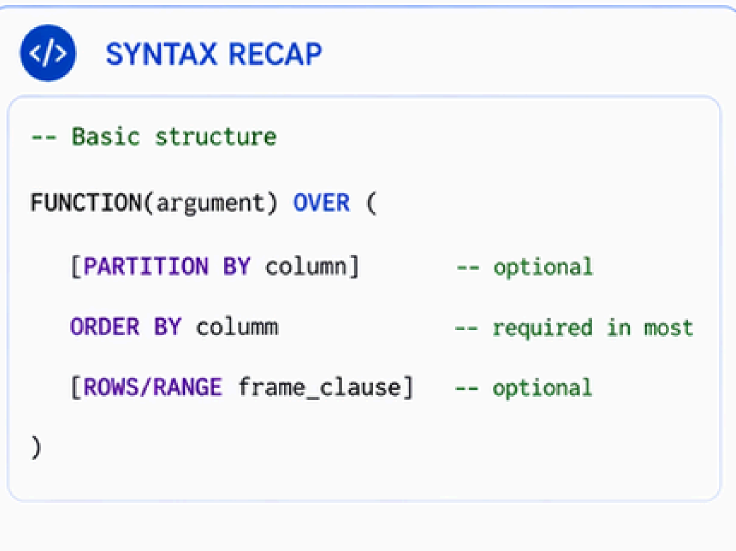
Date	Sales	Running Total	3-Day Avg	Prev Sales	Next Sales
01-Jan	1000	1000	1000.00	NULL	1500
02-Jan	1500	2500	1250.00	1000	1200
03-Jan	1200	3700	1233.33	1500	1800
04-Jan	1800	5500	1500.00	1200	1600
05-Jan	1600	7100	1533.33	1800	NULL

You've reached the last slide! Keep practicing and keep growing!

BONUS – QUICK REFERENCE & PRACTICE

Use this cheat sheet to quickly recall syntax, choose the right function, and practice with mini challenges.

SYNTAX RECAP



```
-- Basic structure

FUNCTION(argument) OVER (
    [PARTITION BY column]        -- optional
    ORDER BY column              -- required in most
    [ROWS/RANGE frame_clause]    -- optional
)
```

CHOOSE THE RIGHT FUNCTION

Goal	Use This
Cumulative total	RUNNING TOTAL (SUM() OVER ...)
Smooth / average trend	MOVING AVERAGE (AVG() OVER ...)
Compare with previous / next row	LAG() / LEAD()
First / last / Nth value in window	FIRST_VALUE() / LAST_VALUE() / NTH_VALUE()

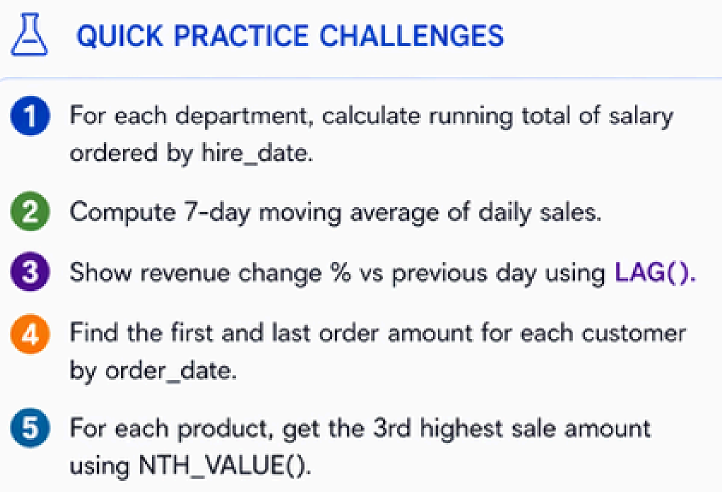
WHEN TO USE WHAT?

PARTITION BY	Group rows before applying window function.
ORDER BY	Defines the logical order inside each partition.
ROWS	Row-based frame (physical count of rows).
RANGE	Value-based frame (logical range of values).
UNBOUNDED PRECEDING	From the start of the partition till current row.
CURRENT ROW	Up to and including the current row.
UNBOUNDED FOLLOWING	From current row till the end of the partition.

COMMON FRAME PATTERNS

Frame	Meaning
UNBOUNDED PRECEDING AND CURRENT ROW	Cumulative from start till current row.
1 PRECEDING AND CURRENT ROW	Current row and previous 1 row.
2 PRECEDING AND 2 FOLLOWING	Centered window: 2 rows before and 2 after.
CURRENT ROW AND UNBOUNDED FOLLOWING	From current row till the end.

QUICK PRACTICE CHALLENGES



1. For each department, calculate running total of salary ordered by hire_date.
2. Compute 7-day moving average of daily sales.
3. Show revenue change % vs previous day using LAG().
4. Find the first and last order amount for each customer by order_date.
5. For each product, get the 3rd highest sale amount using NTH_VALUE().

COMMON MISTAKES

- ✕ Forgetting ORDER BY inside OVER().
- ✕ Using default frame when different frame is needed.
- ✕ Expecting LAST_VALUE() to return last row without adjusting the frame.
- ✕ Using LAG()/LEAD() without handling NULL values.
- ✕ Not understanding the difference between ROWS and RANGE.

PRO TIPS

Always start with PARTITION BY + ORDER BY first. Then add frame if needed.

Visualize your window mentally: know your start, end and order.

Test with small datasets first. Verify edge cases (1st row, last row).

Window functions are powerful for analytics, reports & dashboards.

Practice daily – consistency turns concepts into skills!

You've learned a powerful set of tools! Keep practicing window functions and you'll be unstoppable in SQL analytics. 👍

WHAT YOU CAN EXPECT

Keep learning. Keep building. Keep growing.
See you in the next post! 🚀