

Chapter 24: MySQL Window Functions

24.1 Introduction to Window Functions

- * Window functions perform **calculations** across a set of table rows that are somehow related to the current row.
- * Unlike GROUP BY, window functions **do not reduce** the number of rows in the result.
- * They are very useful for **running totals, ranking, moving averages, percentages, comparisons** etc.
- * Window functions are used with the **OVER()** clause.

Key Points:

- ✓ Window functions work on a set of rows related to the current row.
- ✓ They return a value for each row without collapsing rows.
- ✓ They are executed after WHERE, GROUP BY and HAVING clauses.
- ✓ They are specified using the OVER() clause.



MySQL



24.2 OVER()

- * The OVER() clause determines the **window (set of rows)** on which the window function operates.
- * It may contain **PARTITION BY, ORDER BY** and **frame specification**.
- * Syntax:

```
SELECT column_name(s),  
       window_function() OVER (  
         [ PARTITION BY column_name(s) ]  
         [ ORDER BY column_name(s) ]  
         [ frame_specification ]  
       ) AS alias_name  
FROM table_name;
```



Components of OVER():

Component	Description
PARTITION BY	Divides the result set into partitions or groups.
ORDER BY	Orders rows within each partition.
frame_specification	Defines the subset of rows (window frame) used for calculation. (Optional)



Key Takeaway:



The OVER() clause is the **heart of window functions**. It defines how rows are grouped, ordered and considered for calculations.



24.3 PARTITION BY

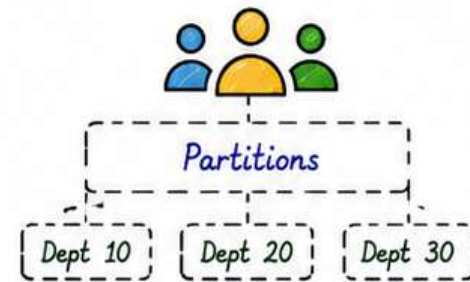
- * **PARTITION BY** divides the result set into **partitions (groups)** based on the values of one or more columns.
- * Window functions are applied **separately to each partition**.
- * It is similar to **GROUP BY** in grouping, but it **does not collapse rows**.

Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       SUM(salary) OVER (PARTITION BY dept_id) AS dept_total  
FROM employees;
```

Output :

emp_id	emp_name	dept_id	salary	dept_total
1	Alice	10	60000	170000
2	Bob	10	70000	170000
3	Charlie	20	50000	120000
4	David	20	70000	120000
5	Eve	30	90000	90000



Key Points :

- ✓ Partitions are created using **PARTITION BY**.
- ✓ Functions are calculated within each partition.
- ✓ The number of rows in the result remains the same.

24.4 ORDER BY in Window Functions

- * **ORDER BY** in window functions determines the **order of rows** within each partition.
- * It is used to define the **sequence** in which calculations are performed.
- * Without **ORDER BY**, the order is **not guaranteed**.

Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS rn  
FROM employees;
```

Output :

emp_id	emp_name	dept_id	salary	rn
2	Bob	10	70000	1
1	Alice	10	60000	2
4	David	20	70000	1
3	Charlie	20	50000	2
5	Eve	30	90000	1

Key Takeaway :



PARTITION BY groups the rows, and **ORDER BY** sorts the rows within each group for accurate calculations.



Key Points :

- ✓ **ORDER BY** controls the ordering within each partition.
- ✓ **ASC** is default (ascending order).
- ✓ **DESC** is used for descending order.
- ✓ It works with ranking, numbering and aggregate functions.

24.5 ROW_NUMBER()

- * ROW_NUMBER() assigns a **unique sequential number** to each row within a **partition**.
- * The numbering starts from **1** for each partition.
- * It does **not** consider **ties**, every row gets a different number.
- * It is commonly used for **pagination**, **removing duplicates**, etc.



ROW_NUMBER()

Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS rn  
FROM employees;
```



Output :

emp_id	emp_name	dept_id	salary	rn
2	Bob	10	70000	1
1	Alice	10	60000	2
4	David	20	70000	1
3	Charlie	20	50000	2
5	Eve	30	90000	1

Key Points :

- ✓ ROW_NUMBER() gives a unique number to each row.
- ✓ It resets for every partition.
- ✓ Useful for top-N queries and pagination.
- ✓ No two rows in a partition will have the same row number.

24.6 RANK()

- * RANK() assigns a **rank** to each row within a partition based on the **order**.
- * If there are **ties**, the **same rank** is given and the next rank is **skipped**.
- * It is useful in **leaderboards**, **competition results**, etc.



Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS rk  
FROM employees;
```



Output :

emp_id	emp_name	dept_id	salary	rk
2	Bob	10	70000	1
1	Alice	10	60000	2
4	David	20	70000	1
3	Charlie	20	50000	2
5	Eve	30	90000	1

Key Points :

- ✓ RANK() assigns the same rank to ties.
- ✓ The next rank is skipped after a tie.
- ✓ Useful for rankings, leaderboards and reports.
- ✓ Ranks start from 1.

Key Takeaway :



- ✓ Use ROW_NUMBER() when every row must have a unique number.
- ✓ Use RANK() when ties should have the same rank and gaps are acceptable.



24.7 DENSE_RANK()

- * `DENSE_RANK()` assigns a **rank** to each row within a partition based on the **order**.
- * If there are ties, it gives the same rank and **does not skip** the next rank.
- * Ranks are **continuous without gaps**.
- * It is useful in reports, leaderboards, grouping similar values, etc.



`DENSE_RANK()`

Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       DENSE_RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS drk  
FROM employees;
```



Output :

emp_id	emp_name	dept_id	salary	drk
2	Bob	10	70000	1
1	Alice	10	60000	2
4	David	20	70000	1
3	Charlie	20	50000	2
5	Eve	30	90000	1

Key Points :

- ✓ `DENSE_RANK()` does not skip ranks after ties.
- ✓ Ranks are continuous.
- ✓ Ties get the same rank.
- ✓ Useful for leaderboards, reports and grouping.
- ✓ Ranks start from 1.



MySQL

24.8 NTILE()

- * `NTILE(n)` divides the rows in a partition into **n** almost equal buckets.
- * Each row is assigned a **bucket number** from 1 to **n**.
- * The difference between buckets will be **at most one row**.
- * Useful for distribution, sampling, percentile analysis, etc.



`NTILE(n)`

Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       NTILE(3) OVER (PARTITION BY dept_id ORDER BY salary DESC) AS bucket_no  
FROM employees;
```

	✓
	✓
	✓
	✓

Output :

emp_id	emp_name	dept_id	salary	bucket_no
2	Bob	10	70000	1
1	Alice	10	60000	2
4	David	20	70000	1
3	Charlie	20	50000	2
5	Eve	30	90000	1

Key Points :

- ✓ `NTILE(n)` splits rows into **n** buckets.
- ✓ Bucket numbers range from 1 to **n**.
- ✓ Buckets are as equal in size as possible.
- ✓ Useful for percentile analysis, sampling and load balancing.

Key Takeaway :



- ✓ Use `DENSE_RANK()` when you want continuous ranks without gaps.
- ✓ Use `NTILE(n)` when you want to distribute rows into **n** equal groups.



24.9 LAG()

- * LAG() accesses data from a **previous row** in the result set.
- * It is used to **compare** the current row with one or more **preceding rows**.
- * The default offset is **1**.
- * You can provide a **default value** if the previous row does not exist.
- * Useful in **trend analysis**, **difference calculations**, etc.

Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       LAG(salary, 1, 0) OVER (PARTITION BY dept_id ORDER BY salary) AS prev_salary  
FROM employees;
```

Output :

emp_id	emp_name	dept_id	salary	prev_salary
3	Charlie	20	50000	0
4	David	20	70000	50000
1	Alice	10	60000	0
2	Bob	10	70000	60000
5	Eve	30	90000	0

Key Takeaway :



Use **LAG()** to look back at previous rows and compare values with the current row.

Key Points :

- ✓ **LAG()** returns data from a previous row.
- ✓ Syntax: **LAG(column, offset, default)**.
- ✓ Default offset is **1**.
- ✓ If no previous row exists, **default value** is returned.
- ✓ Useful for comparison and trend analysis.



24.10 LEAD()

- * LEAD() accesses data from a **following row** in the result set.
- * It is used to **compare** the current row with one or more **subsequent rows**.
- * The default offset is **1**.
- * You can provide a **default value** if the following row does not exist.
- * Useful in **trend analysis**, **forecasting**, etc.

Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       LEAD(salary, 1, 0) OVER (PARTITION BY dept_id ORDER BY salary) AS next_salary  
FROM employees;
```

Output :

emp_id	emp_name	dept_id	salary	next_salary
3	Charlie	20	50000	70000
4	David	20	70000	0
1	Alice	10	60000	70000
2	Bob	10	70000	0
5	Eve	30	90000	0

Key Takeaway :



Use **LEAD()** to look ahead at following rows and compare values with the current row.

Key Points :

- ✓ **LEAD()** returns data from a following row.
- ✓ Syntax: **LEAD(column, offset, default)**.
- ✓ Default offset is **1**.
- ✓ If no following row exists, **default value** is returned.
- ✓ Useful for comparison, forecasting and trend analysis.



24.11 FIRST_VALUE()

- * FIRST_VALUE() returns the value from the first row in the window frame.
- * It is useful when you want the first value for comparison.
- * The default frame is FROM UNBOUNDED PRECEDING TO CURRENT ROW.
- * You can change the frame for different results.
- * Commonly used in reporting, analysis, and data comparison.

Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       FIRST_VALUE(salary) OVER (PARTITION BY dept_id ORDER BY salary ASC  
                                ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS first_sal  
FROM employees;
```

Output :

emp_id	emp_name	dept_id	salary	first_sal
3	Charlie	20	50000	50000
4	David	20	70000	50000
1	Alice	10	60000	60000
2	Bob	10	70000	60000
5	Eve	30	90000	90000

Key Takeaway :



Use FIRST_VALUE() when you need the first value in a partition based on the order.

Key Points :

- ✓ Returns the value from the first row in the window frame.
- ✓ Syntax : FIRST_VALUE(column).
- ✓ Default frame is UNBOUNDED PRECEDING TO CURRENT ROW.
- ✓ Can be used with different frames.
- ✓ Useful for comparisons and reporting.



FIRST_VALUE()



MySQL

24.12 LAST_VALUE()

- * LAST_VALUE() returns the value from the last row in the window frame.
- * It is useful when you want the most recent or last value.
- * The default frame is FROM UNBOUNDED PRECEDING TO CURRENT ROW.
- * To get the true last value in the partition, change the frame to UNBOUNDED FOLLOWING.
- * Useful in trend analysis, comparisons, and reporting.

Example :

```
SELECT emp_id, emp_name, dept_id, salary,  
       LAST_VALUE(salary) OVER (PARTITION BY dept_id ORDER BY salary ASC  
                                ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS last_sal  
FROM employees;
```

Output :

emp_id	emp_name	dept_id	salary	last_sal
3	Charlie	20	50000	70000
4	David	20	70000	70000
1	Alice	10	60000	70000
2	Bob	10	70000	70000
5	Eve	30	90000	90000

Key Takeaway :



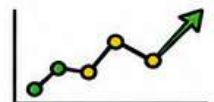
Use LAST_VALUE() to get the last value in a partition.

Key Points :

- ✓ Returns the value from the last row in the window frame.
- ✓ Syntax : LAST_VALUE(column).
- ✓ To get the last value of the partition, use frame UNBOUNDED FOLLOWING.
- ✓ Useful for trend analysis and reporting.



LAST_VALUE()



MySQL



24.13 Running Totals

- * Running totals calculate a **cumulative sum** as you move through the rows.
- * It adds the **current row** value to the total of all **preceding rows**.
- * **ORDER BY** is required to define the sequence.
- * Very useful in financial reports, sales reports, dashboards, etc.
- * Uses aggregate functions like **SUM()** with window frame.



Running Totals

Example:

```
SELECT emp_id, emp_name, dept_id, salary,  
       SUM(salary) OVER (PARTITION BY dept_id ORDER BY emp_id  
                         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS running_total  
FROM employees;
```



Output:

emp_id	emp_name	dept_id	salary	running_total
1	Alice	10	60000	60000
2	Bob	10	70000	130000
3	Charlie	20	50000	50000
4	David	20	70000	120000
5	Eve	30	90000	90000

Key Points:

- ✓ Running totals show **cumulative** sum up to the current row.
- ✓ Uses **SUM()** with window frame.
- ✓ **ORDER BY** defines the sequence.
- ✓ Partitioning allows separate running totals for groups.
- ✓ Useful in reports, dashboards and trend analysis.

Key Takeaway:

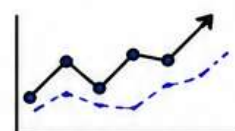


Use **running totals** to calculate cumulative values that grow as you move through ordered rows.



24.14 Moving Averages

- * Moving averages calculate the **average** of a set of rows within a window.
- * It smooths out short-term fluctuations and shows trends.
- * Requires **ORDER BY** and a window frame with a specific number of rows.
- * Commonly used in sales, stock prices, sensor data, etc.
- * Uses **AVG()** with window frame.



Moving Average

Example:

```
SELECT emp_id, emp_name, dept_id, salary,  
       AVG(salary) OVER (PARTITION BY dept_id ORDER BY emp_id  
                        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving_avg  
FROM employees;
```



Output:

emp_id	emp_name	dept_id	salary	moving_avg (3 rows)
1	Alice	10	60000	60000.00
2	Bob	10	70000	65000.00
3	Charlie	10	80000	70000.00
4	David	10	90000	80000.00
5	Eve	10	50000	73333.33

Key Points:

- ✓ Moving averages smooth out short-term variations.
- ✓ Uses **AVG()** with window frame.
- ✓ **ROWS BETWEEN n PRECEDING AND CURRENT ROW** defines the window.
- ✓ Change **n** to adjust the window size.
- ✓ Useful for trend analysis and forecasting.

Key Takeaway:



Use **moving averages** to understand trends by averaging values over a moving window of rows.



24.15 Top-N Queries

- * Top-N queries are used to fetch the **top N** rows from a result set based on some ranking criteria.
- * Window functions like **ROW_NUMBER()**, **RANK()**, **DENSE_RANK()**, **NTILE()** are commonly used.
- * Useful in **leaderboards**, highest salaries, top sales, best performers, etc.
- * We rank the rows and then filter the top **N** rows.

Example :

```
-- Get top 3 highest salaries using ROW_NUMBER()
SELECT emp_id, emp_name, dept_id, salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS rn
FROM employees;

SELECT emp_id, emp_name, dept_id, salary
FROM (
    SELECT emp_id, emp_name, dept_id, salary,
           ROW_NUMBER() OVER (ORDER BY salary DESC) AS rn
    FROM employees
) AS ranked_emps
WHERE rn <= 3;
```

Output (Top 3 Highest Salaries):

emp_id	emp_name	dept_id	salary	rn
5	Eve	30	90000	1
2	Bob	10	70000	2
4	David	20	70000	3

More Examples :

- Top 2 highest salaries in each department
→ Use **ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DESC)**.
- Top N products by sales
→ Use **RANK() OVER (ORDER BY total_sales DESC)**.
- Top N records with ties
→ Use **DENSE_RANK() OVER (ORDER BY score DESC)**.

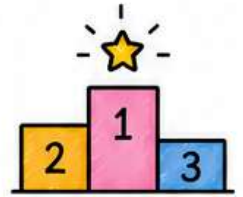
Key Takeaway :



Top-N queries help you quickly find the **best** or **highest** values based on your ranking needs using window functions.



Top-N Queries



MySQL

Key Points:

- ✓ **Rank** rows using a window function based on desired criteria.
- ✓ Filter the ranked results to get **top N** rows.
- ✓ **ROW_NUMBER()** is used when we need exact N rows.
- ✓ **RANK()** or **DENSE_RANK()** can be used when we want to handle ties.
- ✓ Common in **leaderboards**, reports, dashboards, etc.

