**Types of MySQL Functions**

**a. Aggregate Functions**

**Aggregate functions** perform calculations on a set of values and return a single result. They are often used with the GROUP BY clause.

* **COUNT()**: Counts the number of rows.

sql

Copy code

SELECT COUNT(\*) FROM employees;

* **SUM()**: Calculates the sum of a numeric column.

sql

Copy code

SELECT SUM(salary) FROM employees;

* **AVG()**: Computes the average value.

sql

Copy code

SELECT AVG(salary) FROM employees;

* **MIN()** and **MAX()**: Find the minimum and maximum values.

sql

Copy code

SELECT MIN(salary), MAX(salary) FROM employees;

**b. Scalar Functions**

**Scalar functions** operate on a single value and return a single value. They can be used in SELECT, WHERE, and other clauses.

* **UPPER()** and **LOWER()**: Convert strings to uppercase or lowercase.

sql

Copy code

SELECT UPPER(name) FROM employees;

* **ROUND()**: Rounds a numeric value.

sql

Copy code

SELECT ROUND(salary, 2) FROM employees;

* **NOW()**: Returns the current date and time.

sql

Copy code

SELECT NOW();

**c. String Functions**

These functions manipulate string data types.

* **CONCAT()**: Concatenates two or more strings.

sql

Copy code

SELECT CONCAT(first\_name, ' ', last\_name) AS full\_name FROM employees;

* **SUBSTRING()**: Extracts a substring from a string.

sql

Copy code

SELECT SUBSTRING(name, 1, 3) FROM employees;

* **TRIM()**: Removes leading and trailing spaces.

sql

Copy code

SELECT TRIM(name) FROM employees;

**d. Date and Time Functions**

These functions handle date and time data types.

* **DATE\_ADD()** and **DATE\_SUB()**: Add or subtract time intervals.

sql

Copy code

SELECT DATE\_ADD(hire\_date, INTERVAL 1 YEAR) FROM employees;

* **DATEDIFF()**: Calculates the difference between two dates.

sql

Copy code

SELECT DATEDIFF(NOW(), hire\_date) AS days\_worked FROM employees;

* **YEAR(), MONTH(), DAY()**: Extract parts of a date.

sql

Copy code

SELECT YEAR(hire\_date), MONTH(hire\_date), DAY(hire\_date) FROM employees;

**e. Mathematical Functions**

Perform mathematical calculations.

* **ABS()**: Returns the absolute value.

sql

Copy code

SELECT ABS(-5);

* **POWER()**: Raises a number to a power.

sql

Copy code

SELECT POWER(2, 3); -- Returns 8

* **SQRT()**: Calculates the square root.

sql

Copy code

SELECT SQRT(16); -- Returns 4

**f. Conditional Functions**

Enable conditional logic within queries.

* **IF()**: Returns one value if a condition is true and another if false.

sql

Copy code

SELECT name, IF(salary > 50000, 'High', 'Low') AS salary\_level FROM employees;

* **CASE**: Provides more complex conditional logic.

sql

Copy code

SELECT name,

CASE

WHEN salary > 70000 THEN 'Very High'

WHEN salary > 50000 THEN 'High'

ELSE 'Low'

END AS salary\_level

FROM employees;

**g. JSON Functions**

Handle JSON data types (available from MySQL 5.7 onwards).

* **JSON\_EXTRACT()**: Extracts data from a JSON document.

sql

Copy code

SELECT JSON\_EXTRACT(data, '$.name') FROM customers;

* **JSON\_ARRAYAGG()**: Aggregates values into a JSON array.

sql

Copy code

SELECT JSON\_ARRAYAGG(name) FROM employees;

**h. Window Functions**

**Window functions** perform calculations across a set of table rows related to the current row. They are powerful for complex analytics and reporting. The PARTITION BY clause is integral to window functions.

* **ROW\_NUMBER()**, **RANK()**, **DENSE\_RANK()**
* **LEAD()**, **LAG()**
* **SUM() OVER (PARTITION BY ...)**, **AVG() OVER (PARTITION BY ...)**

**3. Window Functions and the PARTITION BY Clause**

**a. What are Window Functions?**

**Window functions** perform calculations across a set of table rows that are somehow related to the current row. Unlike aggregate functions, window functions do not cause rows to become grouped into a single output row; the rows retain their separate identities.

**b. Syntax of Window Functions**

The general syntax for a window function using PARTITION BY is:

sql

Copy code

<window\_function> OVER (

PARTITION BY <expression>

ORDER BY <expression>

[ROWS <frame\_specification>]

)

* **<window\_function>**: The function to apply (e.g., ROW\_NUMBER(), SUM(), etc.).
* **PARTITION BY**: Divides the result set into partitions to which the window function is applied.
* **ORDER BY**: Defines the logical order of rows within each partition.
* **ROWS**: (Optional) Specifies the frame of rows within the partition.

**c. Understanding PARTITION BY**

The PARTITION BY clause divides the result set into partitions (groups of rows). The window function is then applied to each partition independently.

**Key Points:**

* If PARTITION BY is omitted, the window function treats all rows of the query result set as a single partition.
* Similar to GROUP BY, but unlike GROUP BY, PARTITION BY does not collapse rows; it retains individual row identities.

**d. Common Window Functions Using PARTITION BY**

1. **ROW\_NUMBER()**: Assigns a unique sequential integer to rows within a partition.
2. **RANK()**: Assigns a rank to each row within a partition, with gaps in ranking.
3. **DENSE\_RANK()**: Assigns a rank to each row within a partition without gaps.
4. **SUM() OVER (PARTITION BY ...)**: Calculates the cumulative sum within each partition.
5. **AVG() OVER (PARTITION BY ...)**: Computes the average within each partition.
6. **LEAD() and LAG()**: Access data from subsequent or preceding rows within a partition.

**4. Practical Examples**

Let's delve into practical examples to illustrate how PARTITION BY works within various window functions.

**Example Setup**

Assume we have the following employees table:

| **employee\_id** | **name** | **department** | **salary** | **hire\_date** |
| --- | --- | --- | --- | --- |
| 1 | Alice | HR | 60000 | 2018-04-23 |
| 2 | Bob | IT | 80000 | 2016-07-12 |
| 3 | Charlie | IT | 75000 | 2019-03-15 |
| 4 | David | Sales | 50000 | 2020-01-10 |
| 5 | Eva | HR | 65000 | 2017-11-30 |
| 6 | Frank | Sales | 52000 | 2019-06-25 |
| 7 | Grace | IT | 77000 | 2018-09-05 |
| 8 | Henry | HR | 62000 | 2021-02-14 |

**a. Using ROW\_NUMBER() with PARTITION BY**

**Objective:** Assign a unique row number to each employee within their department, ordered by salary descending.

sql

Copy code

SELECT

employee\_id,

name,

department,

salary,

ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row\_num

FROM

employees;

**Result:**

| **employee\_id** | **name** | **department** | **salary** | **row\_num** |
| --- | --- | --- | --- | --- |
| 2 | Bob | IT | 80000 | 1 |
| 7 | Grace | IT | 77000 | 2 |
| 3 | Charlie | IT | 75000 | 3 |
| 5 | Eva | HR | 65000 | 1 |
| 8 | Henry | HR | 62000 | 2 |
| 1 | Alice | HR | 60000 | 3 |
| 6 | Frank | Sales | 52000 | 1 |
| 4 | David | Sales | 50000 | 2 |

**b. Using RANK() with PARTITION BY**

**Objective:** Assign ranks to employees within their department based on salary, allowing for ties.

Assume Frank and another employee have the same salary.

sql

Copy code

SELECT

employee\_id,

name,

department,

salary,

RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank\_num

FROM

employees;

**Result:**

| **employee\_id** | **name** | **department** | **salary** | **rank\_num** |
| --- | --- | --- | --- | --- |
| 2 | Bob | IT | 80000 | 1 |
| 7 | Grace | IT | 77000 | 2 |
| 3 | Charlie | IT | 75000 | 3 |
| 5 | Eva | HR | 65000 | 1 |
| 8 | Henry | HR | 62000 | 2 |
| 1 | Alice | HR | 60000 | 3 |
| 6 | Frank | Sales | 52000 | 1 |
| 4 | David | Sales | 50000 | 2 |

**c. Using DENSE\_RANK() with PARTITION BY**

**Objective:** Similar to RANK(), but without gaps in ranking.

sql

Copy code

SELECT

employee\_id,

name,

department,

salary,

DENSE\_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS dense\_rank\_num

FROM

employees;

**Result:**

| **employee\_id** | **name** | **department** | **salary** | **dense\_rank\_num** |
| --- | --- | --- | --- | --- |
| 2 | Bob | IT | 80000 | 1 |
| 7 | Grace | IT | 77000 | 2 |
| 3 | Charlie | IT | 75000 | 3 |
| 5 | Eva | HR | 65000 | 1 |
| 8 | Henry | HR | 62000 | 2 |
| 1 | Alice | HR | 60000 | 3 |
| 6 | Frank | Sales | 52000 | 1 |
| 4 | David | Sales | 50000 | 2 |

sql

Copy code

### d. Using `SUM() OVER (PARTITION BY ...)`

\*\*Objective:\*\* Calculate the total salary per department and display it alongside each employee.

```sql

SELECT

employee\_id,

name,

department,

salary,

SUM(salary) OVER (PARTITION BY department) AS total\_department\_salary

FROM

employees;

**Result:**

| **employee\_id** | **name** | **department** | **salary** | **total\_department\_salary** |
| --- | --- | --- | --- | --- |
| 1 | Alice | HR | 60000 | 60000 + 65000 + 62000 = 187000 |
| 5 | Eva | HR | 65000 | 187000 |
| 8 | Henry | HR | 62000 | 187000 |
| 2 | Bob | IT | 80000 | 80000 + 75000 + 77000 = 232000 |
| 3 | Charlie | IT | 75000 | 232000 |
| 7 | Grace | IT | 77000 | 232000 |
| 4 | David | Sales | 50000 | 50000 + 52000 = 102000 |
| 6 | Frank | Sales | 52000 | 102000 |

**e. Using LEAD() and LAG() with PARTITION BY**

**Objective:** Access data from the next (LEAD()) and previous (LAG()) rows within the same department.

sql

Copy code

SELECT

employee\_id,

name,

department,

salary,

LEAD(salary, 1) OVER (PARTITION BY department ORDER BY salary DESC) AS next\_highest\_salary,

LAG(salary, 1) OVER (PARTITION BY department ORDER BY salary DESC) AS previous\_salary

FROM

employees;

**Result:**

| **employee\_id** | **name** | **department** | **salary** | **next\_highest\_salary** | **previous\_salary** |
| --- | --- | --- | --- | --- | --- |
| 2 | Bob | IT | 80000 | 77000 | NULL |
| 7 | Grace | IT | 77000 | 75000 | 80000 |
| 3 | Charlie | IT | 75000 | NULL | 77000 |
| 5 | Eva | HR | 65000 | 62000 | NULL |
| 8 | Henry | HR | 62000 | 60000 | 65000 |
| 1 | Alice | HR | 60000 | NULL | 62000 |
| 6 | Frank | Sales | 52000 | 50000 | NULL |
| 4 | David | Sales | 50000 | NULL | 52000 |

sql

Copy code

---

## 5. Best Practices and Tips

1. \*\*Use Appropriate Window Functions:\*\* Choose the window function that best fits your analytical needs. For ranking, use `ROW\_NUMBER()`, `RANK()`, or `DENSE\_RANK()` based on whether you need gaps in rankings.

2. \*\*Optimize Performance:\*\*

- \*\*Indexing:\*\* Ensure that columns used in `PARTITION BY` and `ORDER BY` are properly indexed to improve query performance.

- \*\*Limit Data:\*\* Use `WHERE` clauses to limit the dataset before applying window functions.

3. \*\*Understand Frame Specifications:\*\* Window functions can be further refined using `ROWS` or `RANGE` clauses to define the frame (subset of rows) the function operates on.

4. \*\*Combine with Other Clauses:\*\*

- \*\*`GROUP BY`:\*\* Combine window functions with `GROUP BY` for advanced aggregations.

- \*\*Subqueries and CTEs:\*\* Use Common Table Expressions (CTEs) or subqueries to break down complex queries.

5. \*\*Handle NULLs Appropriately:\*\* Be cautious with NULL values as they can affect ordering and calculations.

6. \*\*Test Incrementally:\*\* Build and test your queries step-by-step to ensure each part works as expected before combining them.

7. \*\*Stay Updated:\*\* Ensure you are using a version of MySQL that supports window functions (MySQL 8.0 and later).

---

## Conclusion

MySQL functions, especially window functions using the `PARTITION BY` clause, provide powerful tools for data analysis and complex querying. By understanding and effectively applying these functions, you can perform sophisticated data manipulations and gain deeper insights from your database.

Feel free to ask if you have specific questions or need further clarification on any of the topics covered!