MySQL Window Functions

**Summary**: in this tutorial, you will learn about MySQL window functions and their practical applications for solving analytical query challenges.

MySQL has supported window functions since version 8.0, allowing you to solve query problems more easily and with better performance.

Suppose that we have the sales table which stores the sales by employees and fiscal years:

CREATE TABLE sales(

sales\_employee VARCHAR(50) NOT NULL,

fiscal\_year INT NOT NULL,

sale DECIMAL(14,2) NOT NULL,

PRIMARY KEY(sales\_employee,fiscal\_year)

);

INSERT INTO sales(sales\_employee,fiscal\_year,sale)

VALUES('Bob',2016,100),

('Bob',2017,150),

('Bob',2018,200),

('Alice',2016,150),

('Alice',2017,100),

('Alice',2018,200),

('John',2016,200),

('John',2017,150),

('John',2018,250);

SELECT \* FROM sales;

It’s probably easier to understand window functions is to start with [aggregate functions](https://www.mysqltutorial.org/mysql-aggregate-functions/).

Aggregate functions summarize data from multiple rows into a single result row. For example, the following [SUM()](https://www.mysqltutorial.org/mysql-aggregate-functions/mysql-sum/)function returns the total sales of all employees in the recorded years:

SELECT

SUM(sale)

FROM

sales;

The [GROUP BY](https://www.mysqltutorial.org/mysql-basics/mysql-group-by/) clause allows you to apply aggregate functions to a subset of rows. For example, you may want to calculate the total sales by fiscal years:

SELECT

fiscal\_year,

SUM(sale)

FROM

sales

GROUP BY

fiscal\_year;

In both examples, the aggregate functions reduce the number of rows returned by the query.

Like the aggregate functions with the [GROUP BY](https://www.mysqltutorial.org/mysql-basics/mysql-group-by/) clause, window functions also operate on a subset of rows but they do not reduce the number of rows returned by the query.

For example, the following query returns the sales for each employee, along with the total sales of the employees by fiscal year:

SELECT

fiscal\_year,

sales\_employee,

sale,

SUM(sale) OVER (PARTITION BY fiscal\_year) total\_sales

FROM

sales;

In this example, the SUM() function works as a window function that operates on a set of rows defined by the contents of the OVER clause. A set of rows to which the SUM() function applies is referred to as a window.

The SUM() window function reports not only the total sales by fiscal year as it does in the query with the GROUP BY clause, but also the result in each row, rather than the total number of rows returned.

Note that window functions are performed on the result set after all [JOIN](https://www.mysqltutorial.org/mysql-basics/mysql-join/), [WHERE](https://www.mysqltutorial.org/mysql-basics/mysql-where/), [GROUP BY](https://www.mysqltutorial.org/mysql-basics/mysql-group-by/), and [HAVING](https://www.mysqltutorial.org/mysql-basics/mysql-having/) clauses and before the [ORDER BY](https://www.mysqltutorial.org/mysql-basics/mysql-order-by/), [LIMIT](https://www.mysqltutorial.org/mysql-basics/mysql-limit/) and [SELECT DISTINCT](https://www.mysqltutorial.org/mysql-basics/mysql-distinct/).

**2nd Example:**

CREATE TABLE employee\_performance (

id INT AUTO\_INCREMENT PRIMARY KEY,

employee\_id INT,

employee\_name VARCHAR(50),

department VARCHAR(50),

performance\_score INT,

review\_date DATE

);

INSERT INTO employee\_performance (employee\_id, employee\_name, department, performance\_score, review\_date) VALUES

(1, 'Alice', 'HR', 85, '2023-01-15'),

(2, 'Bob', 'HR', 90, '2023-01-20'),

(3, 'Charlie', 'IT', 88, '2023-01-25'),

(4, 'David', 'IT', 92, '2023-02-10'),

(5, 'Eva', 'Sales', 95, '2023-02-15'),

(6, 'Frank', 'Sales', 89, '2023-02-20'),

(7, 'Grace', 'HR', 87, '2023-03-05'),

(8, 'Hank', 'IT', 85, '2023-03-10'),

(9, 'Ivy', 'Sales', 90, '2023-03-15');

**Query: Using PARTITION BY with AVG**

To calculate the average performance score for each department, you can use the AVG window function with PARTITION BY:

SELECT

employee\_id,

employee\_name,

department,

performance\_score,

AVG(performance\_score) OVER (PARTITION BY department) AS avg\_department\_score

FROM

employee\_performance

ORDER BY

department, performance\_score DESC;

**Example 3:** Ranking Products by Sales Amount

**CREATE TABLE product\_sales\_rank (**

**id INT AUTO\_INCREMENT PRIMARY KEY,**

**product\_id INT,**

**product\_name VARCHAR(50),**

**sale\_amount DECIMAL(10, 2)**

**);**

**INSERT INTO product\_sales\_rank (product\_id, product\_name, sale\_amount) VALUES**

**(1, 'Product A', 100.00),**

**(2, 'Product B', 150.00),**

**(3, 'Product C', 120.00),**

**(1, 'Product A', 200.00),**

**(2, 'Product B', 180.00),**

**(3, 'Product C', 220.00);**

Query: Ranking Products by Sales Amount

**SELECT**

**product\_id,**

**product\_name,**

**sale\_amount,**

**RANK() OVER (PARTITION BY product\_name ORDER BY sale\_amount DESC) AS product\_rank**

**FROM**

**product\_sales\_rank**

**ORDER BY**

**product\_name, product\_rank;**

**Example:** Calculating Cumulative Sales by Month

**CREATE TABLE monthly\_sales (**

**id INT AUTO\_INCREMENT PRIMARY KEY,**

**sale\_date DATE,**

**sale\_amount DECIMAL(10, 2)**

**);**

**INSERT INTO monthly\_sales (sale\_date, sale\_amount) VALUES**

**('2023-01-01', 1000.00),**

**('2023-01-15', 1500.00),**

**('2023-02-01', 2000.00),**

**('2023-02-15', 1800.00),**

**('2023-03-01', 2200.00),**

**('2023-03-15', 1900.00);**

**Query: Calculating Cumulative Sales by Month**

**SELECT**

**sale\_date,**

**sale\_amount,**

**SUM(sale\_amount) OVER (ORDER BY sale\_date) AS cumulative\_sales**

**FROM**

**monthly\_sales**

**ORDER BY**

**sale\_date;**

SELECT

id,

sale\_date,

sale\_amount,

SUM(sale\_amount) OVER (ORDER BY sale\_date) AS cumulative\_sales,

ROW\_NUMBER() OVER (PARTITION BY YEAR(sale\_date), MONTH(sale\_date) ORDER BY sale\_date) AS row\_num

FROM

monthly\_sales

ORDER BY

sale\_date;

Example: Calculating Yearly Sales Growth

CREATE TABLE yearly\_sales (

id INT AUTO\_INCREMENT PRIMARY KEY,

year INT,

sale\_amount DECIMAL(10, 2)

);

INSERT INTO yearly\_sales (year, sale\_amount) VALUES

(2020, 10000.00),

(2021, 15000.00),

(2022, 20000.00),

(2023, 25000.00);

SELECT

year,

sale\_amount,

LAG(sale\_amount) OVER (ORDER BY year) AS previous\_year\_sales,

sale\_amount - LAG(sale\_amount) OVER (ORDER BY year) AS sales\_growth

FROM

yearly\_sales

ORDER BY

year;

SELECT

year, -- Select the year

sale\_amount, -- Select the sales amount for the year

LAG(sale\_amount) OVER (ORDER BY year) AS previous\_year\_sales, -- Get the sales amount for the previous year

sale\_amount - LAG(sale\_amount) OVER (ORDER BY year) AS sales\_growth -- Calculate the growth in sales compared to the previous year

FROM

yearly\_sales -- From the yearly\_sales table

ORDER BY

year; -- Order the results by year

The LAG function can be used with any time interval, such as years, months, weeks, days, or hours. It is not limited to years. The key is how you structure your data and the ORDER BY clause in the OVER window function.

**Example: Using LAG with Daily Interval**

**CREATE TABLE daily\_sales (**

**id INT AUTO\_INCREMENT PRIMARY KEY,**

**sale\_date DATE,**

**sale\_amount DECIMAL(10, 2)**

**);**

**INSERT INTO daily\_sales (sale\_date, sale\_amount) VALUES**

**('2023-01-01', 1000.00),**

**('2023-01-02', 1500.00),**

**('2023-01-03', 2000.00),**

**('2023-01-04', 1800.00),**

**('2023-01-05', 2200.00);**

**SELECT**

**sale\_date,**

**sale\_amount,**

**LAG(sale\_amount) OVER (ORDER BY sale\_date) AS previous\_day\_sales,**

**sale\_amount - LAG(sale\_amount) OVER (ORDER BY sale\_date) AS sales\_growth**

**FROM**

**daily\_sales**

**ORDER BY**

**sale\_date;**

**Example: Using LAG with Monthly Interval**

**CREATE TABLE monthly\_sales1 (**

**id INT AUTO\_INCREMENT PRIMARY KEY,**

**sale\_date DATE,**

**sale\_amount DECIMAL(10, 2)**

**);**

**INSERT INTO monthly\_sales1 (sale\_date, sale\_amount) VALUES**

**('2023-01-01', 1000.00),**

**('2023-02-01', 1500.00),**

**('2023-03-01', 2000.00),**

**('2023-04-01', 1800.00),**

**('2023-05-01', 2200.00);**

**SELECT**

**DATE\_FORMAT(sale\_date, '%Y-%m') AS sale\_month,**

**sale\_amount,**

**LAG(sale\_amount) OVER (ORDER BY sale\_date) AS previous\_month\_sales,**

**sale\_amount - LAG(sale\_amount) OVER (ORDER BY sale\_date) AS sales\_growth**

**FROM**

**monthly\_sales1**

**ORDER BY**

**sale\_date;**

**Example: Using LAG with Hourly Interval**

**CREATE TABLE hourly\_sales (**

**id INT AUTO\_INCREMENT PRIMARY KEY,**

**sale\_datetime DATETIME,**

**sale\_amount DECIMAL(10, 2)**

**);**

**INSERT INTO hourly\_sales (sale\_datetime, sale\_amount) VALUES**

**('2023-07-29 08:00:00', 1000.00),**

**('2023-07-29 09:00:00', 1500.00),**

**('2023-07-29 10:00:00', 2000.00),**

**('2023-07-29 11:00:00', 1800.00),**

**('2023-07-29 12:00:00', 2200.00);**

**SELECT**

**sale\_datetime,**

**sale\_amount,**

**LAG(sale\_amount) OVER (ORDER BY sale\_datetime) AS previous\_hour\_sales,**

**sale\_amount - LAG(sale\_amount) OVER (ORDER BY sale\_datetime) AS sales\_growth**

**FROM**

**hourly\_sales**

**ORDER BY**

**sale\_datetime;**

**MySQL Window Function list**

The following table shows the window functions in MySQL:

| **Name** | **Description** |
| --- | --- |
| [CUME\_DIST](https://www.mysqltutorial.org/mysql-window-functions/mysql-cume_dist-function/) | Calculates the cumulative distribution of a value in a set of values. |
| [DENSE\_RANK](https://www.mysqltutorial.org/mysql-window-functions/mysql-dense_rank-function/) | Assigns a rank to every row within its partition based on the ORDER BY clause. It assigns the same rank to the rows with equal values. If two or more rows have the same rank, then there will be no gaps in the sequence of ranked values. |
| [FIRST\_VALUE](https://www.mysqltutorial.org/mysql-window-functions/mysql-first_value-function/) | Returns the value of the specified expression with respect to the first row in the window frame. |
| [LAG](https://www.mysqltutorial.org/mysql-window-functions/mysql-lag-function/) | Returns the value of the Nth row before the current row in a partition. It returns NULL if no preceding row exists. |
| [LAST\_VALUE](https://www.mysqltutorial.org/mysql-window-functions/mysql-last_value-function/) | Returns the value of the specified expression with respect to the last row in the window frame. |
| [LEAD](https://www.mysqltutorial.org/mysql-window-functions/mysql-lead-function/) | Returns the value of the Nth row after the current row in a partition. It returns NULL if no subsequent row exists. |
| [NTH\_VALUE](https://www.mysqltutorial.org/mysql-window-functions/mysql-nth_value-function/) | Returns value of argument from Nth row of the window frame |
| [NTILE](https://www.mysqltutorial.org/mysql-window-functions/mysql-ntile-function/) | Distributes the rows for each window partition into a specified number of ranked groups. |
| [PERCENT\_RANK](https://www.mysqltutorial.org/mysql-window-functions/mysql-percent_rank-function/) | Calculates the percentile rank of a row in a partition or result set |
| [RANK](https://www.mysqltutorial.org/mysql-window-functions/mysql-rank-function/) | Similar to the DENSE\_RANK() function except that there are gaps in the sequence of ranked values when two or more rows have the same rank. |
| [ROW\_NUMBER](https://www.mysqltutorial.org/mysql-window-functions/mysql-row_number-function/) | Assigns a sequential integer to every row within its partition |

MySQL CUME\_DIST Function

Overview of MySQL CUME\_DIST() Function

The CUME\_DIST() is a [window function](https://www.mysqltutorial.org/mysql-window-functions/) that returns the cumulative distribution of a value within a set of values. It represents the number of rows with values less than or equal to that row’s value divided by the total number of rows.

The returned value of the CUME\_DIST() function is greater than zero and less than or equal to one (0 < CUME\_DIST() <= 1). The repeated column values receive the same CUME\_DIST() value.

The following shows the syntax of the CUME\_DIST() function:

CUME\_DIST()

OVER (

PARTITION BY expr

ORDER BY expr [ASC | DESC]

)Code language: SQL (Structured Query Language) (sql)

In this syntax:

PARTITION BY

The PARTITION BY clause divides the result set into partitions to which the CUME\_DIST() function is applied independently. If you omit the PARTITION BY clause, the function is applied to the whole result set.

ORDER BY

The ORDER BY clause specifies the order of the rows in each partition or the whole result set in case the PARTITION BY is omitted.

The CUME\_DIST() function calculates the cumulative distribution value of each row based on its order in the partition.

The approximate formula of the CUME\_DIST() function is as follows:

ROW\_NUMBER() / total\_rows

MySQL CUME\_DIST() function example

Let’s [create a table](https://www.mysqltutorial.org/mysql-basics/mysql-create-table/) called scores and populate some data for the demonstration:

CREATE TABLE scores (

name VARCHAR(20) PRIMARY KEY,

score INT NOT NULL

);

INSERT INTO

scores(name, score)

VALUES

('Smith',81),

('Jones',55),

('Williams',55),

('Taylor',62),

('Brown',62),

('Davies',84),

('Evans',87),

('Wilson',72),

('Thomas',72),

('Johnson',100);

The following statement uses the CUME\_DIST() function to find the cumulative distribution of the score in the result set:

SELECT

name,

score,

ROW\_NUMBER() OVER (

ORDER BY

score

) row\_num,

CUME\_DIST() OVER (

ORDER BY

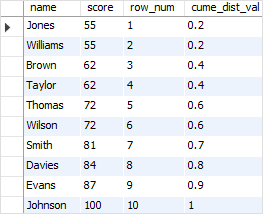
score

) cume\_dist\_val

FROM

scores;

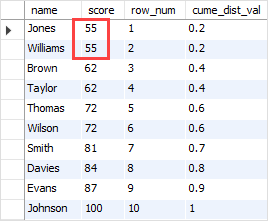
Here is the output:



In this example, the score is sorted in ascending order from 55 to 100. Note that the [ROW\_NUMBER()](https://www.mysqltutorial.org/mysql-window-functions/mysql-row_number-function/) function was added for reference.

So how does the CUME\_DIST() function perform calculation?

In the initial step, the function identifies the number of rows in the result set where the values are less than or equal to 55. This count is found to be 2 for the first row. Next, the CUME\_DIST() function computes the cumulative distribution by dividing this count (2) by the total number of rows in the set, which is 10: 2/10. The result is 0.2 or 20%. The same procedure is then repeated for the second row.



In the case of the third row, the CUME\_DIST() function identifies four rows within the result set where the values are less than or equal to 62. then, the CUME\_DIST() function computes the cumulative distribution by dividing this count (4) by the total number of rows in the set, which is 10: 4/10. The result is 0.4 or 40%.

The same calculation logic is applied to the remaining rows.

**Second Example**-- Step 1: Create the employee table

CREATE TABLE IF NOT EXISTS employee (

emp\_id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255),

salary DECIMAL(10, 2)

);

-- Step 2: Insert sample data into the employee table

INSERT INTO employee (name, salary) VALUES

('John Doe', 50000),

('Jane Smith', 60000),

('Alice Johnson', 70000),

('Bob Brown', 50000),

('Carol White', 80000);

-- Step 3: Use the CUME\_DIST() function to calculate the cumulative distribution of salaries

SELECT

emp\_id,

name,

salary,

CUME\_DIST() OVER (ORDER BY salary) AS salary\_cume\_dist

FROM

employee;

# **MySQL RANK Function**

**Summary**: in this tutorial, you will learn about the MySQL RANK() function and how to apply it to assign a rank to each row within the partition of a result set.

## **Introduction to MySQL RANK() function**

The RANK() function assigns a rank to each row within the partition of a result set. The rank of a row is specified by one plus the number of ranks that come before it.

Suppose you have a sample table as follows:

CREATE TABLE t (

val INT

);

INSERT INTO t(val)

VALUES(1),(2),(2),(3),(4),(4),(5);

SELECT \* FROM t;

The following statement uses the RANK() function to assign a rank to each row from the result set in the t table:

SELECT

val,

RANK() OVER (

ORDER BY val

) my\_rank

FROM

t;

The output indicates that the second and third rows have the same ties so they receive the same rank 2.

The fourth row has rank 4 because the RANK() function skips the rank 3.

## **MySQL RANK() function example**

Let’s use the sales table created in the [window function tutorial](https://www.mysqltutorial.org/mysql-window-functions/) for the demonstration.

If you have not created the sales table yet, here is the script:

CREATE TABLE IF NOT EXISTS sales(

sales\_employee VARCHAR(50) NOT NULL,

fiscal\_year INT NOT NULL,

sale DECIMAL(14,2) NOT NULL,

PRIMARY KEY(sales\_employee,fiscal\_year)

);

INSERT INTO sales(sales\_employee,fiscal\_year,sale)

VALUES('Bob',2016,100),

('Bob',2017,150),

('Bob',2018,200),

('Alice',2016,150),

('Alice',2017,100),

('Alice',2018,200),

('John',2016,200),

('John',2017,150),

('John',2018,250);

SELECT \* FROM sales;

The following statement uses the RANK() function to rank the sales employees by sales amount every year:

SELECT

sales\_employee,

fiscal\_year,

sale,

RANK() OVER (PARTITION BY

fiscal\_year

ORDER BY

sale DESC

) sales\_rank

FROM

sales;

In this example:

* First, the PARTITION BY clause breaks the result sets into partitions by fiscal year.
* Then, the ORDER BY clause sorts the sales employees by sales in descending order.

# **MySQL ROW\_NUMBER() Function**

The ROW\_NUMBER() function in MySQL is used to returns the **sequential number** for each row within its partition. It is a kind of window function. The row number starts from 1 to the number of rows present in the partition.

It is to be noted that MySQL does not support the ROW\_NUMBER() function before version 8.0, but they provide a **session variable** that allows us to emulate this function.

**Syntax**

The following are the basic syntax to use ROW\_NUMBER() in [MySQL](https://www.javatpoint.com/mysql-tutorial):

**Let us demonstrate it using an example**.

First, we are going to create a table named "**Person**" using the below statement:

**CREATE** **TABLE** Person (

**Name** **varchar**(45) NOT NULL,

  Product **varchar**(45) **DEFAULT** NULL,

  Country **varchar**(25) **DEFAULT** NULL,

  Year **int** NOT NULL

);

Next, it is required to add values to this table. Execute the below statement:

**INSERT** **INTO** Person(**Name**, Product, Country, Year)

**VALUES** ('Stephen', 'Computer', 'USA', 2015),

('Joseph', 'Laptop', 'India', 2016),

('John', 'TV', 'USA', 2016),

('Donald', 'Laptop', 'England', 2015),

('Joseph', 'Mobile', 'India', 2015),

('Peter', 'Mouse', 'England', 2016);

Next, execute the SELECT statement to display the records:

mysql> **SELECT** \* **FROM** Person;

Now, we can use the ROW\_NUMBER() function to assign a sequence number for each record using the below statement:

**SELECT** \*,

    ROW\_NUMBER() OVER(PARTITION **BY** Year) **AS** row\_num

**FROM** Person;

Again, we can use the ROW\_NUMBER() function to assign a sequence number for each record within a partition using the below statement:

**SELECT** \*,

    ROW\_NUMBER() OVER(PARTITION **BY** Year) **AS** row\_num

**FROM** Person;

### **MySQL ROW\_NUMBER() Using Session Variable**

In MySQL, session variables are variables that are specific to a particular session or connection. They are used to store temporary data that can be accessed and manipulated during the session. Session variables are defined using the SET statement and can be referenced using the @ symbol.

We can emulate the ROW\_NUMBER() function to add a row number in increasing order using the session variable.

Execute the below statement that add the row number for each row, which starts from 1:

**SET** @row\_number = 0;

**SELECT** **Name**, Product, Year, Country,

    (@row\_number:=@row\_number + 1) **AS** row\_num

**FROM** Person **ORDER** **BY** Country;

In this statement, we have first specify the session variable **@row\_number** indicated by @prfix and set its value 0. Then, we have selected the data from the table Person and increases the value for variable @row\_number by one to each row.

MySQL DENSE\_RANK Function

Introduction to MySQL DENSE\_RANK function

The DENSE\_RANK() is a [window function](https://www.mysqltutorial.org/mysql-window-functions/) that assigns a rank to each row within a partition or result set with no gaps in ranking values.

The rank of a row is increased by one from the number of distinct rank values that come before the row.

Here’s the basic syntax of the DENSE\_RANK() function:

DENSE\_RANK() OVER (

PARTITION BY partition\_expression

ORDER BY sort\_expression [ASC|DESC]

)Code language: SQL (Structured Query Language) (sql)

In this syntax:

* First, the PARTITION BY clause divides the result sets produced by the FROM clause into partitions. The DENSE\_RANK() function is applied to each partition independently.
* Second, the ORDER BY  clause specifies the order of rows in each partition on which the DENSE\_RANK() function operates.

If a partition has two or more rows with the same rank value, each of these rows will be assigned the same rank.

Unlike the [RANK()](https://www.mysqltutorial.org/mysql-window-functions/mysql-rank-function/) function, the DENSE\_RANK() function always returns consecutive rank values.

Suppose you have a table t with some sample data as follows:

CREATE TABLE t (

val INT

);

INSERT INTO t(val)

VALUES(1),(2),(2),(3),(4),(4),(5);

SELECT \* FROM t;

The following statement uses the DENSE\_RANK() function to assign a rank to each row:

SELECT

val,

DENSE\_RANK() OVER (

ORDER BY val

) my\_rank

FROM t;

MySQL DENSE\_RANK() function example

We will use the following sales table for the demonstration:

CREATE TABLE sales(

sales\_employee VARCHAR(50) NOT NULL,

fiscal\_year INT NOT NULL,

sale DECIMAL(14,2) NOT NULL,

PRIMARY KEY(sales\_employee,fiscal\_year)

);

INSERT INTO sales(sales\_employee,fiscal\_year,sale)

VALUES('Bob',2016,100),

('Bob',2017,150),

('Bob',2018,200),

('Alice',2016,150),

('Alice',2017,100),

('Alice',2018,200),

('John',2016,200),

('John',2017,150),

('John',2018,250);

SELECT \* FROM sales;

The following statement uses the DENSE\_RANK() function to rank the sales employees by sale amount.

SELECT

sales\_employee,

fiscal\_year,

sale,

DENSE\_RANK() OVER (

PARTITION BY fiscal\_year

ORDER BY sale DESC

) sales\_rank

FROM

sales;

In this example:

* First, the PARTITION BY clause divided the result sets into partitions using fiscal year.
* Second, the ORDER BY clause specified the order of the sales employees by sales in descending order.
* Third, the DENSE\_RANK() function is applied to each partition with the order of the rows specified by the ORDER BY clause.

Introduction to MySQL FIRST\_VALUE() function

The FIRST\_VALUE() is a [window function](https://www.mysqltutorial.org/mysql-window-functions/) that allows you to select the first row of a window frame, partition, or result set.

MySQL FIRST\_VALUE() function examples

The following statements [create a new table](https://www.mysqltutorial.org/mysql-basics/mysql-create-table/) named overtime and [insert](https://www.mysqltutorial.org/mysql-basics/mysql-insert/) sample data for the demonstration:

CREATE TABLE overtime (

employee\_name VARCHAR(50) NOT NULL,

department VARCHAR(50) NOT NULL,

hours INT NOT NULL,

PRIMARY KEY (employee\_name , department)

);

INSERT INTO overtime(employee\_name, department, hours)

VALUES('Diane Murphy','Accounting',37),

('Mary Patterson','Accounting',74),

('Jeff Firrelli','Accounting',40),

('William Patterson','Finance',58),

('Gerard Bondur','Finance',47),

('Anthony Bow','Finance',66),

('Leslie Jennings','IT',90),

('Leslie Thompson','IT',88),

('Julie Firrelli','Sales',81),

('Steve Patterson','Sales',29),

('Foon Yue Tseng','Sales',65),

('George Vanauf','Marketing',89),

('Loui Bondur','Marketing',49),

('Gerard Hernandez','Marketing',66),

('Pamela Castillo','SCM',96),

('Larry Bott','SCM',100),

('Barry Jones','SCM',65);

1) Using MySQL FIRST\_VALUE() function over the whole query result set example

The following statement gets the employee’s name, overtime, and the employee who has the least overtime:

SELECT

employee\_name,

hours,

FIRST\_VALUE(employee\_name) OVER (

ORDER BY hours

) least\_over\_time

FROM

overtime;

In this example, the ORDER BY clause ordered the rows in the result set by hours and the FIRST\_VALUE() picked the first row indicating the employee who had the least overtime.

2) Using MySQL FIRST\_VALUE() function with partitions example

The following statement finds the employee who has the least overtime in every department:

SELECT

employee\_name,

department,

hours,

FIRST\_VALUE(employee\_name) OVER (

PARTITION BY department

ORDER BY hours

) least\_over\_time

FROM

overtime;

In this example:

* First, the PARTITION BY clause divides the employees into partitions by departments. In other words, each partition consists of employees who belong to the same department.
* Second, the ORDER BY clause specifies the order of rows in each partition.
* Third, the FIRST\_VALUE() operates on each partition sorted by the hours. It returns the first row in each partition which is the employee who has the least overtime within the department.

MySQL LAG Function

**Summary**: in this tutorial, you will learn how to use the MySQL LAG() function to access data of a previous row from the current row in the same result set.

Introduction to MySQL LAG() function

The LAG() function is a [window function](https://www.mysqltutorial.org/mysql-window-functions/) that allows you to access data from a previous row in a result set from the current row without using a [self-join](https://www.mysqltutorial.org/mysql-basics/mysql-self-join/).

MySQL LAG() function examples

Let’s take some examples of using the LAG() function. We’ll use the following sales table for demonstration:

CREATE TABLE sales(

sales\_employee VARCHAR(50) NOT NULL,

fiscal\_year INT NOT NULL,

sale DECIMAL(14,2) NOT NULL,

PRIMARY KEY(sales\_employee,fiscal\_year)

);

INSERT INTO sales(sales\_employee,fiscal\_year,sale)

VALUES('Bob',2016,100),

('Bob',2017,150),

('Bob',2018,200),

('Alice',2016,150),

('Alice',2017,100),

('Alice',2018,200),

('John',2016,200),

('John',2017,150),

('John',2018,250);

SELECT \* FROM sales;

1) Basic MySQL LAG() function example

The following query uses the LAG function to compare the sales of a year with the previous one:

SELECT

sales\_employee,

fiscal\_year,

sale,

LAG(sale, 1 , 0) OVER (

PARTITION BY sales\_employee

ORDER BY fiscal\_year

) 'previous year sale'

FROM

sales;

How it works.

The LAG() function divides the rows in the sales table by sales employees into partitions. Since we have three sales employees, it creates three partitions:

PARTITION BY sales\_employee

In each partition, the LAG() function sorts the rows by fiscal years. Hence, the rows in each partition are sorted by fiscal year column:

ORDER BY fiscal\_year

For each row in a partition, the LAG() function returns the value in the sale column of the previous row. If there is no previous row, it returns 0 as we specify in the default\_value argument of the LAG() function:

LAG(sale, 1 , 0)

As a result, the LAG() function returns the sales of the previous year (or zero) from the current row.

2) Using multiple LAG functions

To compare the sales of the “current” year with the previous year, you can use an additional LAG() function as follows:

SELECT

sales\_employee,

fiscal\_year,

sale,

LAG(sale, 1, 0) OVER (

PARTITION BY sales\_employee

ORDER BY fiscal\_year

) AS previous\_year\_sale,

sale - LAG(sale, 1, 0) OVER (

PARTITION BY sales\_employee

ORDER BY fiscal\_year

) AS SALES\_previous\_year

FROM

sales;

MySQL LAST\_VALUE Function

**Summary**: in this tutorial, you will learn how to use the MySQL LAST\_VALUE() function to return the last row in an ordered set of rows.

MySQL LAST\_VALUE() Function Overview

The LAST\_VALUE() function is a [window function](https://www.mysqltutorial.org/mysql-window-functions/) that allows you to select the last row in an ordered set of rows.

MySQL LAST\_VALUE() function examples

Let’s set up a sample table for demonstration.

The following is the script to create the overtime table and populate data into the table.

CREATE TABLE overtime (

employee\_name VARCHAR(50) NOT NULL,

department VARCHAR(50) NOT NULL,

hours INT NOT NULL,

PRIMARY KEY (employee\_name , department)

);

INSERT INTO overtime(employee\_name, department, hours)

VALUES('Diane Murphy','Accounting',37),

('Mary Patterson','Accounting',74),

('Jeff Firrelli','Accounting',40),

('William Patterson','Finance',58),

('Gerard Bondur','Finance',47),

('Anthony Bow','Finance',66),

('Leslie Jennings','IT',90),

('Leslie Thompson','IT',88),

('Julie Firrelli','Sales',81),

('Steve Patterson','Sales',29),

('Foon Yue Tseng','Sales',65),

('George Vanauf','Marketing',89),

('Loui Bondur','Marketing',49),

('Gerard Hernandez','Marketing',66),

('Pamela Castillo','SCM',96),

('Larry Bott','SCM',100),

('Barry Jones','SCM',65);

1) Using MySQL LAST\_VALUE() function over the whole query result example

The following statement gets the employee name, overtime, and the employee who has the highest overtime:

SELECT

employee\_name,

hours,

LAST\_VALUE(employee\_name) OVER (

ORDER BY hours

RANGE BETWEEN

UNBOUNDED PRECEDING AND

UNBOUNDED FOLLOWING

) highest\_overtime\_employee

FROM

overtime;

2) Using MySQL LAST\_VALUE() function over partitions example

The following statement finds the employee who has the highest overtime in each department:

SELECT

employee\_name,

department,

hours,

LAST\_VALUE(employee\_name) OVER (

PARTITION BY department

ORDER BY hours

RANGE BETWEEN

UNBOUNDED PRECEDING AND

UNBOUNDED FOLLOWING

) most\_overtime\_employee

FROM

overtime;

In this example, first, the PARTITION BY clause divided the employees by departments. Then, the ORDER BY clause orders the employees in each department by overtime from low to high.

The frame specification in this case is the whole partition. As a result, the LAST\_VALUE() function picked the last row in each partition which was the employee who had the highest overtime.

MySQL LEAD Function

**Summary**: in this tutorial, you will learn how to use the MySQL LEAD() function to access data from the next row in the result set

Introduction to MySQL LEAD() function

The LEAD() function is a [window function](https://www.mysqltutorial.org/mysql-window-functions/) that allows you to access data from the next row in a result set

MySQL LEAD() function examples

Let’s take some examples of using the LEAD() function. We’ll use the following sales table for the demonstration:

CREATE TABLE sales(

sales\_employee VARCHAR(50) NOT NULL,

fiscal\_year INT NOT NULL,

sale DECIMAL(14,2) NOT NULL,

PRIMARY KEY(sales\_employee,fiscal\_year)

);

INSERT INTO sales(sales\_employee,fiscal\_year,sale)

VALUES('Bob',2016,100),

('Bob',2017,150),

('Bob',2018,200),

('Alice',2016,150),

('Alice',2017,100),

('Alice',2018,200),

('John',2016,200),

('John',2017,150),

('John',2018,250);

SELECT \* FROM sales;

1) Basic MySQL LEAD() function example

The following example uses the LEAD() function to pull the sales of the next row into the current row:

SELECT

sales\_employee,

fiscal\_year,

sale,

LEAD(sale) OVER (

PARTITION BY sales\_employee

ORDER BY fiscal\_year

) AS next\_year\_sale

FROM

sales;

How it works.

First, the PARTITION BY sales\_employee divides the rows in the table into three partitions, each representing the sales of one sales employee.

Second, the ORDER BY fiscal\_year sorts the rows in each partition by fiscal year in ascending order.

Therefore, the LEAD() function returns the sales of the next year from the current year for each sales employee. If there is no next year such as for the year 2018, the LEAD() function returns NULL.

2) Using multiple LEAD() functions in a query

The following query uses multiple LEAD() functions to calculate the sales vs. next year’s in percentage:

SELECT

sales\_employee,

fiscal\_year,

sale,

LEAD(sale) OVER (

PARTITION BY sales\_employee

ORDER BY fiscal\_year

) AS next\_year\_sale,

ROUND(sale - (LEAD(sale) OVER (

PARTITION BY sales\_employee

ORDER BY fiscal\_year)) / sale \* 100, 2) 'vs\_next\_year (%)'

FROM

sales;

In this example, we added a new column named vs\_next\_year (%). This column calculates the percentage change from the current year’s sales to the next year’s sales using the formula:

**( current year's sale - next year's sale ) x 100 / next year's sale**

**vs\_next\_year (%): This column calculates the percentage difference between the current year’s sales and the next year’s sales. The formula used is:**

This expression calculates the percentage change and assigns it to the vs\_next\_year (%) column.

MySQL NTH\_VALUE Function

**Summary**: in this tutorial, you will learn how to use the NTH\_VALUE() function to get a value from the Nth row in a result set.

Introduction to MySQL NTH\_VALUE() function

The NTH\_VALUE() is a [window function](https://www.mysqltutorial.org/mysql-window-functions/) that allows you to get a value from the Nth row in an ordered set of rows.

The NTH\_VALUE() function returns the value of expression from the Nth row of the window frame. If that Nth row does not exist, the function returns [NULL](https://www.mysqltutorial.org/mysql-basics/mysql-null/). N must be a positive integer e.g., 1, 2, and 3.

The FROM FIRST instructs the NTH\_VALUE() function to begin calculation at the first row of the window frame.

Note that SQL standard supports both FROM FIRST and FROM LAST. However, MySQL only supports FROM FIRST. If you want to simulate the effect of FROM LAST, then you can use the ORDER BY in the over\_clause to sort the result set in reverse order.

MySQL NTH\_VALUE() function examples

We will [create a new table](https://www.mysqltutorial.org/mysql-basics/mysql-create-table/) named basic\_pay for the demonstration.

CREATE TABLE basic\_pays(

employee\_name VARCHAR(50) NOT NULL,

department VARCHAR(50) NOT NULL,

salary INT NOT NULL,

PRIMARY KEY (employee\_name , department)

);

INSERT INTO

basic\_pays(employee\_name,

department,

salary)

VALUES

('Diane Murphy','Accounting',8435),

('Mary Patterson','Accounting',9998),

('Jeff Firrelli','Accounting',8992),

('William Patterson','Accounting',8870),

('Gerard Bondur','Accounting',11472),

('Anthony Bow','Accounting',6627),

('Leslie Jennings','IT',8113),

('Leslie Thompson','IT',5186),

('Julie Firrelli','Sales',9181),

('Steve Patterson','Sales',9441),

('Foon Yue Tseng','Sales',6660),

('George Vanauf','Sales',10563),

('Loui Bondur','SCM',10449),

('Gerard Hernandez','SCM',6949),

('Pamela Castillo','SCM',11303),

('Larry Bott','SCM',11798),

('Barry Jones','SCM',10586);

1) Using MySQL NTH\_VALUE() function over the result set

The following statement uses the NTH\_VALUE() function to find the employee who has the second highest salary :

SELECT

employee\_name,

salary,

NTH\_VALUE(employee\_name, 2) OVER (

ORDER BY salary DESC

) second\_highest\_salary

FROM

basic\_pays;

2) Using MySQL NTH\_VALUE() over partitions example

The following query finds the employee who has the second highest salary in every department:

SELECT

employee\_name,

department,

salary,

NTH\_VALUE(employee\_name, 2) OVER (

PARTITION BY department

ORDER BY salary DESC

RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

) second\_highest\_salary

FROM

basic\_pays;

In this query, we added the PARTITION BY clause to divide the employees by department. Then the NTH\_VALUE() function is applied to each partition independently.

**RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING**

* This clause defines the window frame over which the function operates:
  + **RANGE**: Refers to a set of rows defined by a range of values. In this context, it considers all rows with a range of salary values within the partition.
  + **UNBOUNDED PRECEDING**: Indicates that the frame starts from the first row in the partition.
  + **UNBOUNDED FOLLOWING**: Indicates that the frame ends at the last row in the partition.

### **Why Use RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING?**

This frame definition ensures that the window function has access to all rows in the partition. It allows the function to consistently return the second-highest salary even if the order of the rows changes due to sorting by salary DESC.

In this context, using this frame guarantees that the second-highest value is correctly identified across all rows within each department.

MySQL NTILE Function

**Summary**: in this tutorial, you will learn how to use the MySQL NTILE() function to divide rows into a specified number of groups.

Introduction to MySQL NTILE() function

The MySQL NTILE() function divides rows in a sorted partition into a specific number of groups. Each group is assigned a bucket number starting at one. For each row, the NTILE() function returns a bucket number representing the group to which the row belongs.

See the following table that stores nine integers from one to nine:

CREATE TABLE t (

val INT NOT NULL

);

INSERT INTO t(val)

VALUES(1),(2),(3),(4),(5),(6),(7),(8),(9);

SELECT \* FROM t;

If you use the NTILE() function to divide nine rows into four groups, you will end up at the first group with three rows and the other three groups with four rows.

See the following demonstration:

SELECT

val,

NTILE (4) OVER (

ORDER BY val

) bucket\_no

FROM

t;

Let’s change the number of groups from four to three as shown in the following query:

SELECT

val,

NTILE (3) OVER (

ORDER BY val

) bucket\_no

FROM

t;