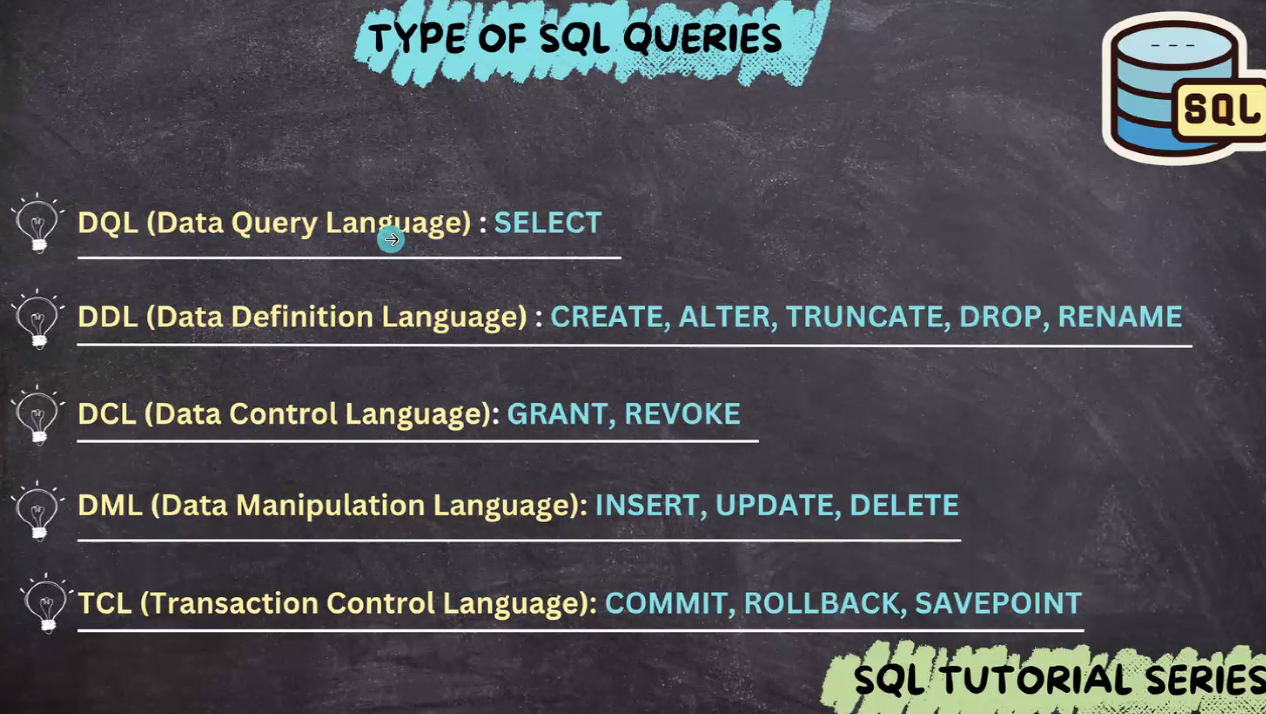
**In this session**

You will be introduced to the concept of windowing functions. You will learn about the 'over'and 'partition'clauses used to implement windowing. After going through this session, you should be able to use window functions such as rank(), dense\_rank() and percent\_rank() in your queries.

You will also be introduced to the concept of named windows. You will learn about framesand how they move within a window. Next, you will learn about the various applications of windowing, including one to calculate an element known as a moving average. Finally, you will learn about the 'lead' and 'lag' functions that are used to fetch data from succeeding and preceding rows, respectively.

<https://dev.mysql.com/doc/refman/8.4/en/window-function-descriptions.html>



**Common Window Functions in MySQL**

**Now we will be learning different Windows Functions in MySQL:**

**1. ROW\_NUMBER()**

**This function is used to assigns a unique sequential integer to rows within a partition**

**Example:**

**SELECT  
 employee\_id,  
 department\_id,  
 salary,  
 ROW\_NUMBER() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS row\_num  
FROM employees;**

**Output:**

| **employee\_id** | **department\_id** | **salary** | **row\_num** |
| --- | --- | --- | --- |
| **101** | **1** | **90000** | **1** |
| **102** | **1** | **85000** | **2** |
| **103** | **2** | **95000** | **1** |
| **104** | **2** | **70000** | **2** |

**2. RANK() and DENSE\_RANK()**

**RANK(): Rank of the current row within its partition, with gaps**

**DENSE\_RANK(): Rank of the current row within its partition, without gaps**

**PERCENT\_RANK(): Percentage rank value, which always lies between 0 and 1**

**The use of this function id to leave gaps in the ranking when they are ties and also assigns a ranking within a partition.**

**Example:**

**SELECT  
 employee\_id,  
 department\_id,  
 salary,  
 RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank,  
 DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS dense\_rank  
FROM employees;**

**Output:**

| **employee\_id** | **department\_id** | **salary** | **rank** | **dense\_rank** |
| --- | --- | --- | --- | --- |
| **101** | **1** | **90000** | **1** | **1** |
| **102** | **1** | **85000** | **2** | **2** |
| **103** | **1** | **85000** | **2** | **2** |
| **104** | **1** | **75000** | **4** | **3** |

**3. SUM()**

**The use of this function is to calculate the sum of the columns with in a window.**

**Example:**

**SELECT  
 employee\_id,  
 salary,  
 SUM(salary) OVER (ORDER BY employee\_id) AS cumulative\_salary  
FROM employees;**

**Output:**

| **employee\_id** | **salary** | **cumulative\_salary** |
| --- | --- | --- |
| **101** | **50000** | **50000** |
| **102** | **60000** | **110000** |
| **103** | **70000** | **180000** |
| **104** | **80000** | **260000** |

**4. AVG()**

**This function is responsible for the moving average of the across the set of rows.**

**Example:**

**SELECT  
 employee\_id,  
 salary,  
 AVG(salary) OVER (ORDER BY employee\_id ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving\_avg  
FROM employees;**

**Output:**

| **employee\_id** | **salary** | **moving\_avg** |
| --- | --- | --- |
| **101** | **50000** | **50000.0** |
| **102** | **60000** | **55000.0** |
| **103** | **70000** | **60000.0** |
| **104** | **80000** | **70000.0** |

**5. LEAD() and LAG()**

**LEAD() and LAG() functions allow you to access subsequent or previous rows' data without the need for self-joins.**

**Example:**

**SELECT  
 employee\_id,  
 salary,  
 LEAD(salary, 1) OVER (ORDER BY employee\_id) AS next\_salary,  
 LAG(salary, 1) OVER (ORDER BY employee\_id) AS previous\_salary  
FROM employees;**

**Output:**

| **employee\_id** | **salary** | **next\_salary** | **previous\_salary** |
| --- | --- | --- | --- |
| **101** | **50000** | **60000** | **NULL** |
| **102** | **60000** | **70000** | **50000** |
| **103** | **70000** | **80000** | **60000** |
| **104** | **80000** | **NULL** | **70000** |

**MySQL Window Functions with Different Clauses**

**1. Using PARTITION BY**

**The ORDER BY in the OVER() clause determines the order of processing of the rows. This is a very key part of window functions' behavior and particularly so with functions like ROW\_NUMBER(), RANK(), and when implementing cumulative calculations.**

**Example:**

**SELECT  
 employee\_id,  
 salary,  
 SUM(salary) OVER (ORDER BY employee\_id) AS cumulative\_salary  
FROM employees;**

**Output:**

| **employee\_id** | **salary** | **cumulative\_salary** |
| --- | --- | --- |
| **101** | **50000** | **50000** |
| **102** | **60000** | **110000** |
| **103** | **70000** | **180000** |
| **104** | **80000** | **260000** |

**2. Using PARTITION BY**

**The PARTITION BY**[**clause**](https://www.geeksforgeeks.org/sql-with-clause/)**divides the result set produced into partitions on which the window function works independently. This is quite useful in doing some calculations on certain groups.**

**SELECT  
 employee\_id,  
 department\_id,  
 salary,  
 SUM(salary) OVER (PARTITION BY department\_id ORDER BY salary DESC) AS dept\_cumulative\_salary  
FROM employees;**

**Output:**

| **employee\_id** | **department\_id** | **salary** | **dept\_cumulative\_salary** |
| --- | --- | --- | --- |
| **105** | **1** | **90000** | **90000** |
| **104** | **1** | **85000** | **175000** |
| **103** | **1** | **75000** | **250000** |
| **108** | **2** | **95000** | **95000** |
| **107** | **2** | **70000** | **165000** |
| **106** | **2** | **60000** | **225000** |

**Window Frames: ROWS and RANGE**

**The window frame also defines which set of rows is included when calling a window function.**

**ROWS: This defines the number of rows in the frame.**

**RANGE: Indicates the range of values to be included.**

**Example of ROWS**

**SELECT  
 employee\_id,  
 salary,  
 SUM(salary) OVER (ORDER BY employee\_id ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS sum\_salary  
FROM employees;**

**Output:**

| **employee\_id** | **salary** | **sum\_salary** |
| --- | --- | --- |
| **101** | **50000** | **50000** |
| **102** | **60000** | **110000** |
| **103** | **70000** | **180000** |
| **104** | **80000** | **210000** |

**Example of RANGE**

**SELECT  
 employee\_id,  
 salary,  
 SUM(salary) OVER (ORDER BY salary RANGE BETWEEN 1000 PRECEDING AND CURRENT ROW) AS sum\_salary  
FROM employees;**

**Output:**

| **employee\_id** | **salary** | **sum\_salary** |
| --- | --- | --- |
| **101** | **50000** | **50000** |
| **102** | **60000** | **110000** |
| **103** | **70000** | **180000** |
| **104** | **80000** | **260000** |

**Using Frame Clauses**

**If we talk about the frames clauses then it indicates that which subset of rows the windows function applies the calculation. Let's see an example of it.**

**Example:**

**SELECT  
employee\_id,  
salary,  
SUM(salary) OVER (ORDER BY employee\_id ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS sum\_salary  
FROM employees;**

**Output:**

| **employee\_id** | **salary** | **sum\_salary** |
| --- | --- | --- |
| **101** | **50000** | **50000** |
| **102** | **60000** | **110000** |
| **103** | **70000** | **180000** |
| **104** | **80000** | **210000** |

**Clustered vs non-clustered indexing:**The major differences between clustered and non-clustered indexing are summarised in the table given below.

| **Clustered Index** | **Non-Clustered Index** |
| --- | --- |
| 1. This is mostly the primary key of the table. | 1. It is a combination of one or more columns of the table. |
| 2. It is present within the table. | 2. The unique list of keys is present outside the table. |
| 3. It does not require a separate mapping. | 3. The external table points to different sections of the main table. |
| 4. It is relatively faster. | 4. It is relatively slower. |

**The problem statement, the required columns and the required tables introduced in this video can be summarised as follows:**

**Problem statement: Extract the required details of the customers who have not placed an order yet.**

**Expected columns:**The columns that are required as the output are as follows:

* 'cust\_id'
* 'cust\_name'
* 'city'
* 'state'
* 'customer\_segment'
* A flag to indicate that there is another customer with the exact same name and city but a different customer ID.

**Tables:**The tables that are required for solving this problem are as follows:

* 'cust\_dimen'
* 'market\_fact\_full'

The ERD for the 'market star' schema is given below for your reference.