SQL Tutorial: A Guide to SQL Queries

Pratik T · Follow
16 min read · Sep 17, 2024

158 2

Structured Query Language (SQL) is the language used to interact with databases. With SQL, you can retrieve, manipulate, and manage data stored in relational databases. This tutorial will cover the fundamental aspects of SQL with examples to get you started.

TUTORIAL INTRO

In this tutorial, we will explore:

- Basic SQL structure
- Querying data using `SELECT`
- Filtering data with `WHERE`, `AND`, `OR`, `NOT`
- Using operators like `BETWEEN`, `IN`, and `LIKE`
- Sorting results using `ORDER BY`
- Aggregating data with `SUM`, `AVG`, and `COUNT`
- Grouping data with `GROUP BY` and filtering grouped data with `HAVING`
- Using `DISTINCT` to find unique values

- Arithmetic operations and math functions in SQL
- Handling `NULL` values
- Using `CASE` statements for conditional logic
- Working with `JOINS` to combine data from multiple tables
- Date manipulation with date functions
- Writing efficient queries using `CTEs` and `subqueries`
- Working with window functions for row-wise operations
- Ranking, lead, and lag functions
- Self-joins and advanced query techniques like `UNION`
- Understanding the execution order of SQL queries
- Pivots and string manipulations

Let’s dive into the core SQL concepts.

SQL SELECT

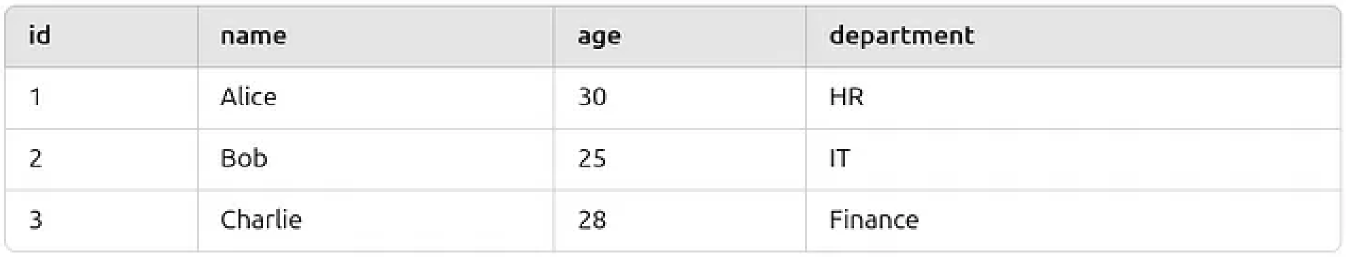
The `SELECT` statement is used to retrieve data from a database. You can choose specific columns or all columns in a table.

Syntax:

```
SELECT column1, column2, … FROM table_name;
```

Example:

Suppose we have a table called `employees`:

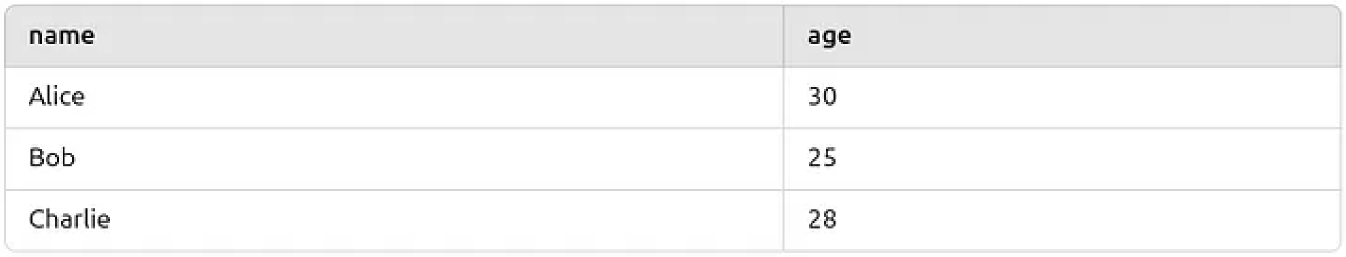


id	name	age	department
1	Alice	30	HR
2	Bob	25	IT
3	Charlie	28	Finance

To select the `name` and `age` columns:

```
SELECT name, age FROM employees;
```

Result:

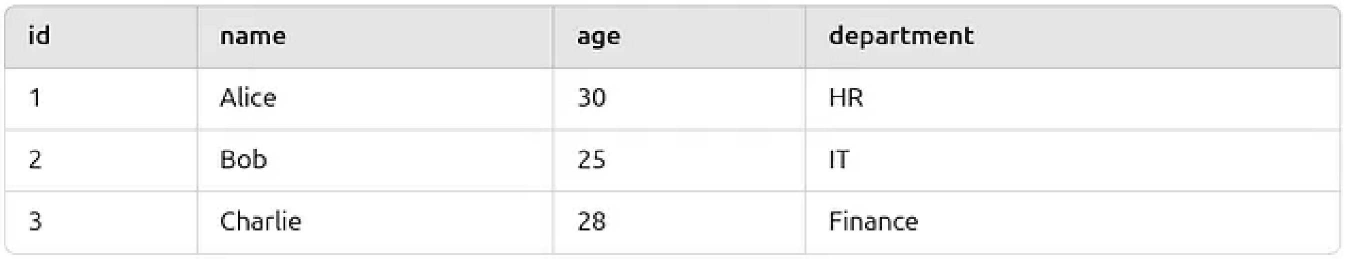


name	age
Alice	30
Bob	25
Charlie	28

To select all columns:

```
SELECT * FROM employees;
```

Result:



id	name	age	department
1	Alice	30	HR
2	Bob	25	IT
3	Charlie	28	Finance

SQL WHERE

The `WHERE` clause is used to filter records based on conditions. It helps you fetch only the rows that match specific criteria.

Syntax:

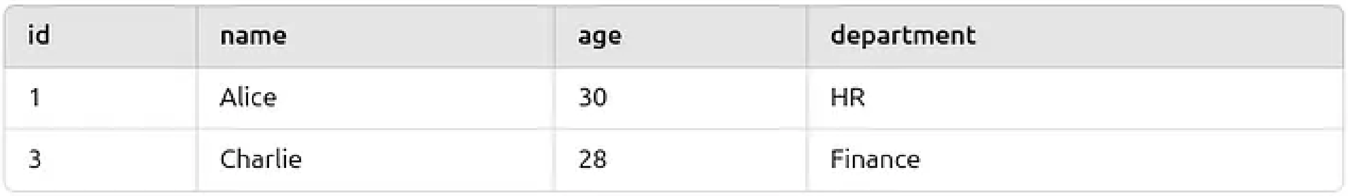
```
SELECT column1, column2, … FROM table_name WHERE condition;
```

Example:

To select employees who are older than 25:

```
SELECT * FROM employees WHERE age > 25;
```

Result:



id	name	age	department
1	Alice	30	HR
3	Charlie	28	Finance

AND, OR, NOT

These operators allow you to combine multiple conditions in your `WHERE` clause.

- `AND`: Returns records when all conditions are true.

- `OR`: Returns records when at least one condition is true.

- `NOT`: Negates a condition.

Syntax:

```
SELECT column1, column2, … FROM table_name WHERE condition1 AND/OR/NOT condition2;
```

Examples:

AND: Select employees who are older than 25 AND work in the HR department:

```
SELECT * FROM employees WHERE age > 25 AND department = 'HR';
```

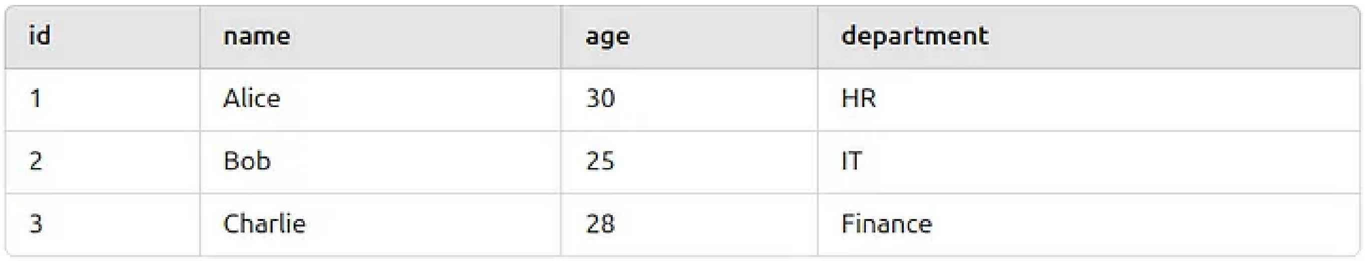
Result:

id	name	age	department
1	Alice	30	HR

OR: Select employees who are older than 25 OR work in the IT department:

```
SELECT * FROM employees WHERE age > 25 OR department = 'IT';
```

Result:

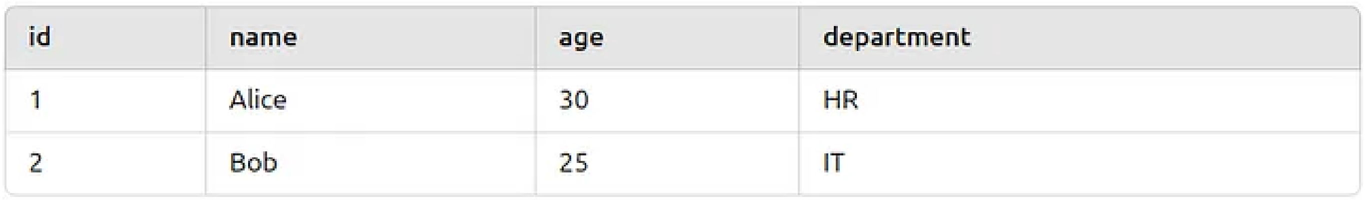


id	name	age	department
1	Alice	30	HR
2	Bob	25	IT
3	Charlie	28	Finance

NOT: Select employees who are NOT in the Finance department:

```
SELECT * FROM employees WHERE NOT department = 'Finance';
```

Result:



id	name	age	department
1	Alice	30	HR
2	Bob	25	IT

SQL BETWEEN

The `BETWEEN` operator selects values within a range. It is inclusive of the boundary values.

Syntax:

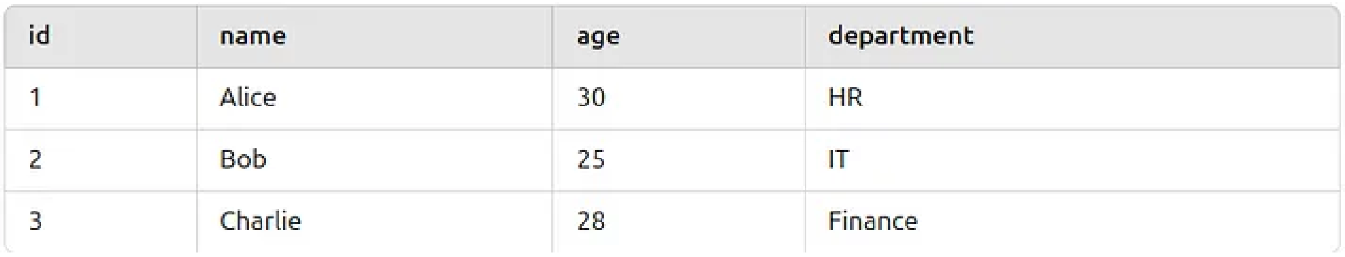
```
SELECT column1, column2, … FROM table_name WHERE column_name BETWEEN value1 AND value
```

Example:

Select employees aged between 25 and 30:

```
SELECT * FROM employees WHERE age BETWEEN 25 AND 30;
```

Result:



id	name	age	department
1	Alice	30	HR
2	Bob	25	IT
3	Charlie	28	Finance

SQL IN

The `IN` operator allows you to specify multiple values in a `WHERE` clause. It's useful when checking against multiple possible values.

Syntax:

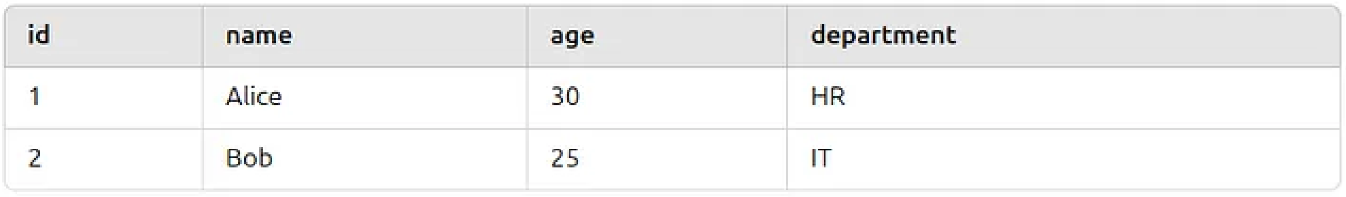
```
SELECT column1, column2, … FROM table_name WHERE column_name IN (value1, value2, …);
```

Example:

Select employees who work in the IT or HR departments:

```
SELECT * FROM employees WHERE department IN ('IT', 'HR');
```

Result:



id	name	age	department
1	Alice	30	HR
2	Bob	25	IT

SQL LIKE

The `LIKE` operator is used in a `WHERE` clause to search for a specified pattern in a column. You can use `%` as a wildcard for any number of characters and `_` for a single character.

Syntax:

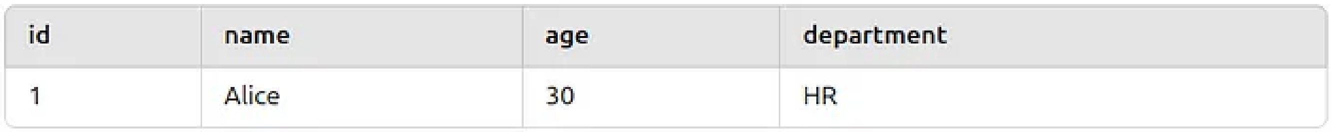
```
SELECT column1, column2, … FROM table_name WHERE column_name LIKE pattern;
```

Example:

Select employees whose name starts with "A":

```
SELECT * FROM employees WHERE name LIKE 'A%';
```

Result:

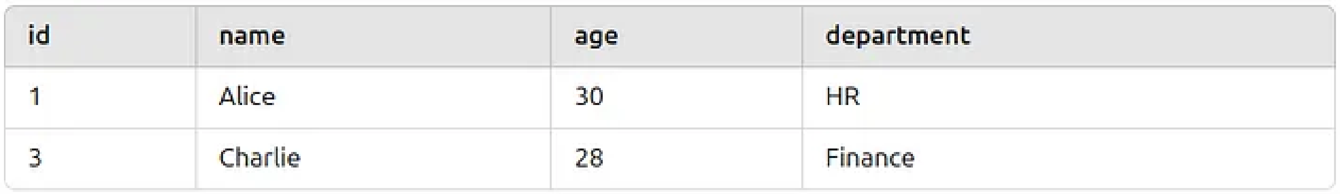


id	name	age	department
1	Alice	30	HR

Select employees whose name contains "li":

```
SELECT * FROM employees WHERE name LIKE '%li%';
```

Result:



id	name	age	department
1	Alice	30	HR
3	Charlie	28	Finance

SQL ORDER BY ↕️

The `ORDER BY` clause is used to sort the result set in ascending or descending order. By default, the `ORDER BY` sorts in ascending order.

Syntax:

```
SELECT column1, column2, … FROM table_name ORDER BY column_name [ASC|DESC];
```

- `ASC`: Ascending order (default).

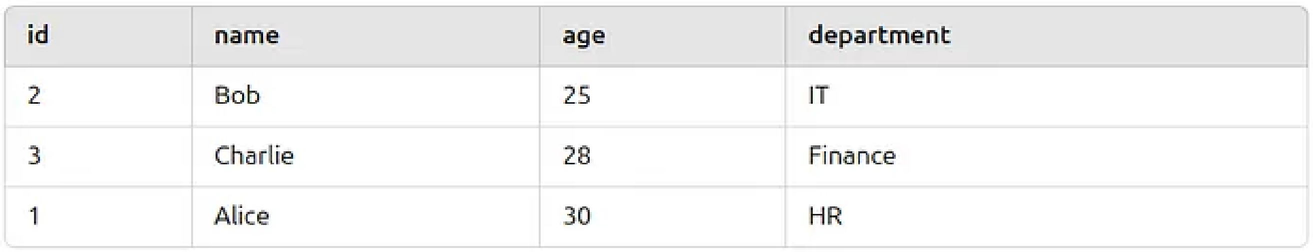
- `DESC`: Descending order.

Example:

Select all employees and sort them by age in ascending order:

```
SELECT * FROM employees ORDER BY age ASC;
```

Result:

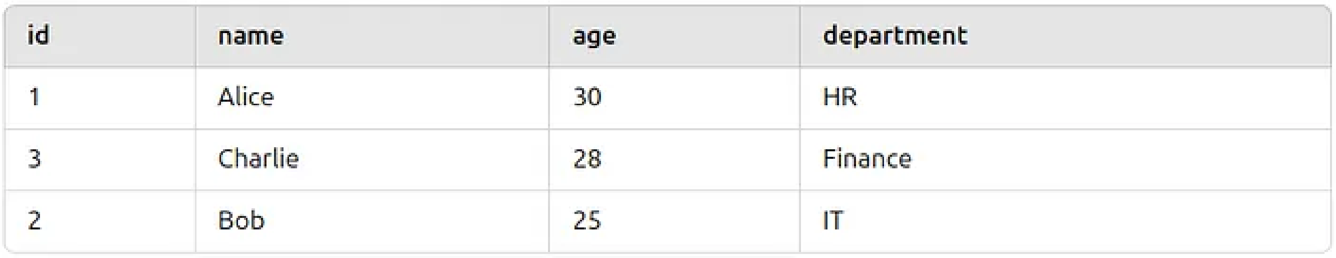


id	name	age	department
2	Bob	25	IT
3	Charlie	28	Finance
1	Alice	30	HR

Select all employees and sort them by age in descending order:

```
SELECT * FROM employees ORDER BY age DESC;
```

Result:



id	name	age	department
1	Alice	30	HR
3	Charlie	28	Finance
2	Bob	25	IT

INTERMEDIATE SQL

In this tutorial, we will explore:

- Aggregating data with `SUM`, `AVG`, and `COUNT`

- Grouping data with `GROUP BY` and filtering grouped data with `HAVING`

- Using `DISTINCT` to find unique values

- Arithmetic operations and math functions in SQL

- Handling `NULL` values

- Using `CASE` statements for conditional logic

- Working with `JOINS` to combine data from multiple tables

- Date manipulation with date functions

SUM, AVG, COUNT

These are aggregate functions used to perform calculations on a set of values.

- `SUM()`: Returns the total sum of a numeric column.

- `AVG()`: Returns the average of a numeric column.

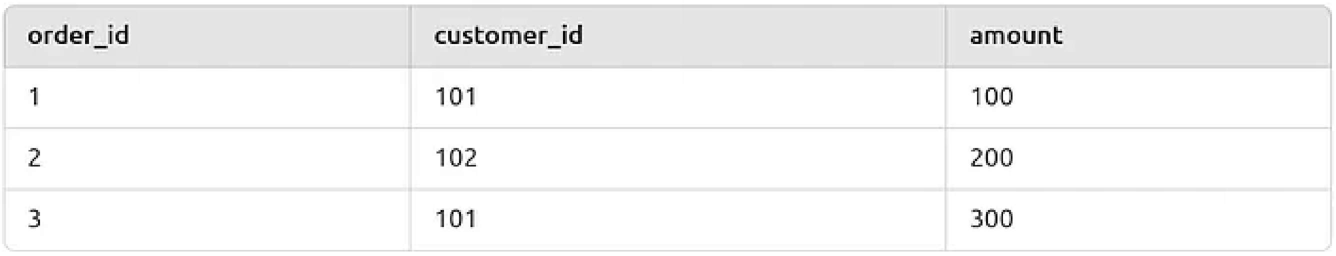
- `COUNT()`: Returns the number of rows or non-`NULL` values.

Syntax:

```
SELECT SUM(column_name), AVG(column_name), COUNT(column_name) FROM table_name;
```

Example:

Consider the `orders` table:

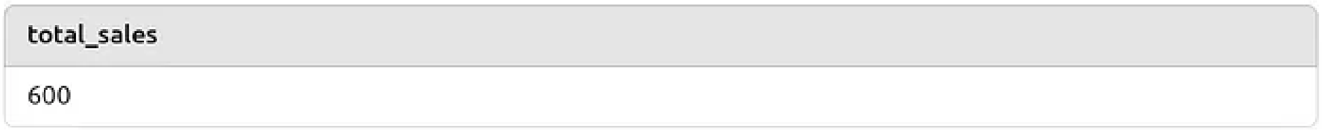


order_id	customer_id	amount
1	101	100
2	102	200
3	101	300

To calculate the total amount of orders:

```
SELECT SUM(amount) AS total_sales FROM orders;
```

Result:



total_sales
600

To calculate the average order amount:

```
SELECT AVG(amount) AS avg_sales FROM orders;
```

Result:

```
| avg_sales |
| - - - - - |
|   200     |
```

To count the number of orders:

```
SELECT COUNT(order_id) AS total_orders FROM orders;
```

Result:

```
| total_orders |
| - - - - - -  |
|     3        |
```

SQL GROUP BY

The `GROUP BY` statement is used to group rows that have the same values in specified columns. Often used with aggregate functions like `SUM()`, `COUNT()`, etc.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

Example:

To find the total order amount for each customer:

```
SELECT customer_id, SUM(amount) AS total_amount
FROM orders
GROUP BY customer_id;
```

Result:

```
| customer_id | total_amount |
| - - - - - - | - - - - - -  |
| 101         | 400          |
| 102         | 200          |
```

SQL HAVING

The `HAVING` clause is used to filter records after grouping. It’s similar to `WHERE`, but it operates on aggregated data.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

Example:

To find customers who have spent more than 300:

```
SELECT customer_id, SUM(amount) AS total_amount
FROM orders
GROUP BY customer_id
HAVING total_amount > 300;
```

Result:

```
| customer_id | total_amount |
| - - - - - - | - - - - - -  |
| 101         | 400          |
```

SQL DISTINCT

The `DISTINCT` keyword is used to return only unique values.

Syntax:

```
SELECT DISTINCT column1 FROM table_name;
```

Example:

To get unique customer IDs from the `orders` table:

```
SELECT DISTINCT customer_id FROM orders;
```

Result:

```
| customer_id |
| - - - - - - |
|    101      |
|    102      |
```

SQL ARITHMETIC

SQL allows arithmetic operations like addition, subtraction, multiplication, and division directly within queries.

Syntax:

```
SELECT column_name + 100, column_name - 50, column_name 2, column_name / 2 FROM table_
```

Example:

To add 10% tax to each order:

```
SELECT amount, amount*1.1 AS amount_with_tax FROM orders;
```

Result:

```
| amount | amount_with_tax |
| - - -  | - - - - - - - - |
|   100  |    110          |
|   200  |    220          |
|   300  |    330          |
```

MATH FUNCTIONS

SQL provides built-in math functions like `ROUND()`, `CEIL()`, `FLOOR()`, and `ABS()` to perform mathematical calculations.

Syntax:

```
SELECT ROUND(column_name), CEIL(column_name), FLOOR(column_name), ABS(column_name) FR
```

Example:

To round the order amounts to the nearest integer:

```
SELECT amount, ROUND(amount 1.1, 2) AS rounded_amount FROM orders;
```

Result:

```
| amount | rounded_amount  |
| - - -  | - - - - - - - - |
| 100    | 110.00          |
| 200    | 220.00          |
| 300    | 330.00          |
```

SQL DIVISION

SQL supports division (`/`) to perform calculations between columns or constants.

Syntax:

```
SELECT column1 / column2 FROM table_name;
```

Example:

To calculate the average order per customer (assuming no `GROUP BY`):

```
SELECT SUM(amount) / COUNT(DISTINCT customer_id) AS avg_per_customer FROM orders;
```

Result:

```
| avg_per_customer |
| - - - - - - - - -  |
|     300          |
```

SQL NULL

`NULL` represents missing or undefined data. SQL provides functions like `IS NULL`, `IS NOT NULL`, and `COALESCE()` to handle `NULL` values.

Syntax:

```
SELECT column1 FROM table_name WHERE column1 IS NULL;
```

Example:

To find orders where the amount is `NULL`:

```
SELECT * FROM orders WHERE amount IS NULL;
```

To handle `NULL` values using `COALESCE()`:

```
SELECT COALESCE(amount, 0) FROM orders; - returns 0 if amount is NULL
```

SQL CASE

The `CASE` statement is used for conditional logic in SQL queries. It acts like an `if-else` structure in SQL.

Syntax:

```
SELECT column1,
    CASE
        WHEN condition1 THEN result1
        WHEN condition2 THEN result2
        ELSE result3
    END AS alias_name
FROM table_name;
```

Example:

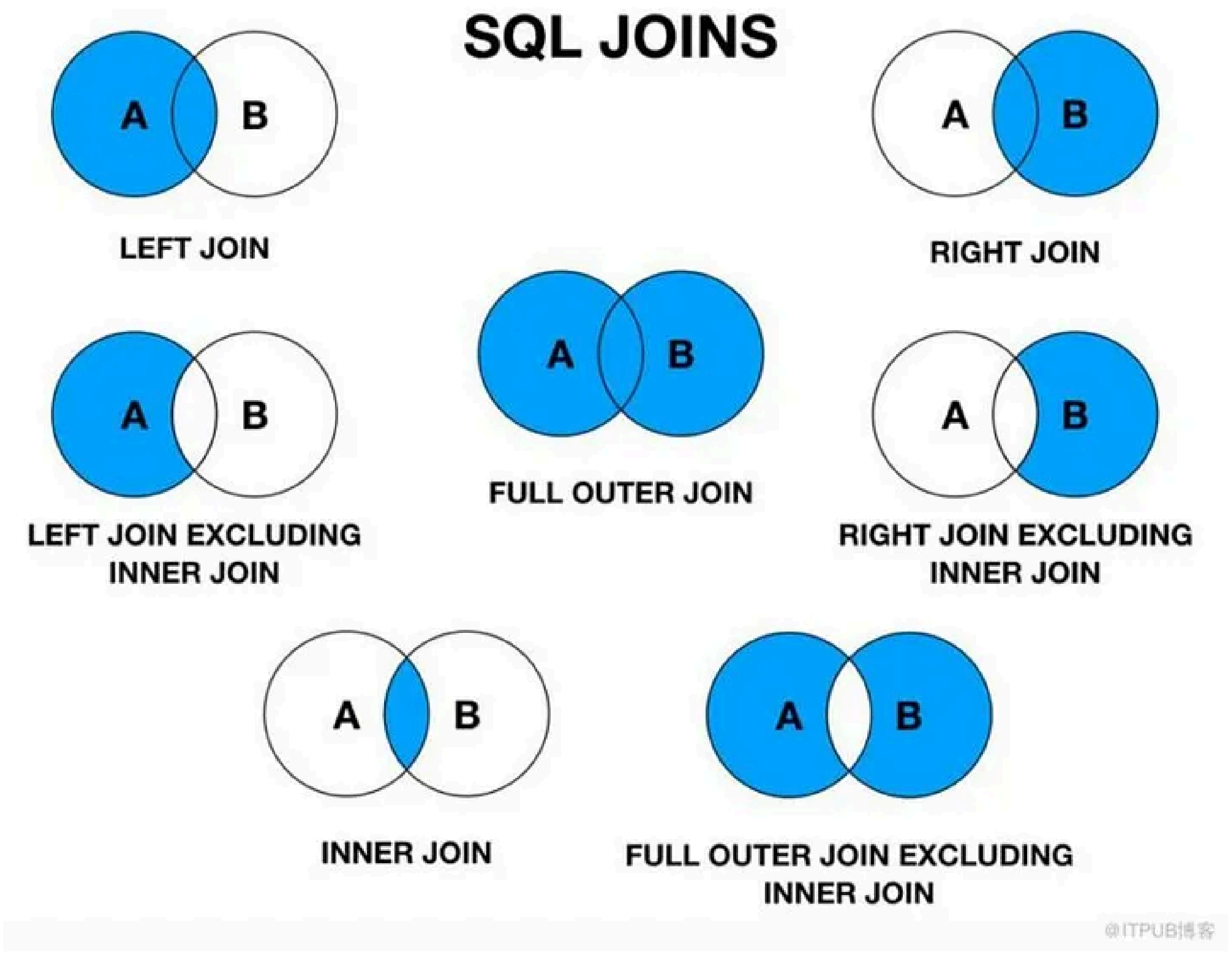
To categorize orders based on amount:

```
SELECT amount,
    CASE
        WHEN amount > 250 THEN 'High'
        WHEN amount > 100 THEN 'Medium'
        ELSE 'Low'
    END AS order_category
FROM orders;
```

Result:

```
| amount | order_category  |
| - - -  | - - - - - - - - |
| 100    | Low             |
| 200    | Medium          |
| 300    | High            |
```

SQL JOINS


SQL JOINS
A
B
LEFT JOIN
RIGHT JOIN
FULL OUTER JOIN
LEFT JOIN EXCLUDING INNER JOIN
RIGHT JOIN EXCLUDING INNER JOIN
INNER JOIN
FULL OUTER JOIN EXCLUDING INNER JOIN
@ITPUB博客

\`JOINS\` are used to combine rows from two or more tables based on a related column between them.

- \`INNER JOIN\`: Returns matching rows.

- \`LEFT JOIN\`: Returns all rows from the left table and matching rows from the right.

- \`RIGHT JOIN\`: Returns all rows from the right table and matching rows from the left.

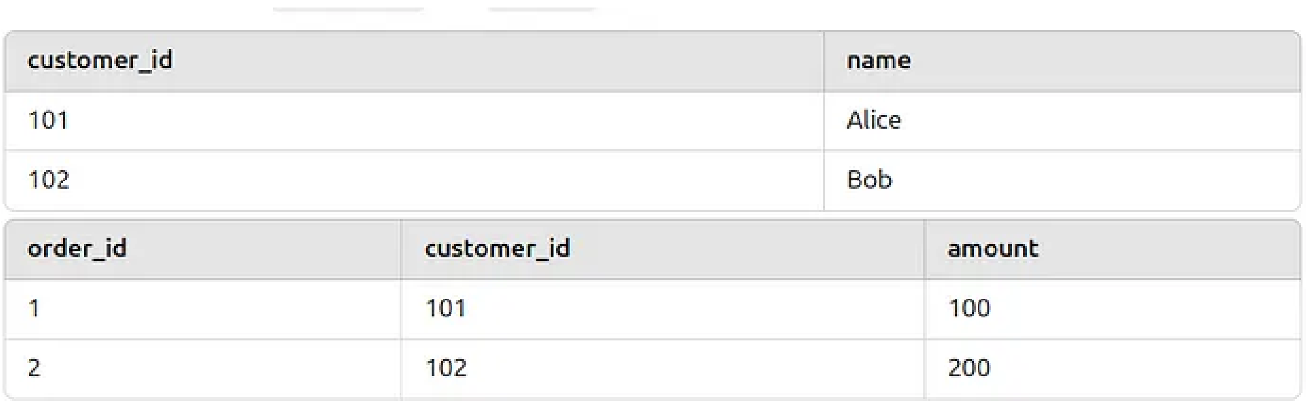
- \`FULL JOIN\`: Returns all rows when there is a match in either table.

Syntax:

```
SELECT columns FROM table1
JOIN table2 ON table1.common_column = table2.common_column;
```

Example:

Consider two tables `customers` and `orders`:



customer_id	name
101	Alice
102	Bob

order_id	customer_id	amount
1	101	100
2	102	200

To join `customers` with `orders`:

```
SELECT customers.name, orders.amount
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id;
```

Result:

name	amount
Alice	100
Bob	200

DATE FUNCTIONS

SQL provides functions to manipulate and format dates, like `NOW()`, `CURDATE()`, `DATEADD()`, and `DATEDIFF()`.

Syntax:

```
SELECT NOW(), CURDATE(), DATEDIFF(date1, date2) FROM table_name;
```

Example:

To get the current date and time:

```
SELECT NOW() AS current_date_time;
```

Result:

```
|   current_date_time   |
| - - - - - - - - - - - |
| 2024-09-12 10:15:30   |
```

To calculate the difference between two dates:

```
SELECT DATEDIFF('2024-09-12', '2024-09-01') AS days_difference;
```

Result:

```
| days_difference |
| - - - - - - - - - |
|       11        |
```

Advanced SQL

In this section, we will cover advanced topics that give you more control over data manipulation, analysis, and reporting. These concepts will help you tackle complex SQL problems with ease.

Advanced SQL concepts help with:

- Writing efficient queries using `CTEs` and `subqueries`

- Working with window functions for row-wise operations

- Ranking, lead, and lag functions

- Self-joins and advanced query techniques like `UNION`

- Writing clean, optimized SQL

- Understanding the execution order of SQL queries

- Pivots and string manipulations

CTE vs. SUBQUERY

Both Common Table Expressions (CTE) and subqueries allow you to create temporary result sets for use within a query. However, they have different use cases and syntax.

CTE (WITH Clause)

A CTE is a temporary result set that can be referenced within the main SQL query. It enhances readability and allows for recursive queries.

Syntax (CTE):

```
WITH cte_name AS (
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
SELECT column1, column2
FROM cte_name;
```

Example:

To calculate total sales for customers using a CTE:

```
WITH sales_summary AS (
    SELECT customer_id, SUM(amount) AS total_sales
    FROM orders
    GROUP BY customer_id
)
SELECT * FROM sales_summary WHERE total_sales > 500;
```

Subquery

A subquery is a query nested inside another query. It can be used to filter data or provide intermediate results.

Syntax (Subquery):

```
SELECT column1 FROM table_name
WHERE column1 = (SELECT MAX(column1) FROM table_name);
```

Example:

To find customers with more than 500 in total sales using a subquery:

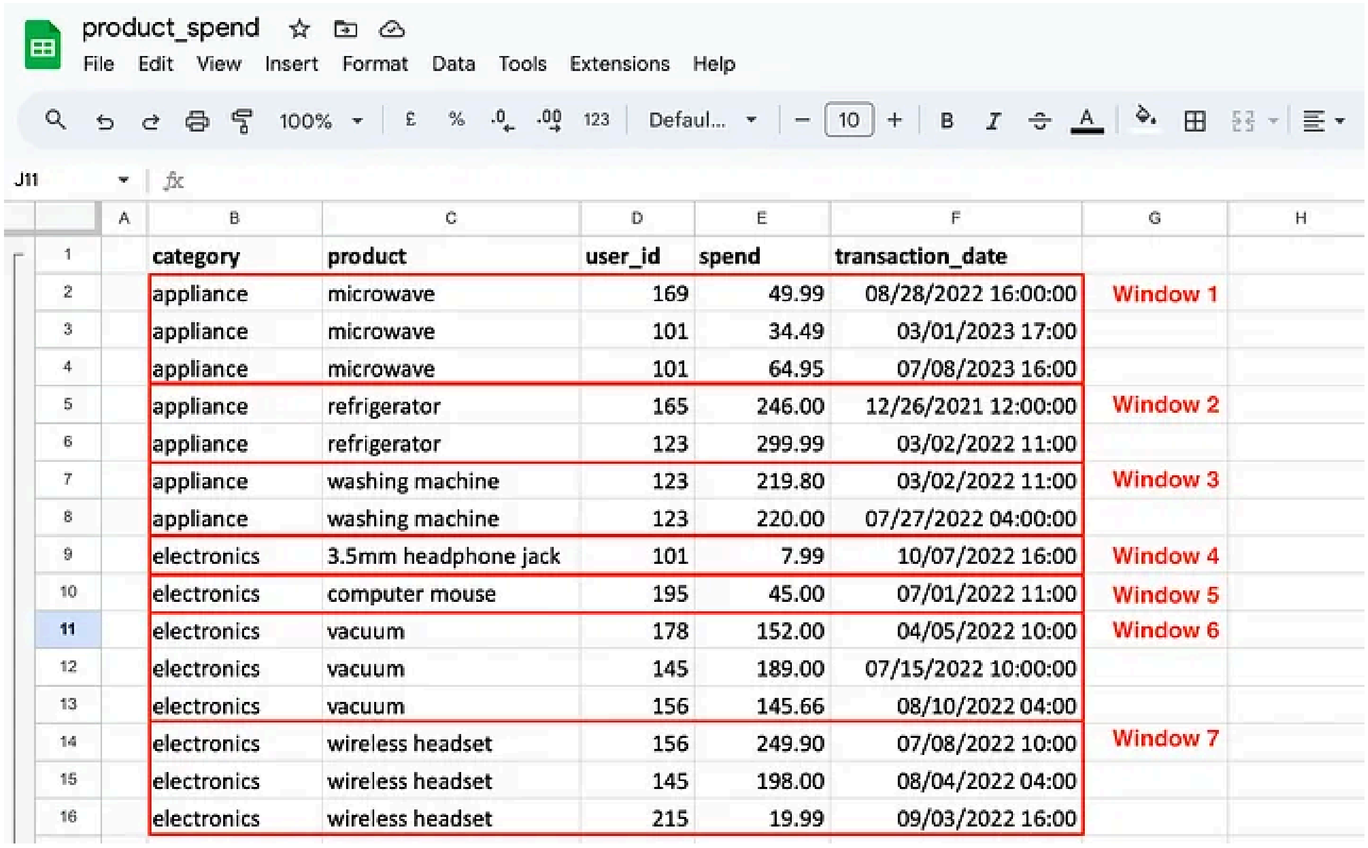
```
SELECT customer_id, total_sales
FROM (
    SELECT customer_id, SUM(amount) AS total_sales
    FROM orders
    GROUP BY customer_id
) AS sales_summary
WHERE total_sales > 500;
```

WINDOW FUNCTION

Window functions operate on a set of rows and return a value for each row in the result set. Unlike aggregate functions, window functions do not group rows into a single output.In simple terms, they are functions that operate by creating virtual "windows" within the dataset.

Syntax:

```
SELECT column1, window_function() OVER (PARTITION BY column ORDER BY column)
FROM table_name;
```



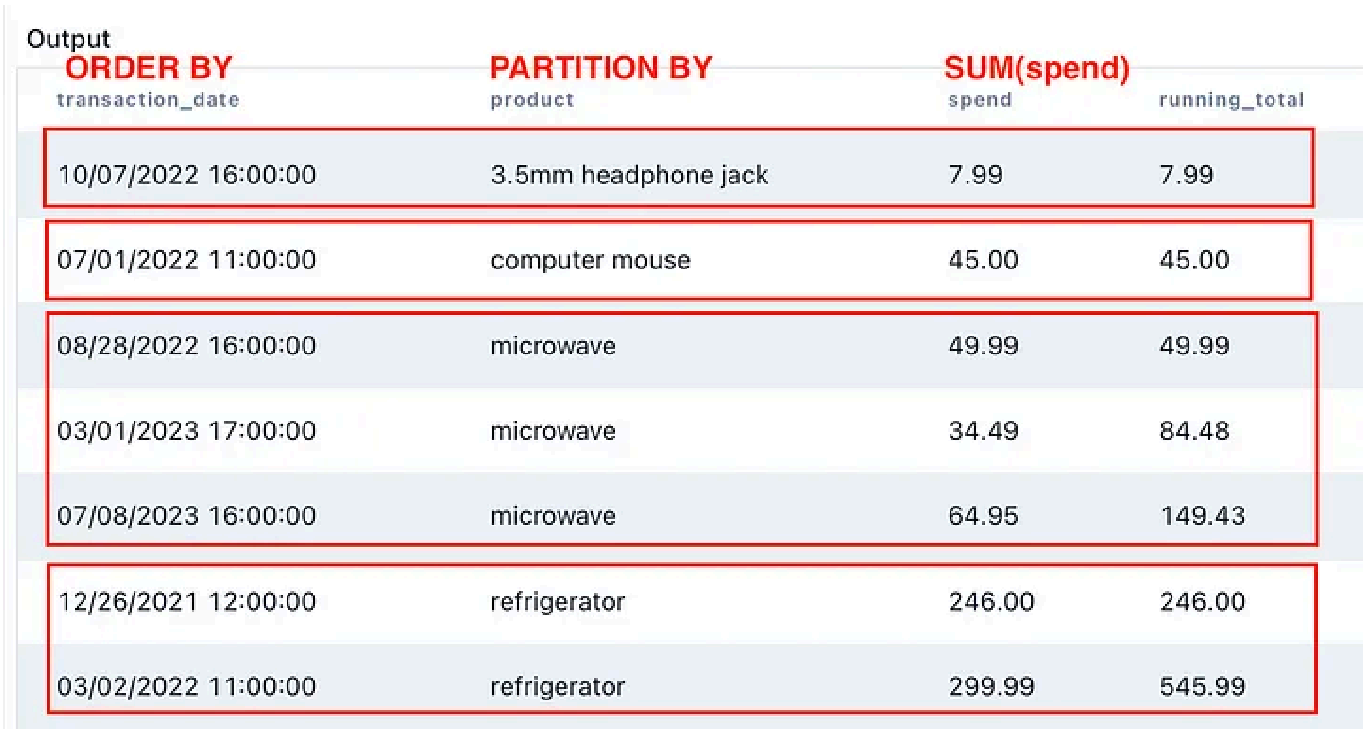
category	product	user_id	spend	transaction_date	
appliance	microwave	169	49.99	08/28/2022 16:00:00	Window 1
appliance	microwave	101	34.49	03/01/2023 17:00	
appliance	microwave	101	64.95	07/08/2023 16:00	
appliance	refrigerator	165	246.00	12/26/2021 12:00:00	Window 2
appliance	refrigerator	123	299.99	03/02/2022 11:00	
appliance	washing machine	123	219.80	03/02/2022 11:00	Window 3
appliance	washing machine	123	220.00	07/27/2022 04:00:00	
electronics	3.5mm headphone jack	101	7.99	10/07/2022 16:00	Window 4
electronics	computer mouse	195	45.00	07/01/2022 11:00	Window 5
electronics	vacuum	178	152.00	04/05/2022 10:00	Window 6
electronics	vacuum	145	189.00	07/15/2022 10:00:00	
electronics	vacuum	156	145.66	08/10/2022 04:00	
electronics	wireless headset	156	249.90	07/08/2022 10:00	Window 7
electronics	wireless headset	145	198.00	08/04/2022 04:00	
electronics	wireless headset	215	19.99	09/03/2022 16:00	

Each window operates independently, so we can do aggregate functions like `SUM` or `AVG` just on a window.

Take a look at this example, which creates a running total of spend for each product type:

```
SELECT
  spend,
   SUM(spend) OVER (
     PARTITION BY product
     ORDER BY transaction_date) AS running_total
  FROM product_spend;
```

You can see that the above query sums up `spend` for each window. We have the results in the image below:


Output
ORDER BY
PARTITION BY
SUM(spend)

transaction_date	product	spend	running_total
10/07/2022 16:00:00	3.5mm headphone jack	7.99	7.99
07/01/2022 11:00:00	computer mouse	45.00	45.00
08/28/2022 16:00:00	microwave	49.99	49.99
03/01/2023 17:00:00	microwave	34.49	84.48
07/08/2023 16:00:00	microwave	64.95	149.43
12/26/2021 12:00:00	refrigerator	246.00	246.00
03/02/2022 11:00:00	refrigerator	299.99	545.99

We're calculating the **sum of spending** for the whole dataset, but we're **organizing it into sections/partitions/windows by `product`**. Within each section/window, we **arrange/order the data by** `transaction_date`, and we're adding up spending as we go down the section.

Window functions are a powerful feature in SQL that allow you to perform calculations across a set of table rows related to the current row without collapsing the result set like `GROUP BY` does. These functions offer the ability to generate running totals, ranking, and more complex aggregations.

Syntax:

```
<window_function> OVER (
[PARTITION BY column1, column2, …]
[ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], …]
)
```

Key Components:

1. PARTITION BY: Similar to `GROUP BY`, it defines subsets (partitions) of data over which the window function will operate.

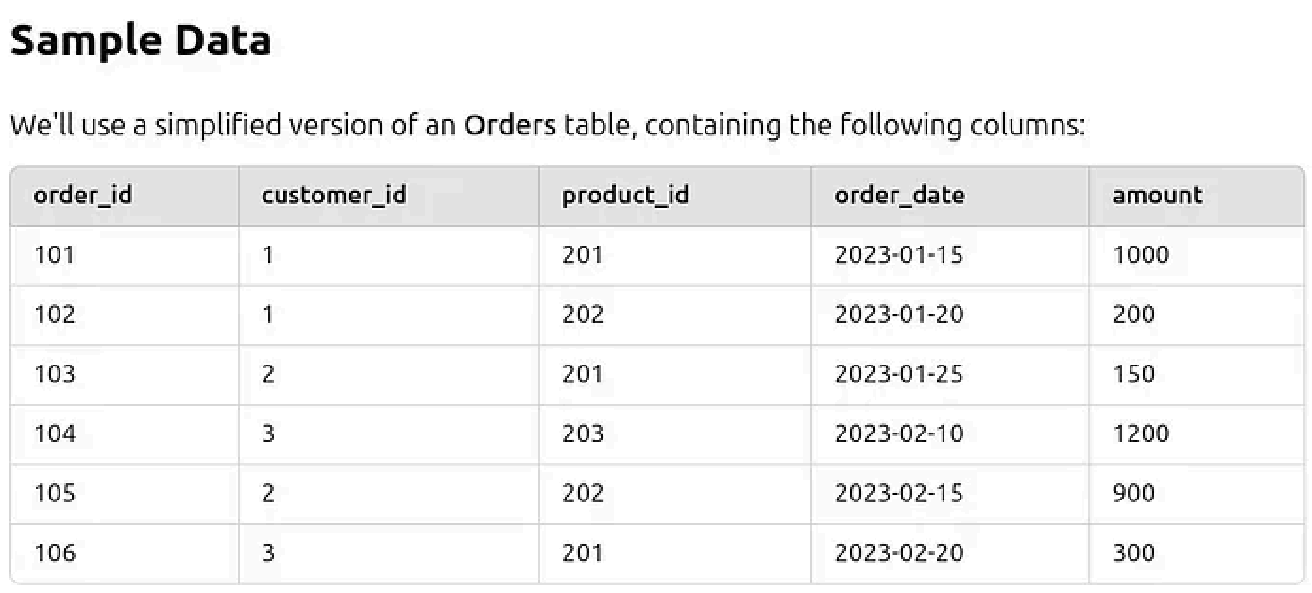
2. ORDER BY: Specifies the order of rows within each partition. It is crucial for window functions like `RANK()` and `ROW_NUMBER()`.

3. Frame specification: (Optional) Determines the range of rows to include within the window.

2. Types of Window Functions

a) Aggregate Window Functions:

- `SUM()`, `AVG()`, `COUNT()`, `MIN()`, and `MAX()` are commonly used aggregate window functions.



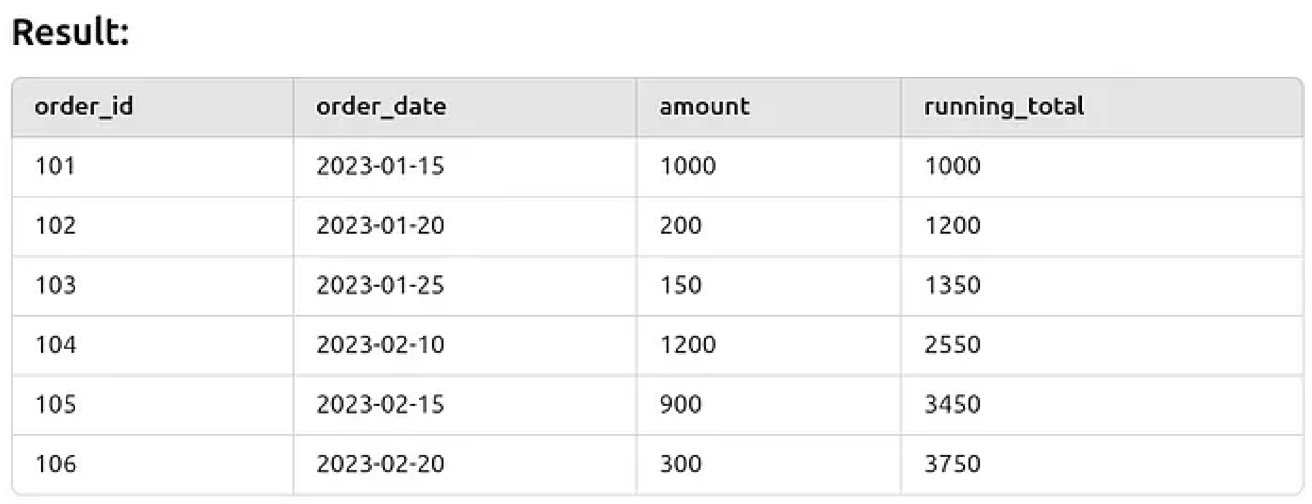
Sample Data

We'll use a simplified version of an **Orders** table, containing the following columns:

order_id	customer_id	product_id	order_date	amount
101	1	201	2023-01-15	1000
102	1	202	2023-01-20	200
103	2	201	2023-01-25	150
104	3	203	2023-02-10	1200
105	2	202	2023-02-15	900
106	3	201	2023-02-20	300

Example: Running Total with SUM()

```
SELECT order_id, order_date, amount,
SUM(amount) OVER (ORDER BY order_date) AS running_total
FROM orders;
```



Result:

order_id	order_date	amount	running_total
101	2023-01-15	1000	1000
102	2023-01-20	200	1200
103	2023-01-25	150	1350
104	2023-02-10	1200	2550
105	2023-02-15	900	3450
106	2023-02-20	300	3750

Explanation:

- The query calculates a running total of `amount` for each order based on the order date.

- The `OVER (ORDER BY order_date)` clause specifies that the running total should be calculated in the order of `order_date`.

b) Ranking Window Functions:

Ranking functions assign a rank or row number to each row in a partition. They are used to rank, number, or assign unique identifiers to rows.

3. Types of Ranking Functions

a) ROW_NUMBER()

`ROW_NUMBER()` assigns a unique, consecutive number to each row, starting from 1 within a partition.

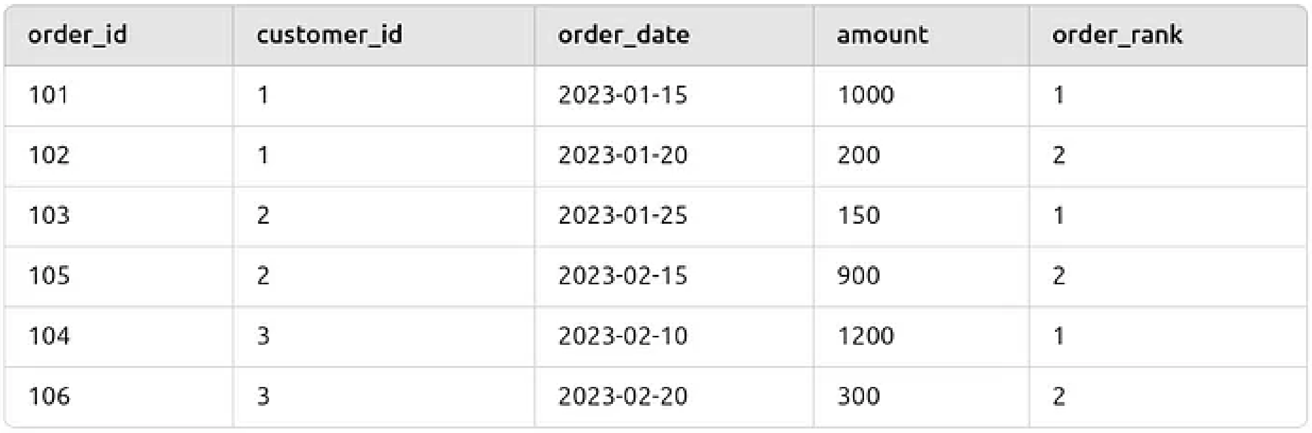
Example: Numbering Orders

```
SELECT order_id, customer_id, order_date, amount,
       ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date) AS order_rank
FROM orders;
```

Explanation:

- `ROW_NUMBER()` assigns a unique rank to each row within the `customer_id` partition based on `order_date`.

Sample Output:



order_id	customer_id	order_date	amount	order_rank
101	1	2023-01-15	1000	1
102	1	2023-01-20	200	2
103	2	2023-01-25	150	1
105	2	2023-02-15	900	2
104	3	2023-02-10	1200	1
106	3	2023-02-20	300	2

b) RANK()

`RANK()` assigns a rank to rows, but if there are ties (rows with the same values in the `ORDER BY` clause), it assigns the same rank to them and skips subsequent ranks.

Example: Ranking Orders by Amount

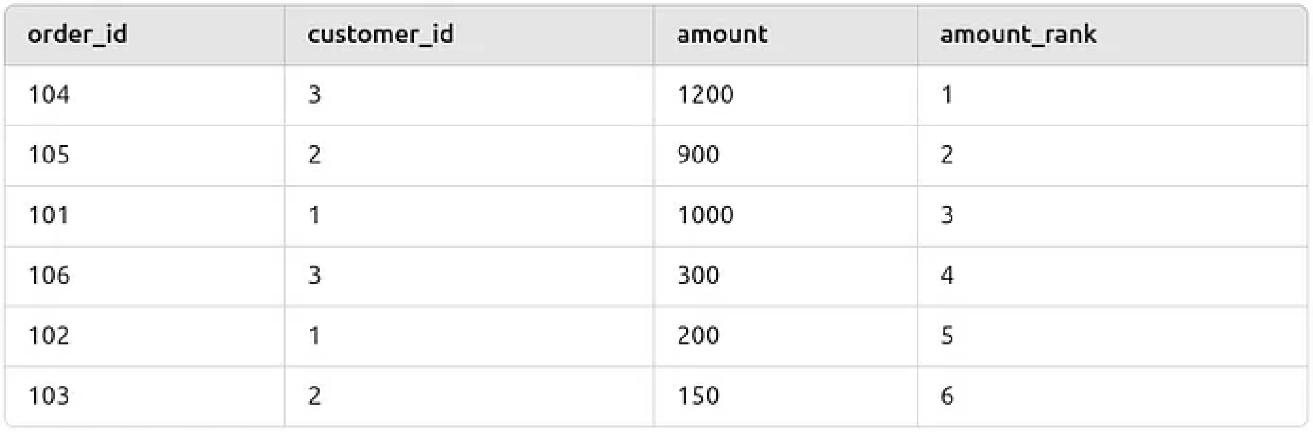
```
SELECT order_id, customer_id, amount,
       RANK() OVER (ORDER BY amount DESC) AS amount_rank
FROM orders;
```

Explanation:

- This query ranks orders based on the `amount` in descending order.

- If two rows have the same amount, they receive the same rank, and the next rank is skipped.

Sample Output:



order_id	customer_id	amount	amount_rank
104	3	1200	1
105	2	900	2
101	1	1000	3
106	3	300	4
102	1	200	5
103	2	150	6

c) DENSE_RANK()

`DENSE_RANK()` works similarly to `RANK()`, but it does not skip ranks when ties occur.

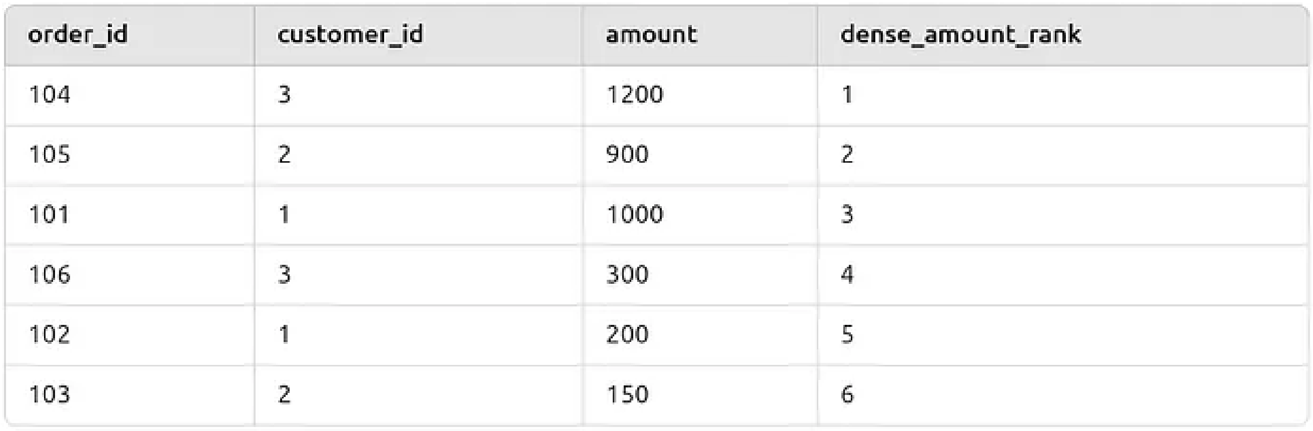
Example: Dense Ranking by Amount

```
SELECT order_id, customer_id, amount,
       DENSE_RANK() OVER (ORDER BY amount DESC) AS dense_amount_rank
FROM orders;
```

Explanation:

- Unlike `RANK()`, `DENSE_RANK()` assigns consecutive ranks even when there are ties in the `ORDER BY` clause.

Sample Output:



order_id	customer_id	amount	dense_amount_rank
104	3	1200	1
105	2	900	2
101	1	1000	3
106	3	300	4
102	1	200	5
103	2	150	6

d) NTILE()

`NTILE()` divides the rows into a specified number of approximately equal-sized groups and assigns a group number to each row.

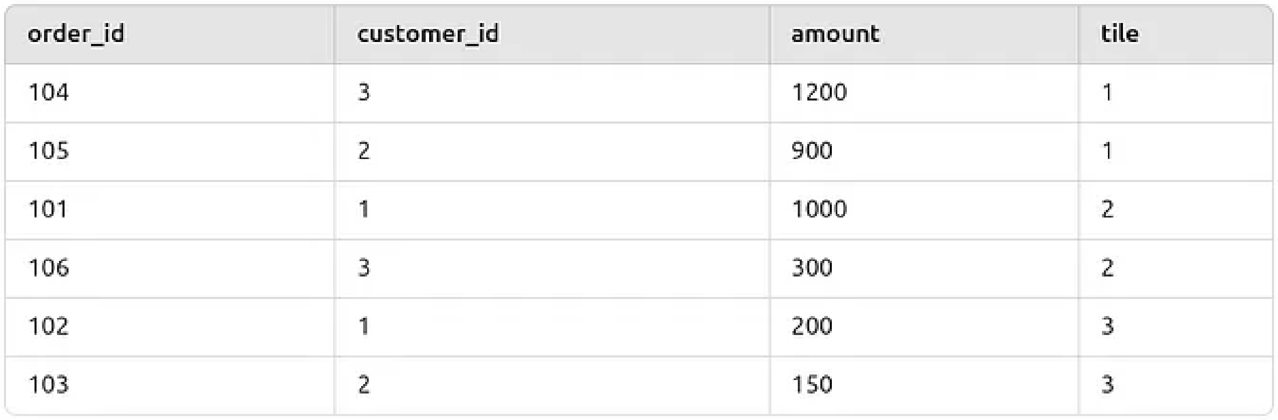
Example: Divide Orders into Quartiles

```
SELECT order_id, customer_id, amount,
       NTILE(3) OVER (ORDER BY amount DESC) AS tile
FROM orders;
```

Explanation:

- This query divides the orders into four quartiles based on the `amount`. The highest amounts are in quartile 1, and the lowest in quartile 4.

Sample Output:



order_id	customer_id	amount	tile
104	3	1200	1
105	2	900	1
101	1	1000	2
106	3	300	2
102	1	200	3
103	2	150	3

4. Lead and Lag Functions

- `LEAD()` and `LAG()` are window functions used to access data from subsequent or preceding rows.

a) LAG()

`LAG()` provides access to the previous row's value in the result set.

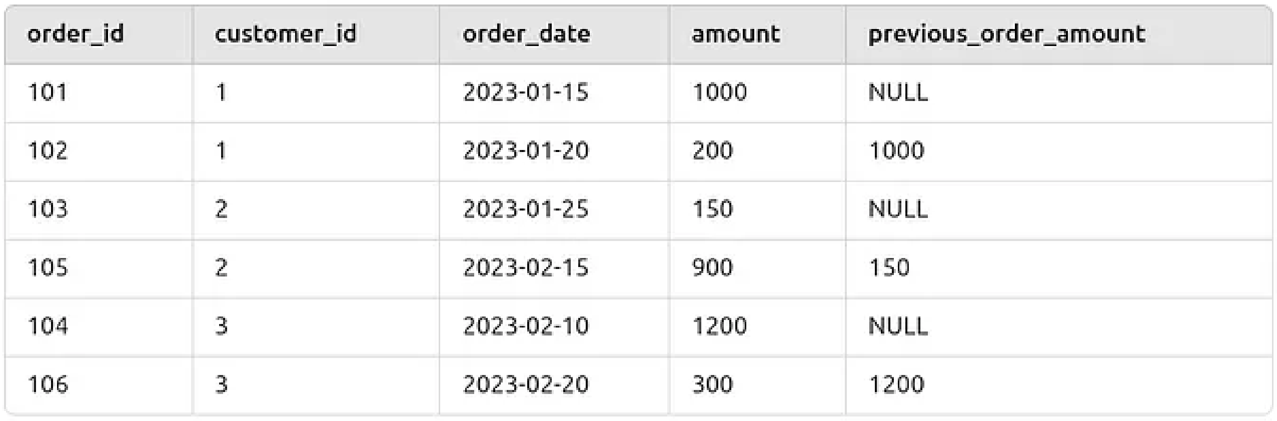
Example: Compare Orders with Previous Order

```
SELECT order_id, customer_id, order_date, amount,
       LAG(amount) OVER (PARTITION BY customer_id ORDER BY order_date) AS previous_or
FROM orders;
```

Explanation:

- This query retrieves the `amount` of the previous order for comparison with the current order.

Sample Output:



order_id	customer_id	order_date	amount	previous_order_amount
101	1	2023-01-15	1000	NULL
102	1	2023-01-20	200	1000
103	2	2023-01-25	150	NULL
105	2	2023-02-15	900	150
104	3	2023-02-10	1200	NULL
106	3	2023-02-20	300	1200

b) LEAD()

`LEAD()` provides access to the next row's value in the result set.

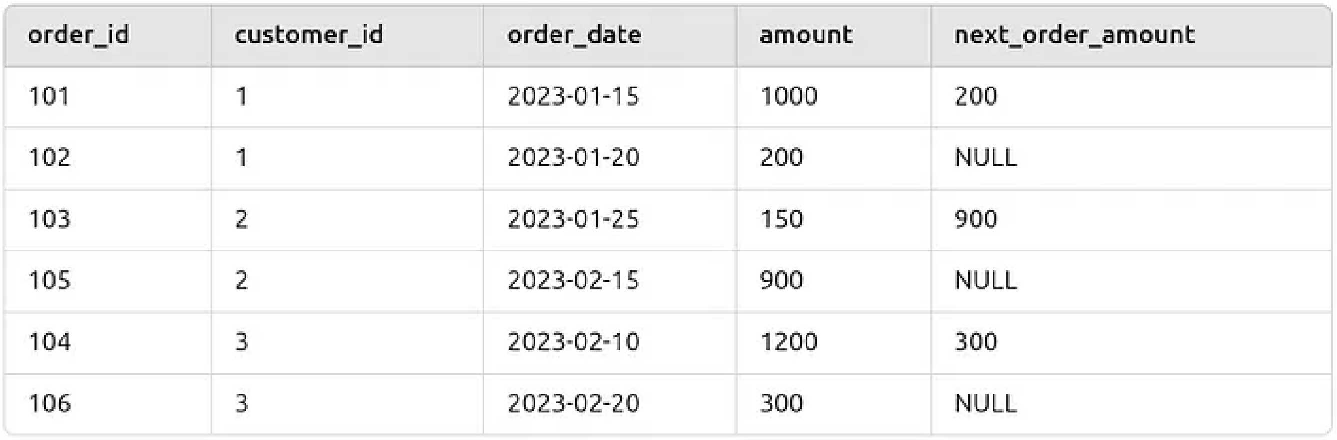
Example: Compare Orders with Next Order

```
SELECT order_id, customer_id, order_date, amount,
       LEAD(amount) OVER (PARTITION BY customer_id ORDER BY order_date) AS next_order_
FROM orders;
```

Explanation:

- This query retrieves the `amount` of the next order to compare with the current order.

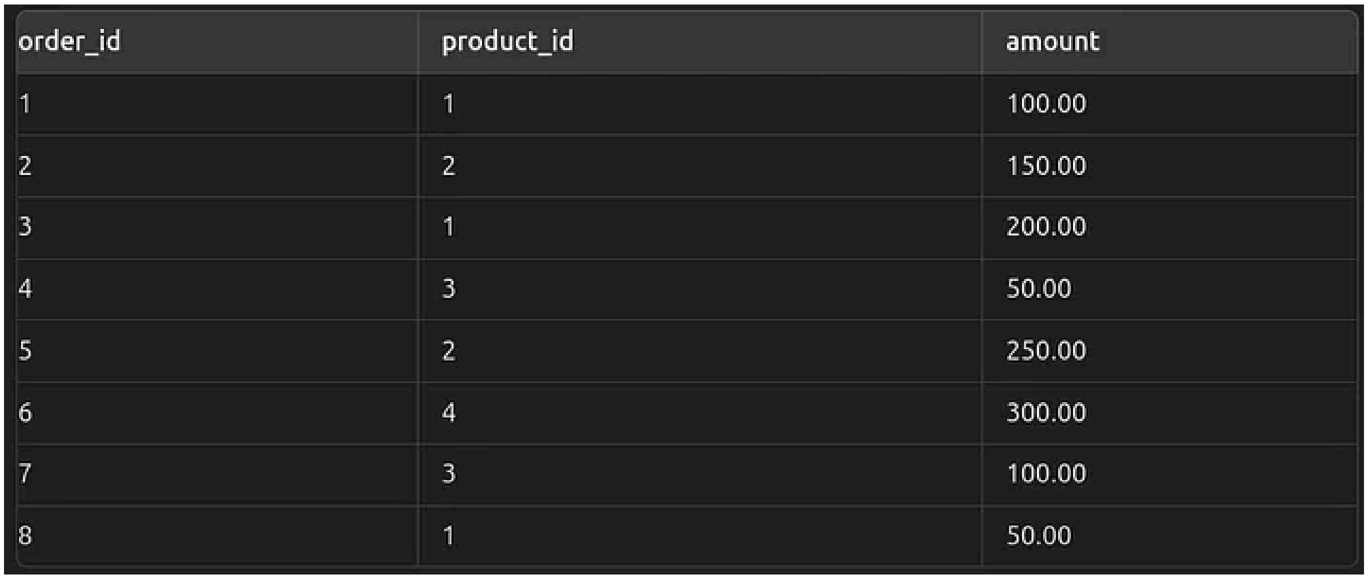
Sample Output:



order_id	customer_id	order_date	amount	next_order_amount
101	1	2023-01-15	1000	200
102	1	2023-01-20	200	NULL
103	2	2023-01-25	150	900
105	2	2023-02-15	900	NULL
104	3	2023-02-10	1200	300
106	3	2023-02-20	300	NULL

5. Practical Use Cases for Window and Ranking Functions

a) Finding the Top 3 Best-Selling Products

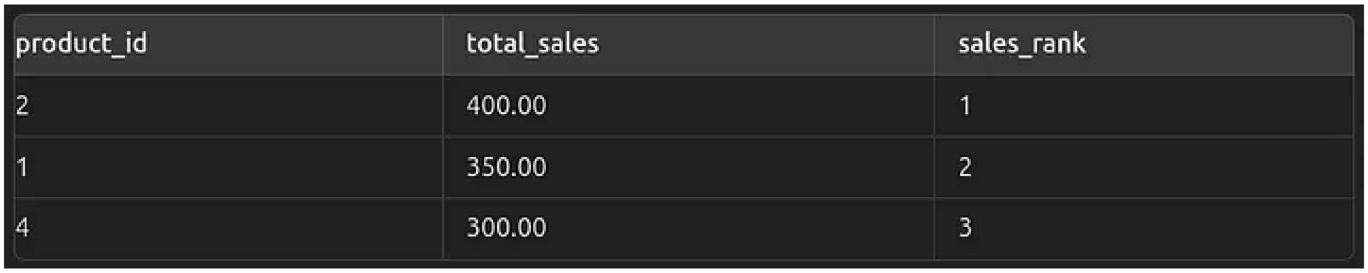


order_id	product_id	amount
1	1	100.00
2	2	150.00
3	1	200.00
4	3	50.00
5	2	250.00
6	4	300.00
7	3	100.00
8	1	50.00

```
SELECT product_id, SUM(amount) AS total_sales,
       RANK() OVER (ORDER BY SUM(amount) DESC) AS sales_rank
FROM orders
GROUP BY product_id
ORDER BY sales_rank
LIMIT 3;
```

Explanation:

This query uses `RANK()` to find the top 3 best-selling products based on total sales.

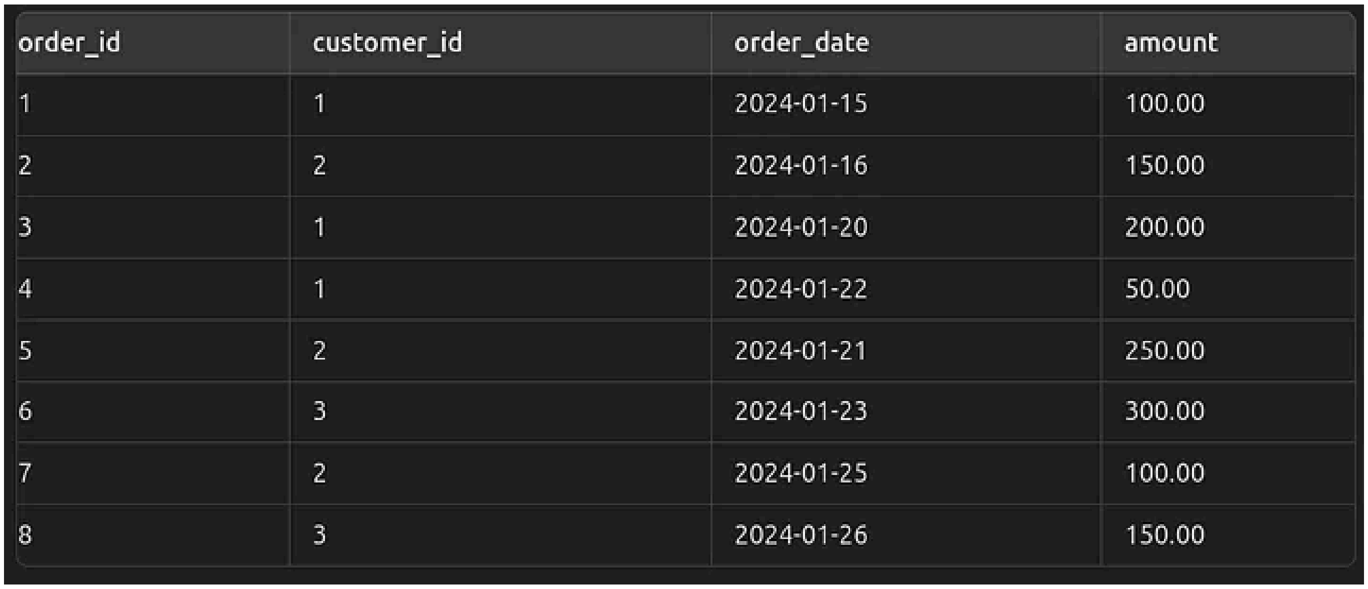


product_id	total_sales	sales_rank
2	400.00	1
1	350.00	2
4	300.00	3

Total Sales Calculation for Each Product:

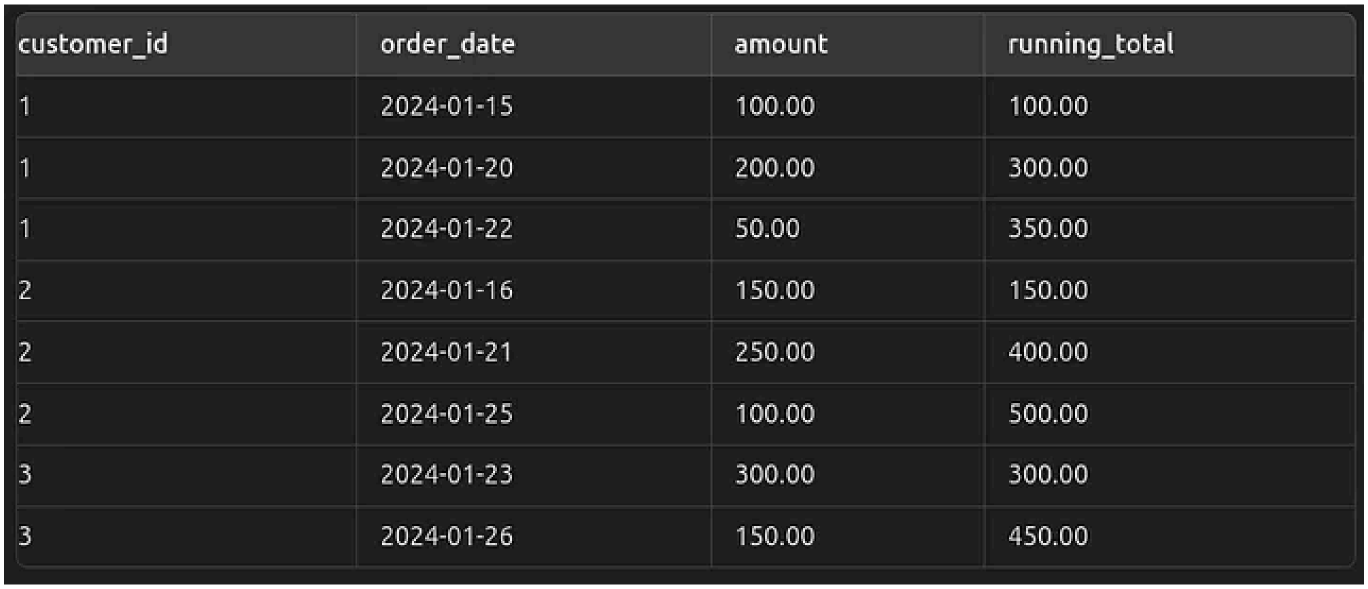
- **product_id 1:** 100.00 + 200.00 + 50.00 = 350.00
- **product_id 2:** 150.00 + 250.00 = 400.00
- **product_id 3:** 50.00 + 100.00 = 150.00
- **product_id 4:** 300.00

b) Calculating a Running Total of Sales Per Customer



order_id	customer_id	order_date	amount
1	1	2024-01-15	100.00
2	2	2024-01-16	150.00
3	1	2024-01-20	200.00
4	1	2024-01-22	50.00
5	2	2024-01-21	250.00
6	3	2024-01-23	300.00
7	2	2024-01-25	100.00
8	3	2024-01-26	150.00

```
SELECT customer_id, order_date, amount,
       SUM(amount) OVER (PARTITION BY customer_id ORDER BY order_date) AS running_tota
FROM orders;
```



customer_id	order_date	amount	running_total
1	2024-01-15	100.00	100.00
1	2024-01-20	200.00	300.00
1	2024-01-22	50.00	350.00
2	2024-01-16	150.00	150.00
2	2024-01-21	250.00	400.00
2	2024-01-25	100.00	500.00
3	2024-01-23	300.00	300.00
3	2024-01-26	150.00	450.00

Explanation of Output:

customer_id 1:

- **2024–01–15:** Order amount is 100.00. Running total is 100.00.
- **2024–01–20:** Order amount is 200.00. Running total is 100.00 + 200.00 = 300.00.
- **2024–01–22:** Order amount is 50.00. Running total is 300.00 + 50.00 = 350.00.

customer_id 2:

- **2024–01–16:** Order amount is 150.00. Running total is 150.00.

- **2024–01–21:** Order amount is 250.00. Running total is 150.00 + 250.00 = 400.00.
- **2024–01–25:** Order amount is 100.00. Running total is 400.00 + 100.00 = 500.00.

customer_id 3:

- **2024–01–23:** Order amount is 300.00. Running total is 300.00.
- **2024–01–26:** Order amount is 150.00. Running total is 300.00 + 150.00 = 450.00.

The `SUM(amount) OVER (PARTITION BY customer_id ORDER BY order_date)` window function calculates the running total of the `amount` for each `customer_id`, ordered by `order_date`.

SQL SELF-JOINS

A self-join is a join where a table is joined with itself. This is useful for comparing rows within the same table.

Syntax:

```
SELECT a.column1, b.column2
FROM table_name a
JOIN table_name b ON a.common_column = b.common_column;
```

Example:

To find orders placed by the same customer on different dates:

```
SELECT a.order_id, a.amount, b.order_id AS other_order_id, b.amount AS other_amount
FROM orders a
JOIN orders b ON a.customer_id = b.customer_id AND a.order_id != b.order_id;
```

SQL UNION

The `UNION` operator combines the results of two or more SELECT queries into a single result set. The `UNION ALL` operator includes duplicate rows, while `UNION` removes them.

Syntax:

```
SELECT column1 FROM table1
UNION
SELECT column1 FROM table2;
```

Example:

To combine customer IDs from `customers` and `orders` tables:

```
SELECT customer_id FROM customers
UNION
SELECT customer_id FROM orders;
```

EXECUTION ORDER ↕

Understanding SQL's execution order can help write better queries. Here's the typical order:

1. FROM: Identify the source tables.

2. JOIN: Apply joins if needed.

3. WHERE: Filter rows.

4. GROUP BY: Group rows for aggregation.

5. HAVING: Filter grouped rows.

6. SELECT: Choose which columns to return.

7. ORDER BY: Sort the result set.

8. LIMIT: Limit the number of rows returned.

SQL PIVOTING

Pivoting in SQL converts rows into columns, allowing for better data reporting and analysis. SQL doesn't have a built-in `PIVOT` function in some databases, so you may need to use aggregate functions and `CASE` statements.

Example:

To pivot sales data by month:

```
SELECT customer_id,
    SUM(CASE WHEN month = 'January' THEN amount ELSE 0 END) AS January,
    SUM(CASE WHEN month = 'February' THEN amount ELSE 0 END) AS February
FROM sales
GROUP BY customer_id;
```

STRING FUNCTIONS

String functions in SQL allow you to manipulate and format text data. Common functions include:

- `CONCAT()`: Concatenates strings.

- `SUBSTRING()`: Extracts a portion of a string.

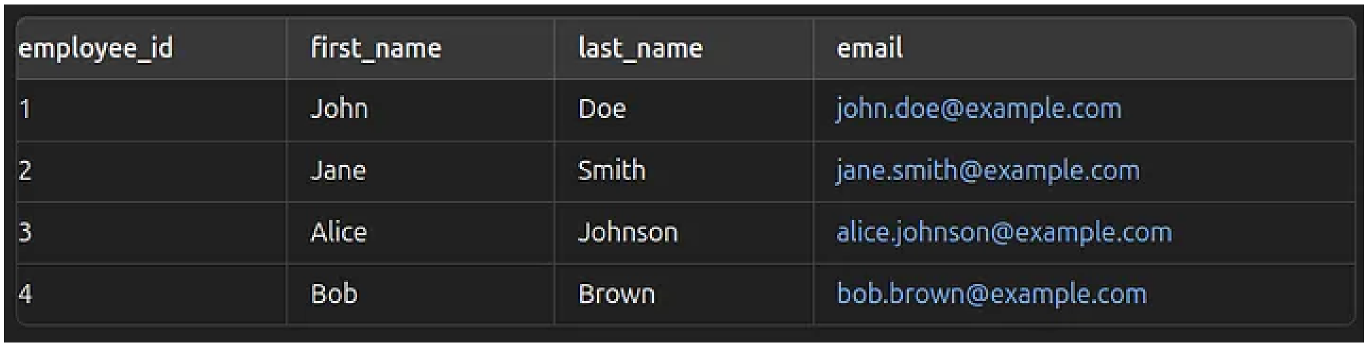
- `LENGTH()`: Returns the length of a string.

- `UPPER()` / `LOWER()`: Converts a string to upper or lowercase.

Here are examples of how to use common SQL string functions with sample data. We'll use a hypothetical table called `employees` to illustrate these functions.

Sample Table: `employees`

Assume we have the following `employees` table:



employee_id	first_name	last_name	email
1	John	Doe	john.doe@example.com
2	Jane	Smith	jane.smith@example.com
3	Alice	Johnson	alice.johnson@example.com
4	Bob	Brown	bob.brown@example.com

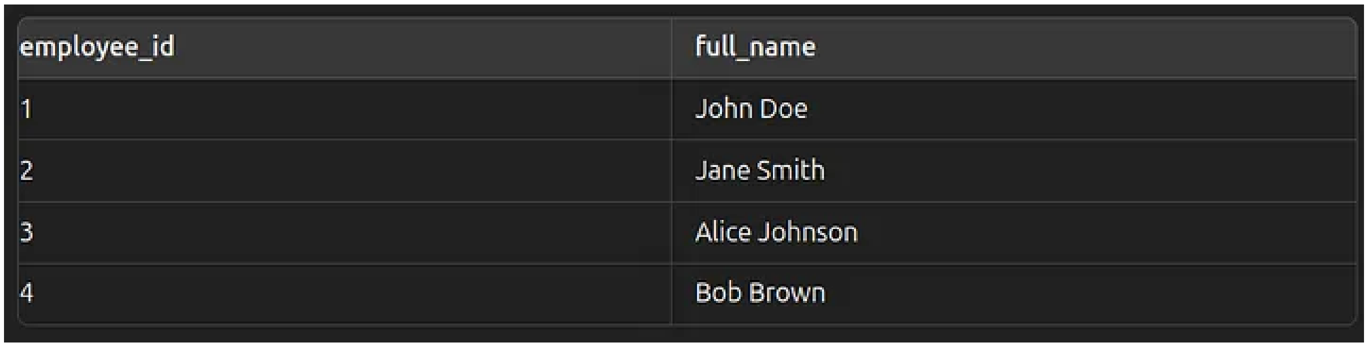
1. `CONCAT()`

Description: Concatenates two or more strings into one string.

Example: Concatenate `first_name` and `last_name` to get the full name.

```
SELECT employee_id,
CONCAT(first_name, ' ', last_name) AS full_name
FROM employees;
```

Output:



employee_id	full_name
1	John Doe
2	Jane Smith
3	Alice Johnson
4	Bob Brown

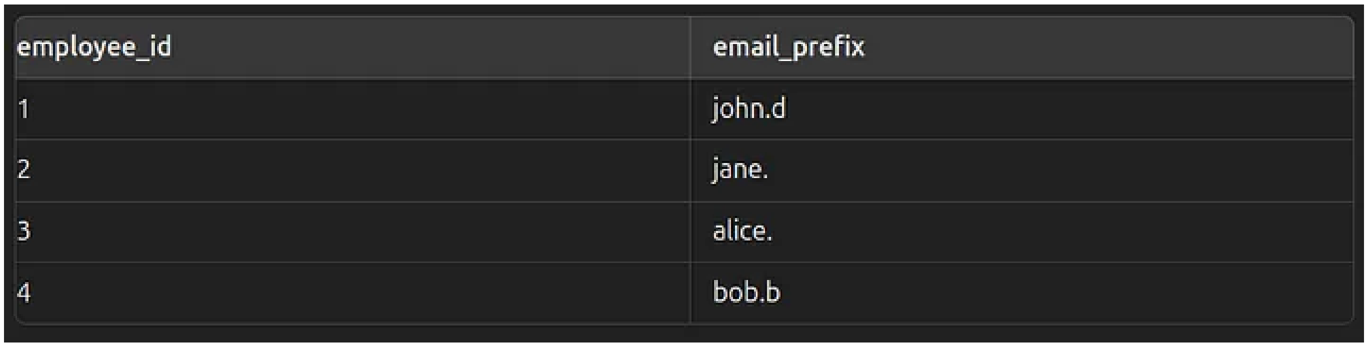
2. `SUBSTRING()`

Description: Extracts a portion of a string, given a starting position and length.

Example: Extract the first 5 characters of the `email`.

```
SELECT employee_id,
SUBSTRING(email, 1, 5) AS email_prefix
FROM employees;
```

Output:



employee_id	email_prefix
1	john.d
2	jane.
3	alice.
4	bob.b

3. `LENGTH()`

Description: Returns the length of a string.

Example: Get the length of each `email`.

```
SELECT employee_id,
LENGTH(email) AS email_length
FROM employees;
```

Output:

```
| employee_id  | email_length  |
| - - - - - - -| - - - - - - - |
| 1            | 19            |
| 2            | 22            |
| 3            | 25            |
| 4            | 20            |
```

4. `UPPER()` / `LOWER()`

Description: Converts a string to uppercase or lowercase.

Examples:

- Convert `first_name` to uppercase:

```
SELECT employee_id,
UPPER(first_name) AS first_name_upper
FROM employees;
```

Output:

```
| employee_id  | first_name_upper  |
| - - - - - - -| - - - - - - - - - |
| 1            | JOHN              |
| 2            | JANE              |
| 3            | ALICE             |
| 4            | BOB               |
```

- Convert `last_name` to lowercase:

```
SELECT employee_id,
LOWER(last_name) AS last_name_lower
FROM employees;
```

Output:

```
| employee_id  | last_name_lower  |
| - - - - - - -| - - - - - - - - -|
| 1            | doe              |
| 2            | smith            |
| 3            | johnson          |
| 4            | brown            |
```

This SQL Tutorial helps you explore the power of SQL to handle more complex queries and large datasets efficiently. By mastering these concepts, you can solve real-world problems, analyze data, and optimize SQL performance. Happy querying!