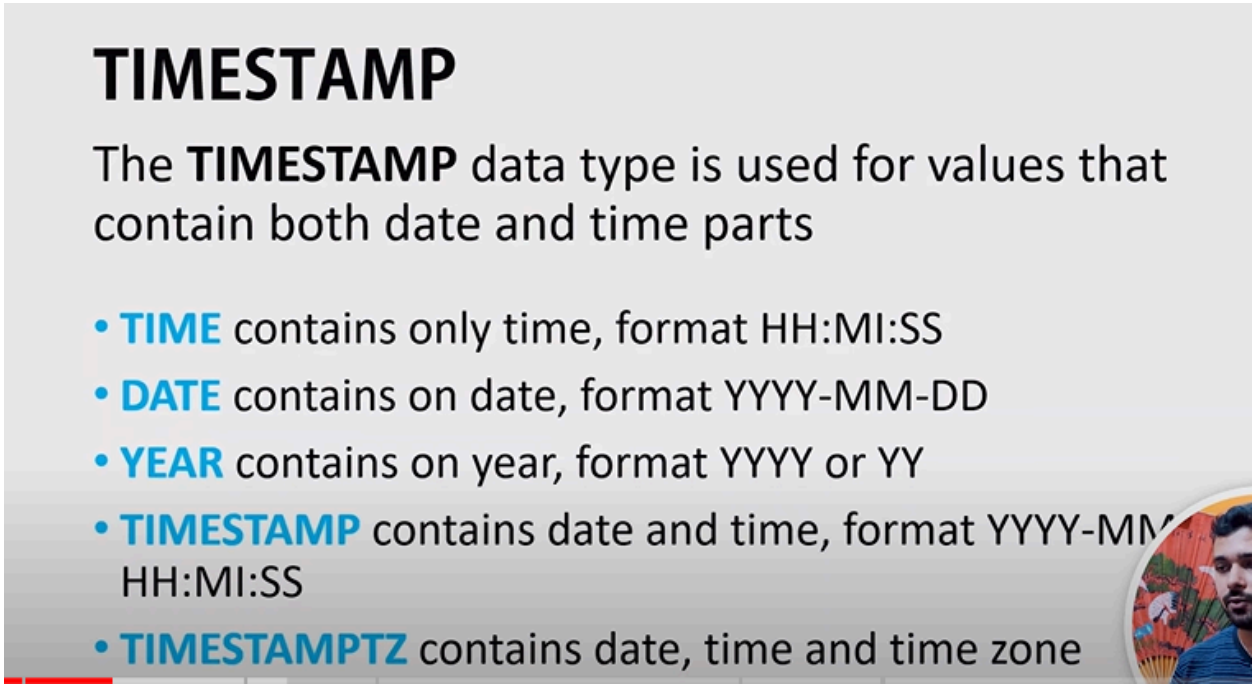
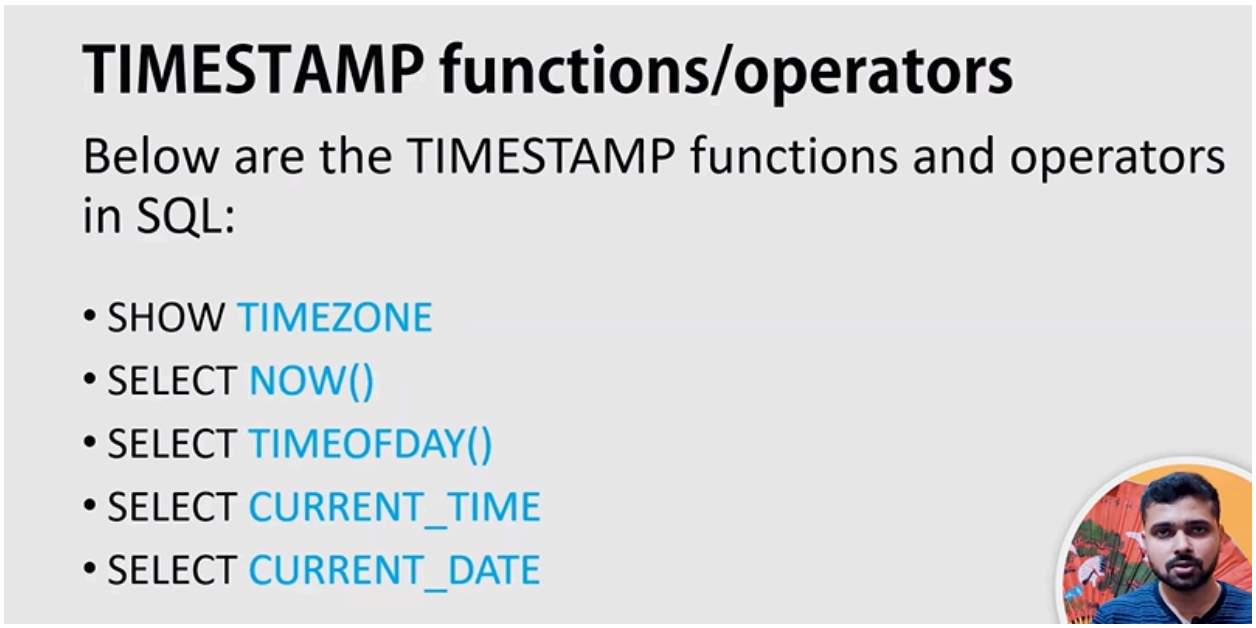
SQL JOINS and TIMESTAMP:





DATE_ADD():
Syntax:DATE_ADD(date, INTERVAL value unit)
date: The original date to which you want to add an interval.
value: The amount of the interval you want to add (can be positive or negative).

unit: The unit of time for the interval (e.g., DAY, MONTH, YEAR, HOUR, MINUTE, SECOND).

Common Units for Intervals

- **DAY**: Days
- **MONTH**: Months
- **YEAR**: Years
- **HOUR**: Hours
- **MINUTE**: Minutes
- **SECOND**: Seconds
- **WEEK**: Weeks

Examples

1. Add Days to a Date

SELECT DATE_ADD('2024-01-01', INTERVAL 10 DAY) AS NewDate;

Output: `2024-01-11`

`Subtract Days from a Date`

```
SELECT DATE_ADD('2024-01-01', INTERVAL -10 DAY) AS
NewDate;
```

`2023-12-22`

`Add Months to a Date`

```
SELECT DATE_ADD('2024-01-01', INTERVAL 2 MONTH) AS
NewDate;
```

`Add Years to a Date`

```
SELECT DATE_ADD('2024-01-01 10:00:00', INTERVAL 5
HOUR) AS NewDateTime;
```

Output: 2024-01-01 15:00:00

Add Hours to a DateTime

```
SELECT DATE_ADD('2024-01-01 10:00:00', INTERVAL 5
HOUR) AS NewDateTime;
```

2024-01-01 15:00:00

DATE_FORMAT():

You can use the DATE_FORMAT function to extract both the year and month in one step:

```
DATE_FORMAT(created_at, '%Y-%m') AS year_month, --
Formats the date as 'YYYY-MM'
```

created_at=column name

In SQL, particularly in MySQL, `SUBDATE` and `DATE_ADD` are functions used to manipulate dates, but they serve opposite purposes:

`SUBDATE` Function

- **Purpose**: Used to subtract a specified time interval from a date.

- **Syntax**: `SUBDATE(date, INTERVAL expr unit)` or `date - INTERVAL expr unit`

- **Example**:

    ```sql

    SELECT SUBDATE('2024-10-01', INTERVAL 7 DAY) AS new_date;

    ```

    - This would return `2024-09-24`, subtracting 7 days from `2024-10-01`.

`DATE_ADD` Function

- **Purpose**: Used to add a specified time interval to a date.

- **Syntax**: `DATE_ADD(date, INTERVAL expr unit)` or `date + INTERVAL expr unit`

- **Example**:

    ```sql
    ```

```
    SELECT DATE_ADD('2024-10-01', INTERVAL 7 DAY)
AS new_date;
    ```

    - This would return `2024-10-08`, adding 7 days
to `2024-10-01`.

Summary

- **`SUBDATE`**: Subtracts time from a date.

- **`DATE_ADD`**: Adds time to a date.

Example of Both in a Query

Here's a combined example to illustrate both
functions:


```sql

SELECT

    '2024-10-01' AS original_date,

    SUBDATE('2024-10-01', INTERVAL 7 DAY) AS
subtracted_date,
```

```
    DATE_ADD('2024-10-01', INTERVAL 7 DAY) AS
added_date;
```

Output:

original_date	subtracted_date	added_date
2024-10-01	2024-09-24	2024-10-08

In this example:

- `original_date` is the starting date.
- `subtracted_date` shows the date 7 days prior to `2024-10-01`.
- `added_date` shows the date 7 days after `2024-10-01`.

EXTRACT Function

The **EXTRACT()** function extracts a part from a given date value.

Syntax: SELECT **EXTRACT**(MONTH FROM date_field) FROM Table

- **YEAR**
- **QUARTER**
- **MONTH**
- **WEEK**
- **DAY**
- **HOUR**
- **MINUTE**
- **DOW** – day of week
- **DOY** – day of year

Query Query History

```
1  SELECT * FROM payment
2
```

Data output Messages Notifications

	customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date
1	1	60	Cash	2020-09-24
2	2	30	Credit Card	2020-04-27
3	8	110	Cash	2021-01-26
4	10	70	mobile Payment	2021-02-28
5	11	80	Cash	2021-03-01

Query | Query History

```
1  SELECT EXTRACT(YEAR FROM payment_date) AS pay_year, payment_date
2  FROM payment
3
```

Data output | Messages | Notifications

	pay_year numeric	payment_date date
1	2020	2020-09-24
2	2020	2020-04-27
3	2021	2021-01-26
4	2021	2021-02-28
5	2021	2021-03-01

Query | Query History

```
1  SELECT EXTRACT(QUARTER FROM payment_date) AS pay_time, payment_date
2  FROM payment
3
```

Data output | Messages | Notifications

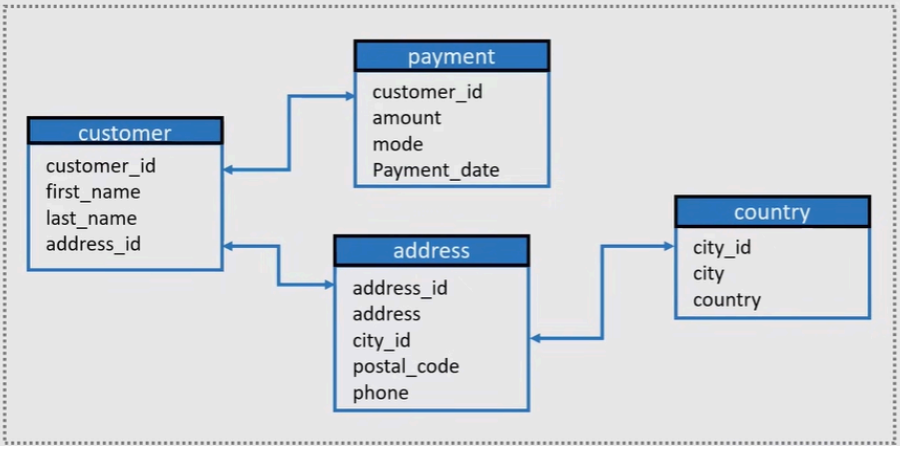
	pay_time numeric	payment_date date
1	3	2020-09-24
2	2	2020-04-27
3	1	2021-01-26
4	1	2021-02-28
5	1	2021-03-01

SQL JOIN

- **JOIN** means to combine something.
- A **JOIN** clause is used to combine data from two or more tables, based on a related column between them
- Let's understand the joins through an example:

JOIN Example

Question: How much amount was paid by customer 'Madan', what was mode and payment date?



JOIN Example

customer_id [PK] bigint	first_name character varying (50)	last_name character varying (50)	address_id bigint
1	Mary	Smith	5
2	Madan	Mohan	6
3	Linda	Williams	7
4	Barbara	Jones	8
5	Elizabeth	Brown	9

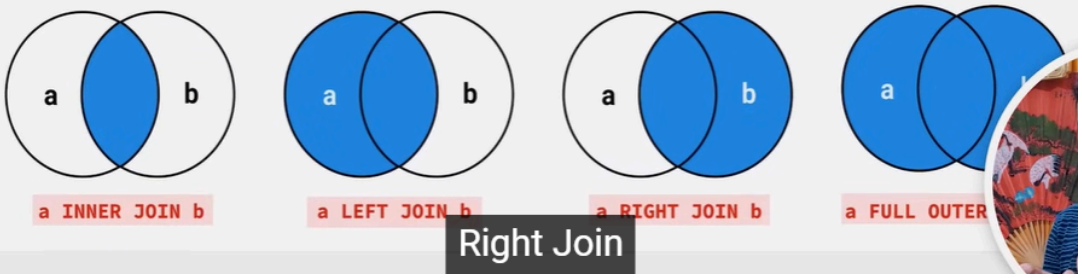
Question: How much amount was paid by customer 'Madan', what was mode and payment date?

customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date
1	60	Cash	2020-09-24
2	30	Credit Card	2020-04-27
3	90	Credit Card	2020-07-07
4	50	Debit Crad	2020-02-12
5	40	Mobile Payment	2020-11-20

Answer: Amount = 30
Mode = Credit Card,
Date = 2020-04-27

TYPES OF JOINS

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

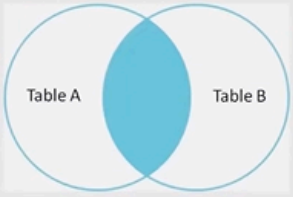


Value same ho boh table main for inner join

INNER JOIN

- **Syntax**

```
SELECT column_name(s)
FROM TableA
INNER JOIN TableB
ON TableA.col_name = TableB.col_name
```



- **Example**

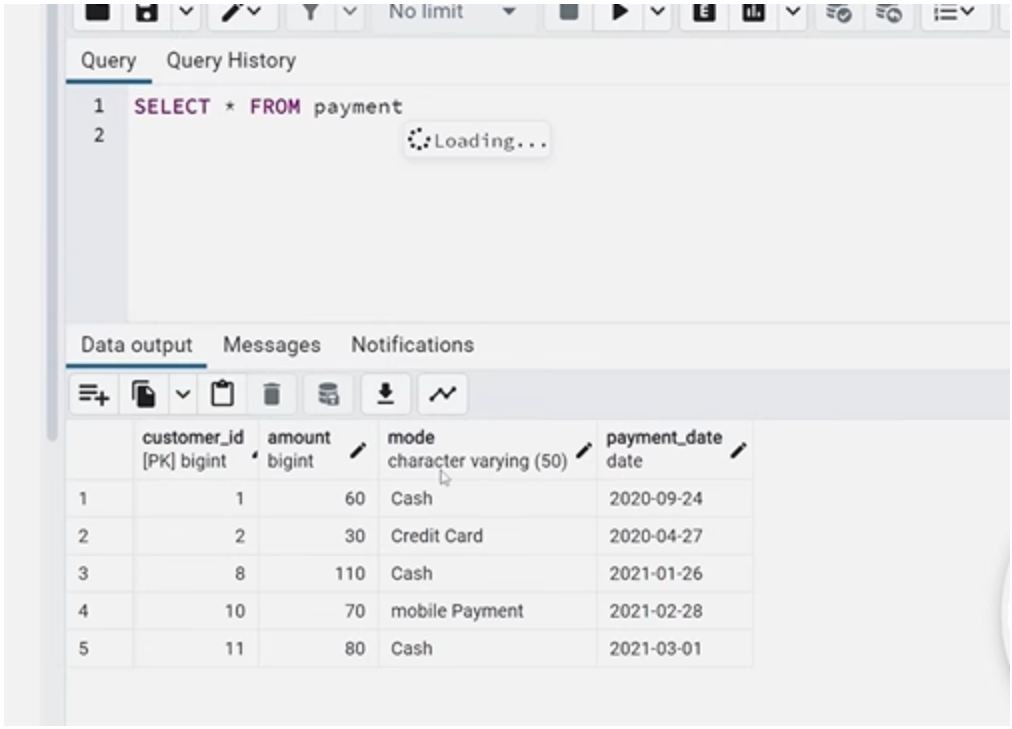
```
SELECT *
FROM customer AS c
INNER JOIN payment AS p
ON c.customer_id = p.customer_id
```

Query | Query History

```
1  SELECT * FROM customer
2
```

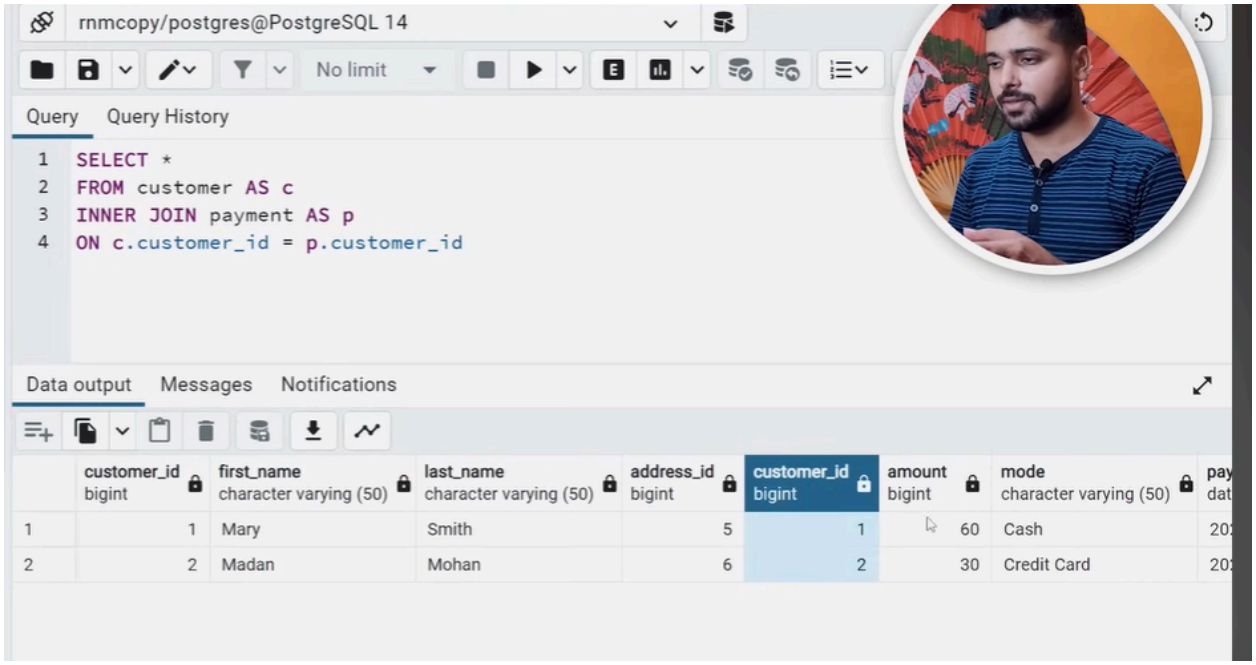
Data output | Messages | Notifications

	customer_id [PK] bigint	first_name character varying (50)	last_name character varying (50)	address_id bigint
1	1	Mary	Smith	5
2	3	Linda	Williams	7
3	4	Barbara	Jones	8
4	2	Madan	Mohan	6



	customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date
1	1	60	Cash	2020-09-24
2	2	30	Credit Card	2020-04-27
3	8	110	Cash	2021-01-26
4	10	70	mobile Payment	2021-02-28
5	11	80	Cash	2021-03-01

Customer same honge bahi data aayega



	customer_id bigint	first_name character varying (50)	last_name character varying (50)	address_id bigint	customer_id bigint	amount bigint	mode character varying (50)	pay dat
1	1	Mary	Smith	5	1	60	Cash	20
2	2	Madan	Mohan	6	2	30	Credit Card	20

```
SELECT c.first_name, p.amount, p.mode
FROM customer AS c
INNER JOIN payment AS p
ON c.customer_id = p.customer_id
```

//left table se data meet karta ho

LEFT JOIN

- **Syntax**

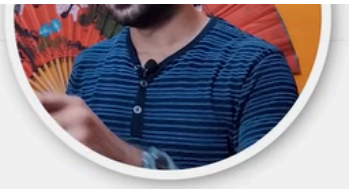
```
SELECT column_name(s)
FROM TableA
LEFT JOIN TableB
ON TableA.col_name = TableB.col_name
```

- **Example**

```
SELECT *
FROM customer AS c
LEFT JOIN payment AS p
ON c.customer_id = p.customer_id
```

Query | Query History

```
1  SELECT *
2  FROM customer AS c
3  LEFT JOIN payment AS p
4  ON c.customer_id = p.customer_id
5
```



Data output | Messages | Notifications

	customer_id bigint	first_name character varying (50)	last_name character varying (50)	address_id bigint	customer_id bigint	amount bigint	mode character varying (50)	pay dat
1	1	Mary	Smith	5	1	60	Cash	20
2	2	Madan	Mohan	6	2	30	Credit Card	20
3	4	Barbara	Jones	8	[null]	[null]	[null]	[nu
4	3	Linda	Williams	7	[null]	[null]	[null]	[nu

SUBSCRIBE

Similar to right

RIGHT JOIN

- **Syntax**

```
SELECT column_name(s)
FROM TableA
RIGHT JOIN TableB
ON TableA.col_name = TableB.col_name
```

- **Example**

```
SELECT *
FROM customer AS c
RIGHT JOIN payment AS p
ON c.customer_id = p.customer_id
```

SELF JOIN

- A **self join** is a regular join in which a table is joined to itself
- **SELF Joins** are powerful for comparing values in a column of rows with the same table
- Syntax

```
SELECT column_name(s)
FROM Table AS T1
JOIN Table AS T2
ON T1.col_name = T2.col_name
```

SELF JOIN example

empid [PK] bigint	empname character varying (50)	manager_id bigint
1	Agni	3
2	Akash	4
3	Dharti	2
4	Vayu	3

→

mngr character varying (50)
Dharti
Vayu
Akash
Dharti

```
SELECT T2.empname, T1.empname
FROM emp AS T1
JOIN emp AS T2
ON T1.empid = T2.manager_id
```

UNION

The SQL **UNION** clause/operator is used to combine/concatenate the results of two or more SELECT statements without returning any duplicate rows and keeps **unique records**

To use this UNION clause, each SELECT statement must have

- The same number of columns selected and expressions
- The same data type and
- Have them in the same order
- **Syntax**

```
SELECT column_name(s) FROM TableA
UNION
SELECT column_name(s) FROM TableB
```

- **Example**

```
SELECT cust_name, cust_amount from custA
UNION
SELECT cust_name, cust_amount from custB
```

UNION ALL

In **UNION ALL** everything is same as **UNION**, it combines/concatenate two or more table but keeps all records, **including duplicates**

- **Syntax**

```
SELECT column_name(s) FROM TableA
UNION ALL
SELECT column_name(s) FROM TableB
```

- **Example**

```
SELECT cust_name, cust_amount from custA
UNION ALL
SELECT cust_name, cust_amount from custB
```

SUB QUERY

A **Subquery** or Inner query or a Nested query allows us to create complex query on the output of another query

- Sub query syntax involves two SELECT statements

- **Syntax**

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator
   ( SELECT column_name  FROM table_name WHE
```

SUB QUERY Example

Question: Find the details of customers, whose payment amount is more than the average of total amount paid by all customers

Divide above question into two parts:

1. Find the average amount
2. Filter the customers whose amount **>** average amount

	customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date
1	1	60	Cash	2020-09-24
2	2	30	Credit Card	
3	8	110	Cash	
4	10	70	mobile Payment	
5	11	80	Cash	

```
select avg(amount) from payment
-- find the average value
-- filter the customer data > avg value

SELECT *
FROM payment
WHERE amount > 164
```

output Messages Notifications

customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date
2	500	Credit Card	2020-04-27

Above code not best,every update karna hoga
So use dynamic
Uske liye use a subquery

Query | Query History

```
1  select avg(amount) from payment
2  -- find the average value
3  -- filter the customer data > avg value
4
5  SELECT *
6  FROM payment
7  WHERE amount > (select avg(amount) from payment)
8
9  select * from payment
```

Data output | Messages | Notifications

	customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date
1	2	500	Credit Card	2020-04-27

```
SELECT customer_id, amount, mode
from payment
where customer_id IN (select customer_id from customer)
```

Query | Query History

```
1  SELECT first_name, last_name
2  FROM customer c
3  WHERE EXISTS (  SELECT customer_id, amount
4                  FROM payment p
5                  WHERE p,customer_id = c.customer_id
6                  AND amount > 100)
7
8
```

Loading...

WINDOW FUNCTION

- **Window functions** applies aggregate, ranking and analytic functions over a particular window (set of rows).
- And **OVER** clause is used with window functions to define that window.

Give output one row per aggregation

The rows maintain their separate identities

WINDOW FUNCTION SYNTAX

```
SELECT column_name(s),
       fun( ) OVER (  [ <PARTITION BY Clause> ]
                      [ <ORDER BY Clause> ]
                      [ <ROW or RANGE Clause> ]  )
FROM table_name
```

Select a function

- Aggregate functions
- Ranking functions
- Analytic functions

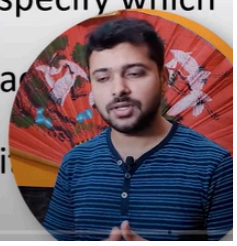
Define a Window

- PARTITION BY
- ORDER BY
- ROWS

WINDOW FUNCTION TERMS

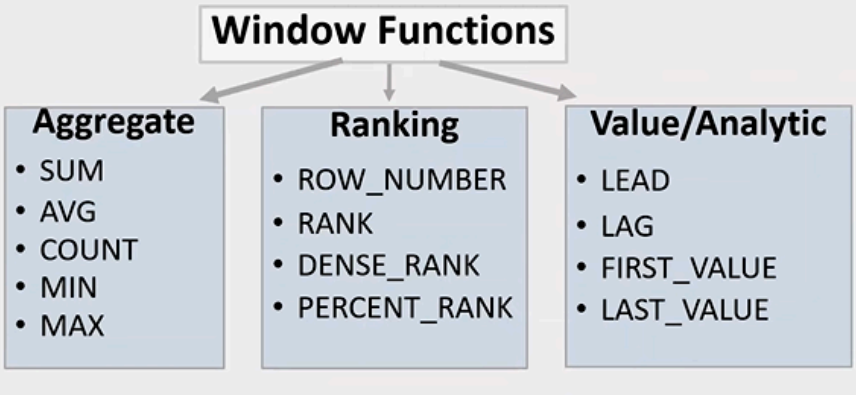
Let's look at some definitions:

- **Window function** applies aggregate, ranking and analytic functions over a particular window; for example, sum, avg, or row_number
- **Expression** is the name of the column that we want the window function operated on. This may not be necessary depending on what window function is used
- **OVER** is just to signify that this is a window function
- **PARTITION BY** divides the rows into partitions so we can specify which rows to use to compute the window function
- **ORDER BY** is used so that we can order the rows within ea This is optional and does not have to be specified
- **ROWS** can be used if we want to further limit the rows wi partition. This is optional and usually not used



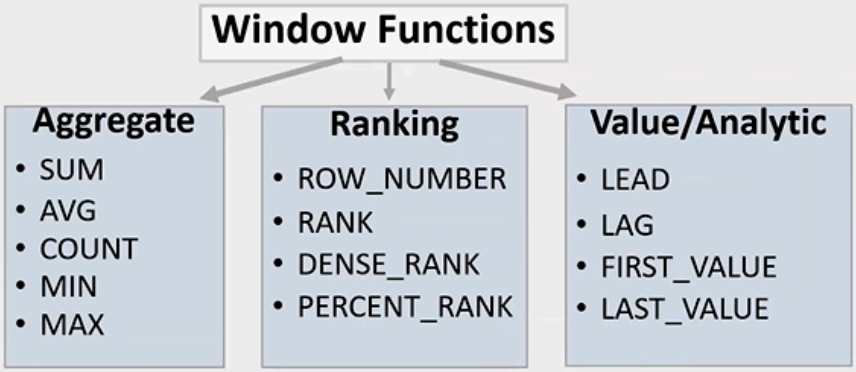
WINDOW FUNCTION TYPES

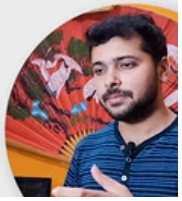
There is no official division of the SQL window functions into categories but high level we can divide into three types



WINDOW FUNCTION TYPES

There is no official division of the SQL window functions into categories but high level we can divide into three types





AGGREGATE FUNCTION Example

```
SELECT new_id, new_cat,
SUM(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Total",
AVG(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Average",
COUNT(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Count",
MIN(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Min",
MAX(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Max"
FROM test_data
```

new_id	new_cat	Total	Average	Count	Min	Max
100	Agni	300	150	2	100	200
200	Agni	300	150	2	100	200
500	Dharti	1200	600	2	500	700
700	Dharti	1200	600	2	500	700
200	Vayu	1000	333.33333	3	200	500
300	Vayu	1000	333.33333	3	200	500
500	Vayu	1000	333.33333	3	200	500

```
SELECT new_id, new_cat,
SUM(new_id) OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Total",
AVG(new_id) OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Average",
COUNT(new_id) OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Count",
MIN(new_id) OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Min",
MAX(new_id) OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Max"
FROM test_data
```

new_id	new_cat	Total	Average	Count	Min	Max
100	Agni	2500	357.14286	7	100	700
200	Agni	2500	357.14286	7	100	700
200	Vayu	2500	357.14286	7	100	700
300	Vayu	2500	357.14286	7	100	700
500	Vayu	2500	357.14286	7	100	700
500	Dharti	2500	357.14286	7	100	700
700	Dharti	2500	357.14286	7	100	700

AGGREGATE FUNCTION Example

NOTE: Above we have used: "**ROWS** BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING" which will give a SINGLE output based on all INPUT Values/PARTITION (if used)

Query | Query History

```
1  SELECT * FROM test_data
```

Data output | Messages | Notifications

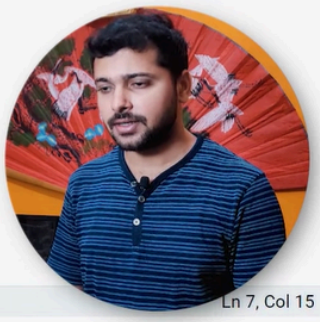
	new_id bigint	new_cat character varying
1	100	Agni
2	200	Agni
3	200	Vayu
4	300	Vayu
5	500	Vayu
6	500	Dharti
7	700	Dharti

Query | Query History

```
SELECT new_id, new_cat,
SUM(new_id)  OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Total",
AVG(new_id)  OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Average",
COUNT(new_id)  OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Count",
MIN(new_id)  OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Min",
MAX(new_id)  OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Max"
FROM test_data
```

Data output | Messages | Notifications

	new_id bigint	new_cat character varying	Total numeric	Average numeric	Count bigint	Min bigint	Max bigint
1	100	Agni	2500	357.1428571	7	100	700
2	200	Agni	2500	357.1428571	7	100	700
3	200	Vayu	2500	357.1428571	7	100	700
4	300	Vayu	2500	357.1428571	7	100	700
5	500	Vayu	2500	357.1428571	7	100	700
6	500	Dharti	2500	357.1428571	7	100	700
7	700	Dharti	2500	357.1428571	7	100	700



Total rows: 7 of 7 | Query complete 00:00:00.498 | Ln 7, Col 15

RANKING FUNCTION Example

```
SELECT new_id,
ROW_NUMBER()  OVER( ORDER BY new_id)  AS "ROW_NUMBER",
RANK()  OVER( ORDER BY new_id)  AS "RANK",
DENSE_RANK()  OVER( ORDER BY new_id)  AS "DENSE_RANK“,
PERCENT_RANK()  OVER( ORDER BY new_id)  AS "PERCENT_RANK"
FROM test_data
```

new_id	ROW_NUMBER	RANK	DENSE_RANK	PERCENT_RANK
100	1	1	1	0
200	2	2	2	0.166
200	3	2	2	0.166
300	4	4	3	0.5
500	5	5	4	0.666
500	6	5	4	0.666
700	7	7	5	1

No limit

Query | Query History

```
1  SELECT new_id,
2  ROW_NUMBER() OVER(ORDER BY new_id) AS "ROW_NUMBER",
3  RANK() OVER(ORDER BY new_id) AS "RANK",
4  DENSE_RANK() OVER(ORDER BY new_id) AS "DENSE_RANK",
5  PERCENT_RANK() OVER(ORDER BY new_id) AS "PERCENT_RANK"
6  FROM test_data
7
```

Data output | Messages | Notifications

	new_id bigint	ROW_NUMBER bigint	RANK bigint	DENSE_RANK bigint	PERCENT_RANK double precision
1	100	1	1	1	0
2	200	2	2	2	0.16666666666666
3	200	3	2	2	0.16666666666666
4	300	4	4	3	0.5
5	500	5	5	4	0.66666666666666
6	500	6	5	4	0.66666666666666
7	700	7	7	5	1

ANALYTIC FUNCTION Example

```
SELECT new_id,
FIRST_VALUE(new_id) OVER( ORDER BY new_id) AS "FIRST_VALUE",
LAST_VALUE(new_id) OVER( ORDER BY new_id) AS "LAST_VALUE",
LEAD(new_id) OVER( ORDER BY new_id) AS "LEAD",
LAG(new_id) OVER( ORDER BY new_id) AS "LAG"
FROM test_data
```

new_id	FIRST_VALUE	LAST_VALUE	LEAD	LAG
100	100	100	200	null
200	100	200	200	100
200	100	200	300	200
300	100	300	500	200
500	100	500	500	300
500	100	500	700	500
700	100	700	null	500

NOTE: If you just want the single last value from whole column, use: "**ROWS** BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING"

rnmcopy/postgres@PostgreSQL 14

No limit

Query | Query History

```
1  WITH my_cte AS (
2      SELECT  *, AVG(amount) OVER(ORDER BY p.customer_id) AS "Average_Price",
3      COUNT(address_id) OVER(ORDER BY c.customer_id) AS "Count"
4      FROM payment as p
5      INNER JOIN customer AS c
6      ON p.customer_id = c.customer_id
7  )
8  SELECT first_name, last_name, amount
9  FROM my_cte
```

Data output | Messages | Notifications

	first_name character varying (50)	last_name character varying (50)	amount bigint
1	Mary	Smith	60
2	Madan	Mohan	500
3	R	Madhav	250

Quick Assignment: WINDOW FUNCTION

Offset the LEAD and LAG values by 2 in the output columns ?

INPUT

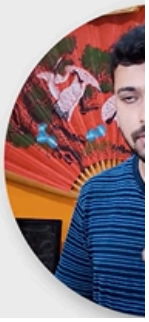
new_id
100
200
200
300
500
500
700

OUTPUT

new_id	LEAD	LAG
100	200	NULL
200	300	NULL
200	500	100
300	500	200
500	700	200
500	NULL	300
700	NULL	500

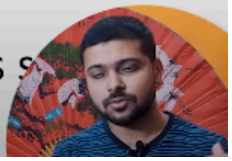
```
SELECT new_id,
LEAD(new_id, 2)  OVER( ORDER BY new_id) AS "LEAD_by2",
LAG(new_id, 2)  OVER( ORDER BY new_id) AS "LAG_by2"
FROM test_data
```

new_id	LEAD_by2	LAG_by2
100	200	null
200	300	null
200	500	100
300	500	200
500	700	200
500	null	300
700	null	500



Common Table Expression (CTE)

- A common table expression, or CTE, is a temporary named result set created from a simple SELECT statement that can be used in a subsequent SELECT statement
- We can define CTEs by adding a WITH clause directly before SELECT, INSERT, UPDATE, DELETE, or MERGE statement.
- The **WITH** clause can include one or more CTEs s
  by commas



Common Table Expression (CTE)

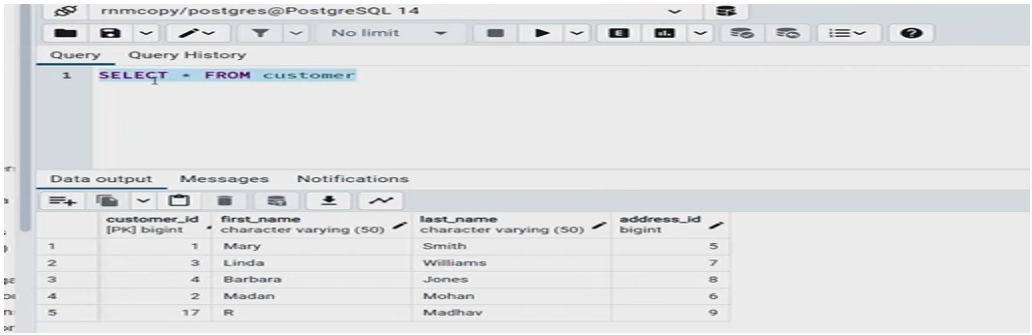
- **Syntax**

```
WITH my_cte AS (
        SELECT a,b,c
        FROM Table1 )
SELECT a,c
FROM my_cte
```

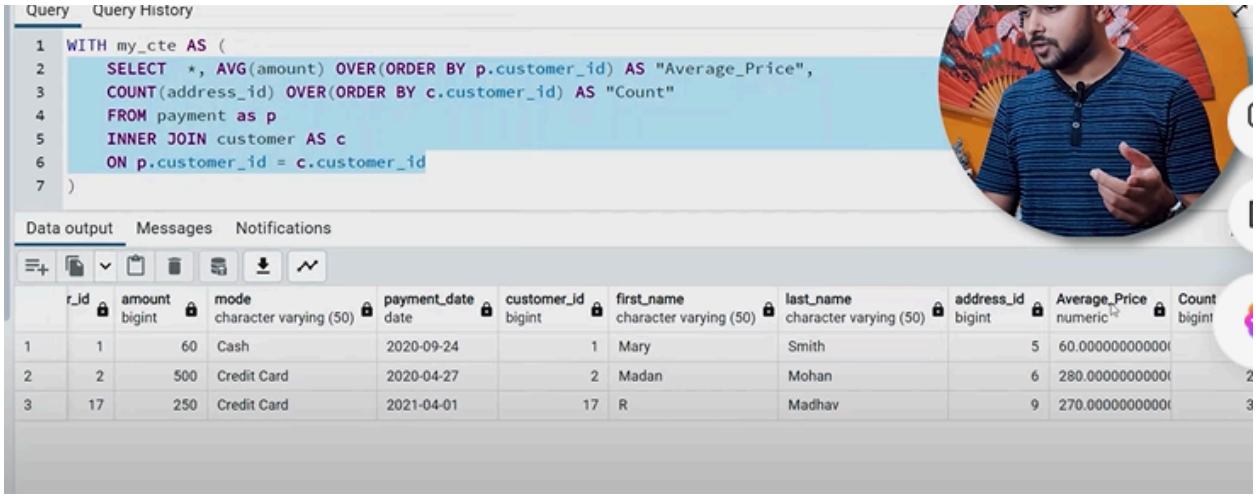
CTE query

Main query

The name of this CTE is my_cte, and the CTE query is SELECT a,b,c FROM Table1. The CTE starts with the **WITH** keyword, after which you specify the name of your CTE, then the content of the query in parentheses. The main query comes after the closing parenthesis and refers to the CTE. Here, the main query (also known as the outer query) is SELECT a,c FROM my_cte



	customer_id [PK] bigint	first_name character varying (50)	last_name character varying (50)	address_id bigint
1	1	Mary	Smith	5
2	3	Linda	Williams	7
3	4	Barbara	Jones	8
4	2	Madan	Mohan	6
5	17	R	Madhav	9



r_id	amount bigint	mode character varying (50)	payment_date date	customer_id bigint	first_name character varying (50)	last_name character varying (50)	address_id bigint	Average_Price numeric	Count bigint
1	60	Cash	2020-09-24	1	Mary	Smith	5	60.00000000000	
2	500	Credit Card	2020-04-27	2	Madan	Mohan	6	280.00000000000	2
17	250	Credit Card	2021-04-01	17	R	Madhav	9	270.00000000000	3

CASE Expression

- The CASE expression goes through conditions and returns a value when the first condition is met (like if-then-else statement). If no conditions are true, it returns the value in the ELSE clause.

- If there is no ELSE part and no conditions are true, it returns NULL.

CASE Statement Syntax

- **General CASE Syntax**

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE other_result
END;
```

- **Example:**

```
SELECT customer_id, amount,
CASE
    WHEN amount > 100 THEN 'Expensive product'
    WHEN amount = 100 THEN 'Moderate product'
    ELSE 'Inexpensive product'
END AS ProductStatus
FROM payment
```

Query | Query History

```
1  SELECT * FROM payment
```

Data output | Messages | Notifications

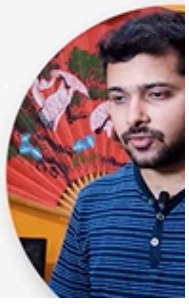
	customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date
1	1	60	Cash	2020-09-24
2	10	70	mobile Payment	2021-02-28
3	11	80	Cash	2021-03-01
4	2	500	Credit Card	2020-04-27
5	8	100	Cash	2021-01-26

Query | Query History

```
1  SELECT customer_id, amount,
2  CASE
3      WHEN amount > 100 THEN 'Expensive product'
4      WHEN amount = 100 THEN 'Moderate product'
5      ELSE 'Inexpensive product'
6  END AS ProductStatus
7  FROM payment
8
```

Data output | Messages | Notifications

	customer_id [PK] bigint	amount bigint	productstatus text
1	1	60	Inexpensive product
2	10	70	Inexpensive product
3	11	80	Inexpensive product
4	2	500	Expensive product
5	8	100	Moderate product



CASE Expression Syntax

- **CASE Expression Syntax**

```
CASE Expression
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    WHEN valueN THEN resultN
    ELSE other_result
END;
```

- **Example:**

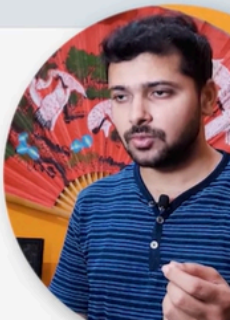
```
SELECT customer_id,
CASE amount
    WHEN 500 THEN 'Prime Customer'
    WHEN 100 THEN 'Plus Customer'
    ELSE 'Regular Customer'
END AS CustomerStatus
FROM payment
```

Query | Query History

```
1  SELECT customer_id,
2  CASE amount
3      WHEN 500 THEN 'Prime Customer'
4      WHEN 100 THEN 'Plus Customer'
5      ELSE 'Regular Customer'
6  END AS CustomerStatus
7  FROM payment
8
9  SELECT * FROM payment
```

Data output | Messages | Notifications

	customer_id [PK] bigint	customerstatus text
1	1	Regular Custom...
2	10	Regular Custom...
3	11	Regular Custom...
4	2	Prime Customer
5	8	Plus Customer



Most Asked SQL Interview Question

INPUT table

source character varying (20)	destination character varying (20)	distance integer
Mumbai	Bangalore	500
Bangalore	Mumbai	500
Delhi	Mathura	150
Mathura	Delhi	150
Nagpur	Pune	500
Pune	Nagpur	500

OUTPUT table

source character varying (20)	destination character varying (20)	distance integer
Mumbai	Bangalore	500
Mathura	Delhi	150
Nagpur	Pune	500

SQL Interview Question

source character varying (20)	destination character varying (20)	distance integer
Mumbai	Bangalore	500
Bangalore	Mumbai	500
Delhi	Mathura	150
Mathura	Delhi	150
Nagpur	Pune	500
Pune	Nagpur	500

INPUT table

source character varying (20)	destination character varying (20)	distance integer
Mumbai	Bangalore	500
Mathura	Delhi	150
Nagpur	Pune	500

OUTPUT

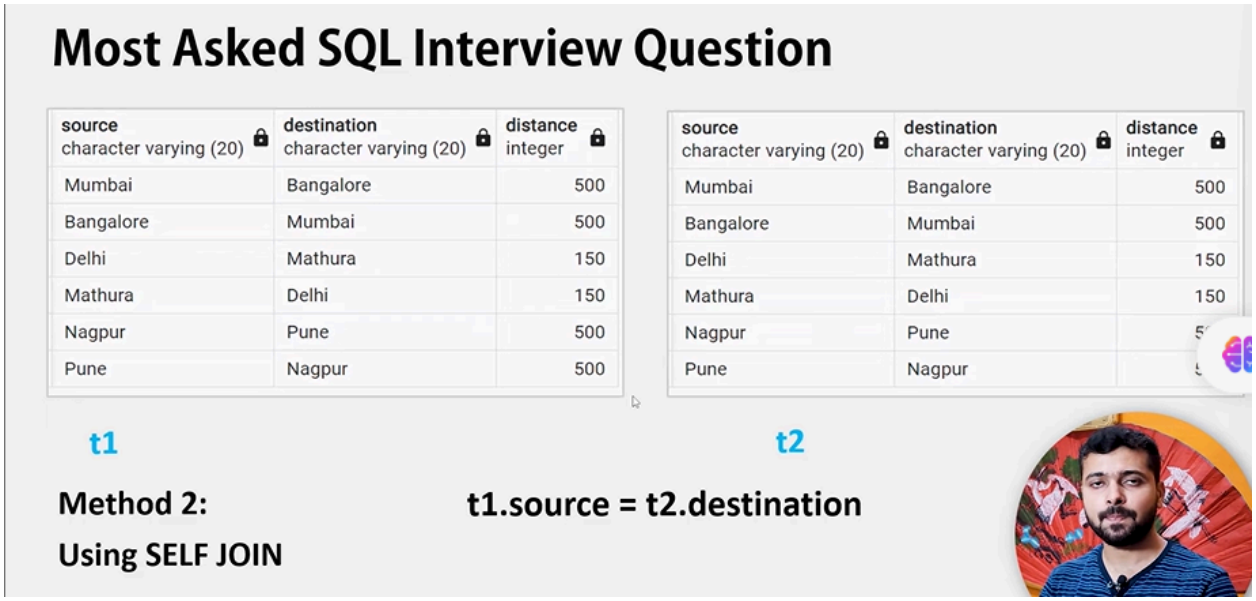
GREATEST() and LEAST() functions using

Query | Query History

```
1  SELECT greatest(source, destination), least(source, destination), max(distance)
2  FROM travel
3  group by greatest(source, destination), least(source, destination)
```

Data output | Messages | Notifications

	greatest character varying (20)	least character varying (20)	max integer
1	Mumbai	Bangalore	500
2	Mathura	Delhi	150
3	Pune	Nagpur	500



Important function:

ROUND

The ROUND function in SQL is used to round a numeric value to a specified number of decimal places. This is useful for controlling the precision of numeric data in your results.

Syntax:

```
ROUND(number, decimal_places)
```

1. Basic Rounding to Integer

SELECT ROUND(12.345) AS RoundedValue;

Output:

RoundedValue

12

example2

SELECT ROUND(12.345, 2) AS RoundedValue;

RoundedValue

12.35
COALESCE():

The `COALESCE` function in SQL is a powerful tool for handling NULL values. It allows you to specify multiple expressions and returns the first non-null expression among them. This is particularly useful for ensuring that you don't return NULL in your query results when you want a default value instead.
General Syntax:

```
COALESCE(expression_1, expression_2, ..., expression_n)
```

The function evaluates the expressions in order from left to right and returns the first non-null value.
If all the expressions evaluate to NULL, then COALESCE will return NULL.

Example:
COALESCE(SUM(O.order_amount), 0)
COALESCE(ROUND(SUM(P.price * Un.units) /
NULLIF(SUM(Un.units), 0), 2), 0)

LAG() and LEAD():window functions

2. We can use the `LAG()` and `LEAD()` window functions to look at the previous and next rows to compare values.

1. **LAG(num) OVER (ORDER BY id)**: Retrieves the `num` from the previous row in order of `id`.
2. **LEAD(num) OVER (ORDER BY id)**: Retrieves the `num` from the next row in order of `id`.

Syntax

LAG() Syntax

sql Copy code

```sql
LAG(column_name, offset, default_value) OVER (PARTITION BY partition_column ORDER BY order
```

- **column_name**: The column from which you want to retrieve the value.
- **offset**: The number of rows back from the current row from which to retrieve the value (default is 1).
- **default_value**: The value to return if the requested offset goes beyond the scope of the window (default is `NULL`).
- **PARTITION BY**: (Optional) Divides the result set into partitions to which the function is applied. If omitted, the entire result set is treated as a single partition.
- **ORDER BY**: Defines the order of the rows in each partition.

sql Copy code

```sql
ffset, default_value) OVER (PARTITION BY partition_column ORDER BY order_column)
```

LEAD() Syntax

sql Copy code

```
LEAD(column_name, offset, default_value) OVER (PARTITION BY partition_column ORDER BY orde
```

Summary

- **LAG()**: Accesses a value from a previous row.
- **LEAD()**: Accesses a value from a subsequent row.
- Both functions are commonly used for time series analysis and reporting to compare current values with previous or next values efficiently.

String Operation

The `SUBSTRING()` function in MySQL is used to extract a substring from a string. Its basic syntax is:

```
SUBSTRING(Col_name, start_position, length)
```

`Col_name`: The col name from which you want to extract the substring.
`start_position`: The position to start extracting the substring. The first character starts at position 1.
`length` (optional): (Number of characters) The number of characters to extract. If omitted, the substring will be extracted from the start position to the end of the string.
Examples:
Extract a substring from position 3, with length 5:

```
SELECT SUBSTRING('Hello, World!', 3, 5);
```

```
Output: 'llo, '
```

```
SELECT SUBSTRING(column_name, start_position, length)
FROM table_name;
```

UPPER(): Converts a string to uppercase.

```
SELECT UPPER(column_name)
FROM table_name;
```

LOWER(): Converts a string to lowercase.

```
SELECT LOWER(column_name)
FROM table_name;
```

CONCAT(): Concatenates two or more strings.

```
SELECT CONCAT(string1, string2, ...)
FROM table_name;
```

LENGTH(): Returns the length of a string (in bytes).

```
SELECT LENGTH(column_name)
FROM table_name;
```

INSTR(): Finds the position of the first occurrence of a substring.

```
SELECT INSTR(column_name, 'substring')
```

```
FROM table_name;
```

REPLACE(): Replaces occurrences of a substring within a string.

```
SELECT REPLACE(column_name, 'old_string', 'new_string')
FROM table_name;
```

LTRIM() and **RTRIM()**: Removes leading and trailing spaces from a string.

```
SELECT LTRIM(column_name)
FROM table_name;

SELECT RTRIM(column_name)
FROM table_name;
```

TRIM(): Removes both leading and trailing spaces (or specified characters) from a string.

```
SELECT TRIM([LEADING | TRAILING | BOTH] 'char' FROM
column_name)
FROM table_name;
```

```
SELECT TRIM(BOTH 'x' FROM username) AS
trimmed_username
FROM users;
```

LEFT() and RIGHT(): Extracts characters from the left or right of a string.

```
SELECT LEFT(column_name, length)
FROM table_name;

SELECT RIGHT(column_name, length)
FROM table_name;
```

Get the first 3 characters:

```
SELECT LEFT(username, 3) AS first_three_chars
FROM users;
```

REVERSE(): Reverses the string.

```
SELECT REVERSE(column_name)
FROM table_name;
```

GROUP_CONCAT:

`GROUP_CONCAT` is a SQL aggregate function used to concatenate values from multiple rows into a single string. It is especially useful when you want to combine related data from a group of rows into one field.

```
GROUP_CONCAT(expression [ORDER BY ...] [SEPARATOR 'string'])
```

Parameters

- **expression: The column or expression whose values you want to concatenate.**

- **ORDER BY: Optional. Specifies the order of the concatenated values. You can sort the values in ascending or descending order.**
- **SEPARATOR: Optional. Specifies a string that separates the concatenated values. The default separator is a comma (`,`).**

```
GROUP_CONCAT(DISTINCT project_name ORDER BY project_name) AS projects
```

REGEXP() (short for "regular expression") is used in SQL to perform pattern matching using regular expressions.. It allows you to check if a string matches a specific pattern, which is useful for validating formats such as email addresses, phone numbers, or any string that needs to adhere to a specific structure.

Syntax of `REGEXP`

```
column_name REGEXP 'pattern'
```

```
WHERE mail REGEXP
'^[a-zA-Z][a-zA-Z0-9._-]*@leetcode\\.com$'
```

`^`: Indicates the start of the string.

`[a-zA-Z]`: Matches any letter (uppercase or lowercase) as the first character.

`[a-zA-Z0-9._-]*`: Matches any combination of letters, digits, underscores, periods, and dashes.

The * means "zero or more" of these characters can appear after the first character.
@leetcode\\.com: Matches the exact string @leetcode.com. The double backslash is used to escape the dot, which is a special character in regular expressions (it matches any single character)
$: Indicates the end of the string.

Common Regular Expression Components

Anchors:

- ^: Asserts position at the start of a line.
- $: Asserts position at the end of a line.

Character Classes:

- [abc]: Matches any single character within the brackets (a, b, or c).
- [^abc]: Matches any single character not in the brackets.
- [a-z]: Matches any lowercase letter from a to z.
- [A-Z]: Matches any uppercase letter from A to Z.
- [0-9]: Matches any digit from 0 to 9.
- [a-zA-Z]: Matches any letter (uppercase or lowercase).

Quantifiers:

- *: Matches 0 or more occurrences of the preceding element.
- +: Matches 1 or more occurrences of the preceding element.
- ?: Matches 0 or 1 occurrence of the preceding element.
- {n}: Matches exactly n occurrences of the preceding element.
- {n,}: Matches n or more occurrences of the preceding element.
- {n,m}: Matches between n and m occurrences of the preceding element.

Special Characters:

- .: Matches any single character except newline.
- \: Escapes a special character (e.g., \. matches a literal period).
- |: Acts as an OR operator (e.g., abc|def matches "abc" or "def").

Finding Users with Numeric Names:

```
SELECT *
FROM Users
WHERE name REGEXP '^[0-9]+$';  -- Names that
consist only of digits
```

Selecting Users with Names Starting with "A" or "B":

```
SELECT *
FROM Users
WHERE name REGEXP '^[AB].*';  -- Names that start with A or B
```