SUB QUERY

SQL Tutorial In Hindi-13

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 WHERE ...**);

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

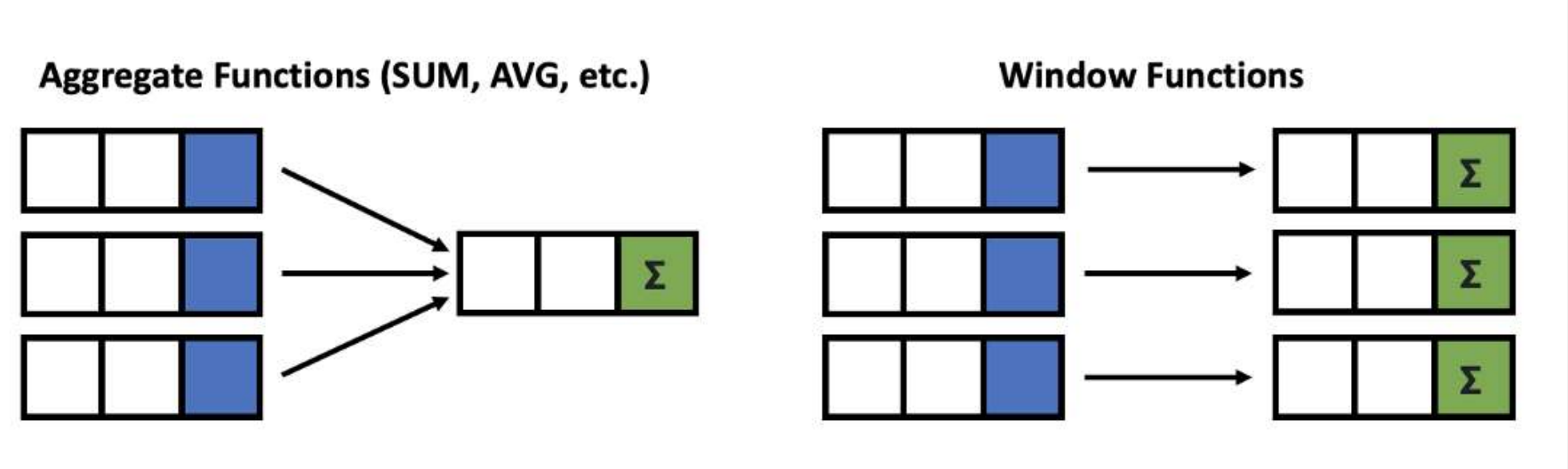
	customer_id [PK] bigint	amount bigint	mode character varying (50)	payment date
1	1	60	Cash	2020-09
2	2	30	Credit Card	2020-04
3	8	110	Cash	2021-01
4	10	70	mobile Payment	2021-02
5	11	80	Cash	2021-03

WINDOWS FUNCTION

SQL Tutorial In Hindi-14

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 whic rows to use to compute the window function
- **ORDER BY** is used so that we can order the rows within each partitio This is optional and does not have to be specified
- **ROWS** can be used if we want to further limit the rows within our 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 Functions

Aggregate

- SUM
- AVG
- COUNT
- MIN
- MAX

Ranking

- ROW_NUMBER
- RANK
- DENSE_RANK
- PERCENT_RANK

Value/Analytic

- LEAD
- LAG
- FIRST_VALUE
- LAST_VALUE

```
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

AGG
FU
E

```
ELECT new_id, new_cat,
UM(new_id)  OVER( ORDER BY new_id  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)  AS "Total",
VG(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"
ROM 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

AGG
FU
E

NOTE: Above we have used: "**ROWS** BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED OLLOWING" which will give a SINGLE output based on all INPUT Vaused)

```
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

RA
FU
E

```
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"

AN
FU
E

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

A
FU
Ass

CASE EXPRESSION

SQL Tutorial In Hindi-15

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.
- Also called CASE STATEMENT

CSV file

- payment csv file: https://bit.ly/41kJLRW

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
    WHEN amount = 100 THEN 'Moderate
    ELSE 'Inexpensive product'
END AS ProductStatus
FROM payment
```

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
```

COMMON TABLE EXPRESSION

SQL Tutorial In Hindi-16

Common Table Expression (CTE)

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

CSV files

- 👉 **customer csv file: https://bit.ly/3xGABBR**
- 👉 **payment csv file: https://bit.ly/3Zc0GUV**

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

CTE- Example

ASY

```
AS (
 SELECT  *, AVG(amount) OVER(ORDER BY
l) AS "Average_Price",
 COUNT(address_id) OVER(ORDER BY
) AS "Count"
 FROM payment as p
 INNER JOIN customer AS c
 ON p.customer_id = c.customer_id

ame, last_name
```

1. Example Multiple CTEs

```
WITH my_cp AS (
        SELECT  *, AVG(amount) OVER(ORDER BY p.customer_id)
AS "Average_Price",
        COUNT(address_id) OVER(ORDER BY c.customer_id) AS
"Count"
        FROM payment as p
        INNER JOIN customer AS c
        ON p.customer_id = c.customer_id
),
my_ca AS (
        SELECT *
        FROM customer as c
        INNER JOIN address AS a
        ON a.address_id = c.address_id
        INNER JOIN country as cc
        ON cc.city_id = a.city_id
)
SELECT cp.first_name, cp.last_name, ca.city, ca.country, cp.amount
FROM my_ca as ca , my_cp as cp
```

2. Example Advance

```
WITH my_cte AS (
  SELECT mode, MAX(amount) AS highe
SUM(amount) AS total_price
  FROM payment
  GROUP BY mode
)
SELECT payment.*, my.highest_price, m
FROM payment
JOIN my_cte my
  ON payment.mode = my.mode
  ORDER BY payment.mode
```

Now You Know All Concepts in SQL! ☺

Nest Step: Practice SQL Interview Question

Click Here