COUNT - SELECT COUNT (*) [or any column name]

This function returns the number of rows that matches a specified criteria

To ignore duplicates - SELECT COUNT(DISTINCT column_name)

To count multiple fields - SELECT COUNT(C_N), COUNT(C.N)

★ Filtering

WHERE - helps us to look at a specific area, table, Range, Column.

WHERE with Comparison operators.

```
SELECT
FROM
WHERE column_name > 1960;
```

★ Comparison operators

= Equals
<> Not equal to
> Greater than
< Less than
>= Greater than or equal to
<= Less than or equal to.

★ WHERE with Strings.

```
SELECT
FROM
WHERE Country = 'Japan'
```

★ Multiple Criteria

OR, AND, BETWEEN XOR	OR - When you need to satisfy at least one condition AND - When you need to satisfy all the criteria XOR - When you need to satisfy only one or the other condition, not both.

```
SELECT
FROM
WHERE COLOR = 'Yellow' OR length = 'Short';
```

OR

```
WHERE Color = 'Yellow' AND length = 'Short';
```

OR

```
WHERE buttons BETWEEN 1 AND 5;
```

- If a query has multiple filtering conditions, we will need to enclose the individual clauses in parentheses to ensure the correct execution order.

Example:

```
SELECT
FROM
WHERE (abba_ibba = 27 OR kab_ib = 2)
    AND (chombu = 'CP' OR cest = 'R');
```

Example - BETWEEN

```
SELECT
FROM
WHERE Column_name
    BETWEEN 1994 AND 2000 AND Column_name = 'UK'
```

★ Filtering Texts

1) LIKE _ - used to search for a pattern in a field.
   There are two wild cards used in LIKE.
   % match zero, one or many characters.
2) (under score) _ match a single character.

LIKE followed by	
'AL%'	- Find names starting with "AL"
'%son'	- Find names ending with "son"
'%est%'	- Find strings containing "est" (anywhere)
'A_e'	- Find strings starting with "A" & ending with "e"

LIKE followed by	
'_a%'	- Find strings where the second character is "a" 1) _ = exactly one character 2) followed by "a" 3) then any characters
'_n%'	- Flexible matching with fixed position 1) starts with "J" 2) third letter is "n" 3) followed by anything

Ex: "Sam" "Jenny" Second letter is "a"

Ex: "Jane" & "Jinx" Note: Does not match with "Jon" (third letter is "n", but only 3 letters in total)

3) NOT LIKE - opposite to LIKE
4) IN - Instead of using multiple OR function we can use IN

   Example: SELECT
   FROM WHERE release_years IN (1920, 1930, 1940)

5) LEFT - It is a string function that extracts a specified numbers of characters from the beginning (left side) of a string.

   LEFT (string, numbers_of_characters)

★ NULL Values

1) IS NOT NULL
2) IS NULL

★ Aggregate functions.

NUMERICAL DATA

AVG (), SUM(), MIN(), MAX(), COUNT ()

Each of the functions are used in SELECT statement.

NON-NUMERICAL DATA.

COUNT()
MIN ()
MAX ()
} Various data types.

★ Subset

ROUND (number_to_round, decimal_places)

Can only be used with numerical data types.

If you have to ROUND a number to the nearest 10's, 100's, 1000's etc then

ROUND (number_to_round, -1) - this is of nearest 10's

★ Arithmetic

+, -, * & /

Ex SELECT (4+3);

• Difference b/w aggregate & Arithmetic function is,

Ex:- function like SUM() calculates the entire Column. Where as arithmetic functions calculates along the rows (horizontally).

★ Sorting results

ORDER BY (by defult it Sort in ascending order)

ORDER BY Column_name DESC (for descending order)

For multiple fields

ORDER BY field_one, field_two (think of field_two as a tie_breakes)

Choosing fields in different orders.

```
SELECT birthdate, name
FROM
ORDER BY birth date, name DESC
```

★ Grouping data

GROUP BY is a clause used with aggregate functions (like SUM(), COUNT(), AVG(), MAX(), MIN()) to group rows that have the same values into summary rows.

In Simpler Words:

- Without GROUP BY, aggregate functions calculate on the entire table
- With GROUP BY, you split the table into small groups first & then perform the aggregate on each group Separately.

Imp Rules

Rule	Explanation
• Every Selected column must either be grouped or be inside an aggregate function	You cannot Select a column that's neither grouped or aggregated
GROUP BY, happens before ORDER BY	First SQL groups, then you can order the grouped results

Note: "GROUP BY, product_name, region;" means first by product, then inside each product by region

Final Summary.

- GROUP BY is essential for summarizing data.
- Always think "What column do I want to GROUP BY?" & "What aggregation do I want inside each group"
- Remember: HAVING filters groups. WHERE filters individual rows.

JOINS

INNER JOIN

What is INNER JOIN
- A type of SQL Join that returns only rows with matching values in both tables.
- It combines records where join condition evaluates to TRUE
- If there's no match, the row is excluded from the result.

Purpose of INNER JOIN
- To merge related data from two or more tables based on a common key.
- Common used in relational databases to normalize data & reduce redundancy.

How it works
- SQL compares the specified columns from both tables
- For each pair of rows, if the values match, the row is included in the output
- The join operates row by row, checking the ON condition

Syntax Structure
- Use INNER JOIN followed by the second table name.
- The ON clause defines the condition that links the tables.

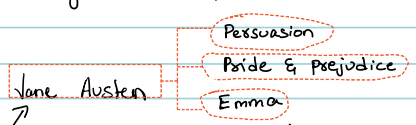
```
SELECT
FROM table1
INNER JOIN table2
ON table1.key = table2.key
```

Key Characteristics
- only matched records are returned
- Requires at least one common column (foreign key)
- Does not modify data - just queries & present it.
- Can be used with multiple joins in one query.

Performance Note
- Efficient with indexed column
- Fewer rows are returned compared to other joins (e.g. LEFT JOIN), which can reduce processing time

Use cases
- Fetching transaction with customer details
- Linking employee records with their departments
- Joining order data with product information.

Definition of relationship
- One to many relationship

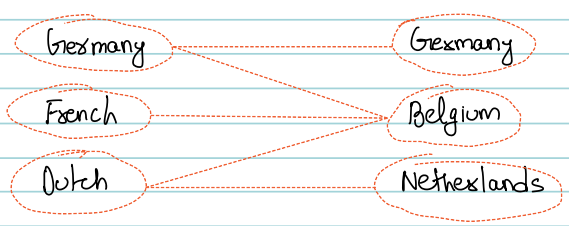


Ex:- One artist can produce many music

One to one relationship

one-to-one relationship imply unique pairings b/w entities & are therefore less common.

Many to Many



Join on Joins
Syntax

```
SELECT *
FROM left_table
INNER JOIN right_table
ON left_table.id = right_table.id
INNER JOIN another_table
ON left_table.id = another_table.id ;
```

Note: Research about the conditions & limits of the join on joins depending upon the use case

★ Multiple JOIN in SQL

- What are multiple Joins?
  - A multi table join Combines three or more tables in a single query
  - You perform multiple INNER JOIN's (or other joins) in sequence.
  - SQL processes the joins from left to right, one pair of tables at a time.

- Why use Multiple Joins?
  In a normalized database
  - Data is spread across many tables for efficiency
  - To generate useful reports (e.g. customers name + order date + product name), you often need to join several related tables.

- Syntax

```
SELECT
FROM table1
INNER JOIN table2 ON table1.key = table2.key
INNER JOIN table3 ON table2.key = table3.key;
```

- You can also join table3 on table1 if needed, depending on relations of the tables.

- How SQL processes Multiple Joins
  1) Joins table1 & table2 → temp result
  2) Joins temp results with table3
  3) Filters, groups or sorts the final results if needed.

★ Joining on multiple Columns

- What does it mean?
  You can use more than one column to define a match b/w rows.

Ex:

```
ON table1.id = table2.id AND table1.date = table2.date;
```

- This means a row will only join if both id & date match.

- When to use multiple join keys?
  - When tables have Composite keys
  - When you want to match by ID & exact date

★ LEFT JOIN (AKA OUTER JOIN)

- What it does
  - Returns all rows from the left table & the matching rows from the right table
  - if there is no match, the right table's columns will show NULL

Ex. use case: Show all customers, even if they haven't placed an orders.

- RIGHT JOIN – is same as LEFT JOIN but opposite (AKA OUTER JOIN)

- When to use INNER, LEFT OR RIGHT JOIN

	Situation	USE
1)	You want only full matches	INNER JOIN
2)	You want to keep all rows from one side	LEFT or RIGHT JOIN
3)	you wants gaps or missing links to show up	"
4)	You want to count or summarize even when something didn't happen	LEFT JOIN + COUNT() or SUM()

- FULL JOIN (AKA OUTER JOIN)
  - What it does?
  - A FULL OUTER JOIN returns all rows from both tables
  - If there's a match → combines the rows
  - If a row exists only in one table → includes the row with NULLs for the missing side

- Use case
  - Give me all customers & all orders, even if there's no match b/w them
  - You want to analyze unmatched records on both sides
  - You're auditing data completeness b/w systems.

- When to use FULL OUTER JOIN

	Tips	Why
1)	Use it with caution	Result set can be very large with lots of NULLs
2)	Useful for data reconciliation	Shows mismatches b/w systems or logs
3)	Filter NULLs to find gaps	Helps in debugging or data quality checks

- CROSS JOIN
  - A CROSS JOIN returns the cartesian product of two tables
  - Every row from the first table is paired with every row from the second table.
  - It does not require a join condition

  - If a table A has m rows & Table B has n rows, the result will be m x n rows.

- When to use CROSS JOIN

Scenario	Description
Generate all combination	Useful in simulations or testing
Create a matrix / Grid	E.g. every product with every region
Pairing unrelated entities	When data isn't naturally joinable but must be combined

Ex Scenario
If you have:
- 3 products
- 4 stores

A CROSS JOIN gives you all $3 \times 4 = 12$ product-store combinations.

- Practical use case
- Lets say you want to create a price list for each product in every currency the company operates
- Use CROSS JOIN to generate the full matrix of combination before applying conversion rates.

- SELF JOIN
What is a SELF JOIN?
- A self join is a regular join where a table is joined to itself.
- You use it to compare rows within the same table
- It requires using aliases to treat the same table as two separate ones.
- Think of it as - Match each row with other rows from the same table based on some relationship.

- Why use SELF JOIN?
- To compare or relate one row to another in the same table
Ex: 1) Hierarchical structure (e.g. employees reporting to managers)
2) Time comparison (e.g. Sales today vs yesterday)
3) Finding pairs of related values

Syntax

```
SELECT
FROM table_name AS A
JOIN table_name AS B
ON A.common_column = B.common_column;
```

- A & B are aliases for the same table.
- ON defines how rows should match.

Set Operations

★ SET Theory in SQL (UNION vs UNION ALL)

- What are set operations?

Set operations in SQL let you combine rows from two or more SELECT queries.
The key ones are:

1) UNION
2) UNION ALL
3) INTERSECT
4) EXCEPT

- Imagine you're managing two lists
  1) A list is from Event A - people who signed up.
  2) Another list is from Even B - people who also signed up there
- Some people may have signed up for both events. Now you want to combine the two list. This is where UNION & UNION ALL come in.

- What is UNION?

Think of UNION like merging two lists & saying:
"if someone appears twice, just include them once."

So:
1) You combine two sets of data
2) SQL looks through the results & remove duplicates.
3) The final list contains unique records only.

- What is UNION ALL?

- Now imagine you want to show everything, even if someone shows up twice
- It combines both lists, exactly as they are
- If a name appears twice, it shows up twice
- Nothing is removed.

- UNION ALL would be useful when:

- Counting how many times something happened
- Reviewing logs, sales or transactions
- or you care about the volume, not just uniqueness

Ex: How many total sign-ups did we have, including repeat sign-ups?

- Syntax Ex:

```
SELECT Column 1
FROM table 1
UNION
SELECT Column1
FROM table 2
WHERE / ORDER BY
```

Important Notes

1) Both SELECT queries must have same numbers of columns.
2) The columns should be of compatible types (same data types)

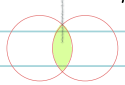
Example Situation

1) If you're making a summary report, use UNION to avoid duplicate entries.
2) If you're analyzing total transactions use UNION ALL to make sure every transaction in included.

- INTERSECT

-What it does?

- Returns only the rows that exist in both result sets
- Think of it like the overlap or "middle" part of Venn diagram



-Why use it?

- To find common records across two data sets
- Useful for matching entities, like users who are in both lists, or products in both catalogs.

Ex: Find customers who shopped in both 2023 & 2024

```
SELECT email FROM Customer_2023
INTERSECT
SELECT email FROM Customer_2024;
```

Note: Removes duplicates automatically
Not all databases support INTERSECT (MySQL doesn't; PostgreSQL & SQL Server do)

- EXCEPT Sometimes called as MINUS

- What it does?

- Returns rows from the first query that do not appear in the second
- Its a filter - keeping what's unique to the first set.

- Why use it?

- To find records that are missing in one data set compared to another.
- Very useful for auditing, Churn analysis, or detecting changes.

Ex: Find customers who bought last year but not this year.

```
SELECT email FROM customers_2023
EXCEPT
SELECT email FROM Customers_2024
```

Note: It also removes duplicates
Like INTERSECT, not supported in all databases

Joins vs Set operations

1) Conceptual Difference

Feature	Joins (INNER & OUTER)	Set operations (UNION, INTERSECT, etc)
Purpose	Combine columns from two (or more) related tables.	Combine rows from two (or more) result sets.
Layout result	Horizontal (side-by-side)	Vertical (top-to-bottom)
Data requirement	Based on relationship (e.g. foreign key match).	Based on structure (same number & type of columns
Typical use case	Merge related data from different tables.	Merge similar results from diff queries.
Condition required?	Yes (ON/USING clause)	No matching condition – but columns must align
Duplicates	Controlled by JOIN type	Controlled by UNION vs UNION ALL

2) How it work?

- INNER / OUTER JOIN
- Merges rows from two tables based on a matching condition (usually a key)
- Produces wider rows: Columns from both tables
- Returns only: 1) Matched rows (INNER JOIN)
  2) Unmatched rows from one or both tables (OUTER JOINs)

- Set operations (UNION, INTERSECT, EXCEPT)
- Stack results from two queries that have the same column count & types.
- Produces longer rows: each row is one complete result.
- No ON clause, just aligned SELECTs

3) Types of JOINs & Set operations

- JOINs (based on matching logic)

Type	Description
INNER JOIN	only rows that match in both tables
LEFT JOIN	All rows from left table + matches in the right table
RIGHT JOIN	All rows from right table + " " " left table
FULL JOIN	All rows from both tables, matched or not.

- Set operations (based on row inclusion)

Type	Description
UNION	Combines rows from both queries, remove duplicates
UNION ALL	Combines all rows from both queries, keeps duplicates
INTERSECT	Returns only rows present in both queries
EXCEPT	Returns rows from the first query not in the second

4) Structural Rules

JOIN
- Tables can have different numbers of columns
- Only matched rows are merged into one output row
- Columns must be joined based on a logical relationship (e.g. ON A.id = B.id).

Set operations
- Each SELECT must return the same number of columns, in the same order, with compatible data types.
- Otherwise, SQL throws an error.

5) When to use each.

- Use JOIN when:
- you're working with normalized relational data (e.g. users & orders, employees & dept
- You need to build one row of combined information from multiple sources.
- You want to preserve table structure & relationship.

- Use Set Operations when:
- You have two result sets with the same columns (even from diff tables)
- You want to create a unified or filtered list from multiple sources.
- You're dealing with log data, multiple regions, time-based data, etc.

6) Performance Considerations

Operations	Performance note
INNER JOIN	Fast if joined columns are indexed
OUTER JOIN	Slower - returns more data, especially with FULL JOIN
UNION	Slower than UNION ALL - it remove duplicates
UNION ALL	Faster - no duplication needed
INTERSECT & EXCEPT	May not be supported in all DB's; Slower due to comparisons.

Subquery

- What is a Subquery?

• A Subquery (or nested query) is a query placed inside another SQL query. It is usually embedded within the SELECT, INSERT, UPDATE or DELETE statements & can be placed inside:

• WHERE, FROM & SELECT

• Subqueries help you to write complex queries by breaking them down into smaller, simpler components

- Types of Subqueries

• Subqueries can be categorized into different types based on their placement & purpose:

Type of Subquery	Description
1) Single row	Return only one row & one column
2) Multi-row	Return more than one row.
3) Multiple-column	Return multiple column
4) Correlated	Subquery depends on data from outer query

1) Single-row Subquery

Ex: Find movies released in the same year as the movie "Inception".

```
SELECT Filmname, year_released
FROM Films
WHERE year_released = (
    SELECT year_released
    FROM Films
    WHERE Filmname = 'Inception'
);
```

• The inner query returns the year of release for "inception".
• The outer query finds films from the same year.

2) Multiple-row Subquery

Ex: Find movies in categories that members have listed as favorites.

```
SELECT Filmname, Category_Id
FROM Films
WHERE Category_Id IN (
    SELECT DISTINCT Category_Id
    FROM Fav_Category
);
```

• The inner query gets all distinct favorite categories from members
• The outer query selects films belonging to these categories.

3) Multiple-Column Subquery

Ex: Find details of members who share the same city & state as "John Doe"

```
SELECT first_name, last_name, city, state
FROM members_details
WHERE (city, state) = (
    SELECT city, state
    FROM members_details
    WHERE first_name = 'John' AND last_name = 'Doe'
);
```

- The subquery fetches both city & state
- Outer query finds all members with that exact combination

4) Correlated Subqueries

- In correlated subqueries, the inner query depends on data from the outer query. The inner query runs once for each row of the outer query.

Ex: List films whose rating is above average rating for their category.

```
SELECT film_name, rating, category_id
FROM film AS F1
WHERE rating > (
    SELECT AVG(rating)
    FROM Film AS F2
    WHERE F1.category_id = F2.category_id
```

- The inner query calculates the average rating for the category of the current row in the outer query
- Each film's rating is compared with this average

— Using Subqueries with Different Clauses.

1) Subquery in SELECT Clause

- Find each move & the number of members who list its category as favorite

```
SELECT
    film_name
    (SELECT COUNT(*)
     FROM fav_category As F
     WHERE F.category_id = films.category_id) AS FavCount
FROM films;
```

2) Subquery in FROM clause (inline view)

List average films rating/category.

```
SELECT Category, AvgRating
FROM (
```

```
    SELECT C.category, AVG(F.Rating) AS AvgRating
    FROM Films AS F
    JOIN Category AS C ON F.category_id = C.category_id
    GROUP BY C.category
) AS AvgCatRatings;
```

- Using Operators in Subqueries

•Subqueries often use special operators such as:

IN, NOT IN, ANY, ALL, EXISTS, NOT EXISTS

1) Ex using EXISTS - Check if there are any films in each category:

```
SELECT category_id
FROM Category AS C
WHERE EXISTS (
        SELECT 1
        FROM Films AS F
        WHERE F.category_id = C.category_id
);
```

2) Ex using NOT EXISTS - Find categories without any films

```
SELECT category_id
FROM Category AS C
WHERE NOT EXISTS (
        SELECT 1
        FROM Films AS F
        WHERE F.category_id = C.category_id
);
```

- Performance & Best Practices
  - Subqueries can be very readable but sometimes slower than joins
  - Consider JOINs for performance-intensive queries
  - Keep correlated subqueries minimal, as they run once for each row of outer query.
  - Always ensure subqueries return the correct type of data (single-row vs multiple-row)

- Difference b/w JOINs & Subqueries

Feature	JOIN	Subquery
Definition	Combines rows from two or more tables based on a related column	A query nested inside another query (SELECT, WHERE, FROM, etc.)
Execution	Processes tables simultaneously	Inner query executes first, then outer query uses the result
Result	Returns columns from multiple tables in a single result set	Typically returns a single value, list, or table for the outer query
Performance	Usually faster for simple relationships	Can be slower but sometimes more readable for complex logic
Use Cases	- Retrieving columns from multiple tables - Simple table relationships - Filtering based on another table	- Calculating aggregates for filtering - Existence checks (EXISTS/ NOT EXISTS) - Comparing a value to a computed result - Complex step-wise logic
Syntax Example	SELECT a.col1, b.col2 FROM table1 a JOIN table2 b ON a.id = b.id;	SELECT col1 FROM table1 WHERE id IN (SELECT id FROM table2 WHERE condition);
Readability	Better for straightforward table relationships	Better for multi-step or conditional logic
Scalability	More efficient for large datasets	May slow down with large data due to nested execution

When to Use Which?

- **Use JOIN** when you need to combine data from multiple tables efficiently.
- **Use Subquery** when you need step-wise filtering, aggregation, or existence checks.

CASE Statements

- CASE statements are used to perform conditional logic in queries, similar to if-else statements in programming languages. They are very useful when you want to return different values depending on condition applied to each row in your result set.

- Basic Syntax of CASE

```
CASE
    WHEN condition_1 THEN result1
    WHEN condition_2 THEN result2
   ELSE default_result
END
```

- You can use this inside a
SELECT
UPDATE
ORDER BY
WHERE

- Tips
1) Always include a ELSE to avoid unexpected NULL values.
2) The CASE expression returns only the first WHEN clause
3) You cannot have expressions like "Rating BETWEEN 2 AND 4" unless using Searched CASE

- Where can you use CASE ?
SELECT
WHERE
ORDER BY
GROUP BY
Inside aggregate functions [COUNT/SUM/AVG]

- Case statement Limitation

Limitation	Description
⚠️ Cannot return multiple columns	Each CASE returns a single value
⚠️ Becomes unreadable when overused	Use `WITH` clauses (CTEs) or temp views for complex logic
⚠️ Slower on large datasets	Especially with many nested `CASE` s and no indexing
⚠️ Can't be used outside query context	E.g., not for schema design or DDL statements

✅ CASE Statement Tips

1. **Always use an `ELSE` clause** to avoid returning `NULL` when no conditions match.
2. **Use descriptive aliases** (e.g., `AS outcome`, `AS win_type`) to make results easier to read.
3. **Keep CASE logic simple and readable**—too many conditions or nesting can make queries hard to debug.
4. **Use CASE inside aggregates** (`COUNT`, `SUM`, `AVG`) to calculate conditional totals, averages, or percentages.
5. **Use boolean flags (1/0) with `AVG()`** to calculate percentages.
6. **CASE can appear in SELECT, WHERE, ORDER BY, and even GROUP BY**—be strategic with placement.
7. **Use compound conditions** in `WHEN` clauses using `AND` / `OR` to refine logic.
8. **Do not return multiple columns** inside a CASE—only one value per CASE expression is allowed.
9. **Avoid overusing nested CASEs**—complex logic is better handled with CTEs or subqueries.
10. **Check for NULL results**—especially when no ELSE is used or when filtering with `CASE` in `WHERE`.
11. **CASE in WHERE** should end with `IS NOT NULL` to filter only matching rows.
12. **Use ROUND()** with `AVG()` + CASE to format numeric output (e.g., percentages).
13. **CASE can help label categories, groupings, or outcomes** dynamically in reports.
14. **Test your CASE logic step by step**—run partial queries to verify each condition.

- Syntax example

Exercise

In CASE of rivalry

Barcelona and Real Madrid have been rival teams for more than 80 years. Matches between these two teams are given the name *El Clásico* (The Classic). In this exercise, you will query a list of matches played between these two rivals, where Barcelona is the home team, categorizing as a home or away win depending on multiple conditions.

Instructions 100 XP

- Construct the `CASE` statement identifying who won each match.
- Fill in the logical operators to identify Barcelona or Real Madrid as the winner.

query.sql — Light mode

```sql
SELECT
    date,
    CASE WHEN hometeam_id = 8634 THEN 'FC Barcelona'
         ELSE 'Real Madrid CF' END as home,
    CASE WHEN awayteam_id = 8634 THEN 'FC Barcelona'
         ELSE 'Real Madrid CF' END as away,
    -- Identify possible home match outcomes
    CASE WHEN home_goal > away_goal AND hometeam_id = 8634 THEN 'Barcelona win!'
         WHEN home_goal < away_goal AND awayteam_id = 8633 THEN 'Real Madrid win!'
         ELSE 'Tie!' END AS outcome
FROM matches_spain
WHERE hometeam_id = 8634 AND awayteam_id = 8633;
```

Exercise

Filtering your CASE statement

Let's generate a list of matches won by Italy's *Bologna* team! There are quite a few additional teams in the two tables, so a key part of generating a usable query will be using your `CASE` statement as a filter in the `WHERE` clause.

`CASE` statements allow you to categorize data that you're interested in -- and exclude data you're not interested in. In order to do this, you can use a `CASE` statement as a filter in the `WHERE` statement to remove output you don't want to see.

Here is how you might set that up:

```
SELECT *
FROM table
WHERE
    CASE WHEN a > 5 THEN 'Keep'
         WHEN a <= 5 THEN 'Exclude' END = 'Keep';
```

In essence, you can use the `CASE` statement as a filtering column like any other column in your database. The only difference is that you *don't* alias the statement in `WHERE`.

query.sql — Light mode

```sql
SELECT
    season,
    date,
    home_goal,
    away_goal
FROM matches_italy
WHERE
    -- Find games where home_goal is more than away_goal
    CASE WHEN hometeam_id = 9857 AND home_goal > away_goal THEN 'Bologna Win'
        -- Find games where away_goal is more than home_goal
        WHEN awayteam_id = 9857 AND away_goal > home_goal THEN 'Bologna Win'
        -- Exclude games not won by Bologna
        END IS NOT NULL;
```

Run Code | Submit Answer

query result | teams_italy | matches_italy

Exercise

COUNT using CASE WHEN

Do the number of soccer matches played in a given European country differ across seasons? We will use the European Soccer Database to answer this question.

You will examine the number of matches played in 3 seasons within each country listed in the database. This is much easier to explore with each season's matches in separate columns. Using the `country` and unfiltered `match` table, you will count the number of matches played in each country during the 2012/2013 and 2013/2014 seasons.

Instructions 100 XP

query.sql — Light mode

```sql
SELECT
    c.name AS country,
    -- Count matches in 2012/13
    COUNT(CASE WHEN m.season = '2012/2013' THEN m.id END) AS matches_2012_2013,
    -- Count matches in 2013/14
    COUNT(CASE WHEN m.season = '2013/2014' THEN m.id END) AS matches_2013_2014
FROM country AS c
LEFT JOIN match AS m
ON c.id = m.country_id
GROUP BY country;
```

Exercise

Filtering and totaling using CASE WHEN

You can use `CASE` statements to apply a filter and perform a calculation, by writing the statement inside an aggregate function such as `SUM()`!

In this exercise, your goal is to filter for a specific team (Real Sociedad) and calculate their total home and away goals per season.

Instructions 100 XP

query.sql — Light mode

```sql
SELECT season,
    -- SUM the home goals
    SUM(CASE WHEN hometeam_id = 8560 THEN home_goal END) AS home_goals,
    -- SUM the away goals
    SUM(CASE WHEN awayteam_id = 8560 THEN away_goal END) AS away_goals
FROM match
-- Group the results by season
GROUP BY season
```

Exercise

Calculating percent with CASE and AVG

`CASE` statements will return any value you specify in your `THEN` clause. This is an incredibly powerful tool for robust calculations and data manipulation when used in conjunction with an aggregate statement. One key task you can perform is using `CASE` inside an `AVG` function to calculate a percentage of information in your database.

Here's an example of how you set that up:

```
AVG(CASE WHEN condition_is_met THEN 1
         WHEN condition_is_not_met THEN 0 END)
```

With this approach, it's important to accurately specify which records count as `0`, otherwise your calculations may not be correct!

Your task is to examine the number of wins, losses, and ties in each country. The `matches` table is filtered to include all matches from the 2013/2014 and 2014/2015 seasons.

query.sql — Light mode

```sql
SELECT
    c.name AS country,
    -- Calculate the percentage of tied games in each season
    AVG(CASE WHEN m.season='2013/2014' AND m.home_goal = m.away_goal THEN 1
             WHEN m.season='2013/2014' AND m.home_goal <> m.away_goal THEN 0
             END) AS ties_2013_2014,
    AVG(CASE WHEN m.season='2014/2015' AND m.home_goal = m.away_goal THEN 1
             WHEN m.season='2014/2015' AND m.home_goal != m.away_goal THEN 0
             END) AS ties_2014_2015
FROM country AS c
LEFT JOIN matches AS m
ON c.id = m.country_id
GROUP BY country;
```

Run Code | Submit Answer

Common Table Expression

- In SQL there are multiple ways to break down complex problems into manageable steps. The three most useful tools are:

1) Subqueries (both regular & correlated)
2) Common table expression
3) Joins.

- Each of this, has a specific flavor & purpose, kind of like choosing the right tool from a toolbox depending on whether you're fixing a bicycle or building a book shelf.

CTE - SQL's "Staging Area"

- Imagine you're working on a complex report with multiple steps. Instead of stacking everything into one giant mess of a query, CTE's lets you pause, define a smaller step (like a temp table), give it a name & refer back to it as if its part of your original table.

Example analogy: Think of a CTE as drafting a mini-spreadsheet inside your spreadsheet, just for calculations that make the final result easier.

Syntax:

```
WITH name AS (
    SELECT
    ...
    ...
)

SELECT ....
.....;
```

Why its helpful?
1) Makes your queries clean & readable
2) Lets you build step-by-step logic
3) Easier to debug & reuse.

- CTEs Vs Subqueries Vs Correlated Subqueries

Technique	Think of it as	Behavior
Subquery	A quick side note inside a sentence	Executes independently of outer query
Correlated Subquery	A follow up question for every row	Runs once per row in outer query
CTE	A named draft sheet you build & refer to later.	Runs first, results used like a temp table

- Ask yourself

- Do I need to use this result in multiple places? → CTE
- Is this a one-time calculation? → Subquery
- Do I need this to depend on each row of the outer table → Correlated Subquery.

Window Function

- They allow to perform calculations like aggregation on a specific subset of data without losing the level of details of rows.
- Similar to GROUP BY but here we don't lose details.

Ex. GROUP BY

what is the total sales for each product?

SLNo.	Prod.	Sales
1	Cap	10
2	Cap	30
3	Car	5
4	Car	20

Prod.	Total Sales.	
Cap	40	Σ
Car	25	Σ

- So basically in GROUP BY the results are squeezed & grouped together.

- The disadvantage is that we always have to Group the all agregating dimension in the query.
- Returns a single row for each group

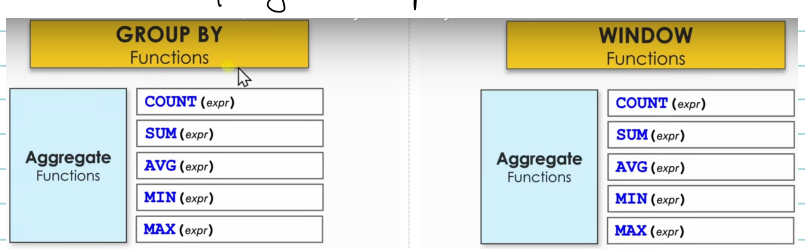
Window

what is the total sales for each product?

SLNo.	Prod.	Sales		SLNo.	Prod.	Sales	
1	Cap	10		1	Cap	40	Σ
2	Cap	30		2	Cap	40	Σ
3	Car	5		3	Car	25	Σ
4	Car	20		4	Car	25	Σ

- Each row will be executed individually, row-by row
- Returns a result for each row

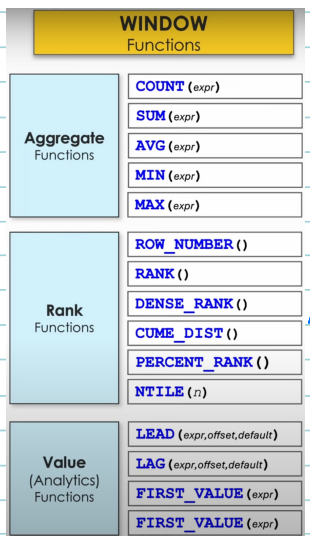
- For simple aggregations use GROUP BY. But if the detail of returned query is imp then use Window function

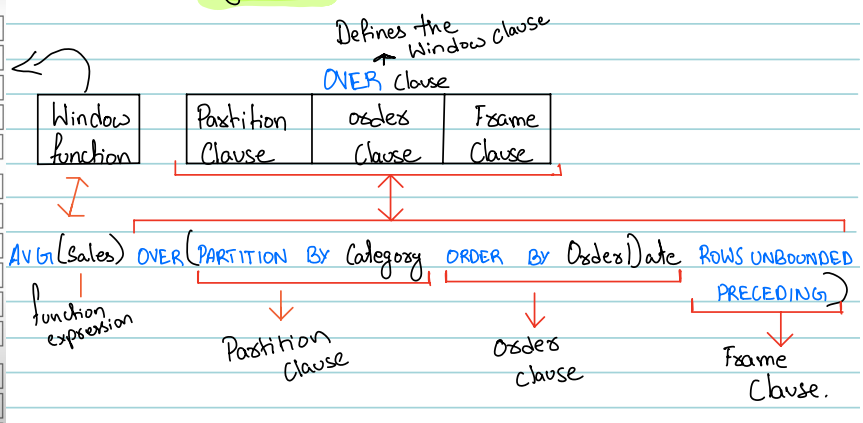


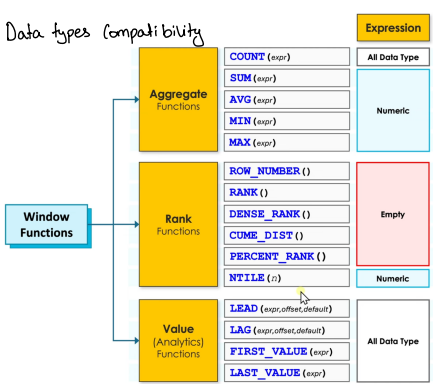
Aggregate function Comparison.

- However Window functions have way more compared to GROUP BY

Syntax







PARTITION BY: · Divides the result set in partitions (Window). Similar to GROUP BY clause.
· We can combine multiple column to partition at a time

ORDER BY : Sort the data within a window (Ascending or Descending)

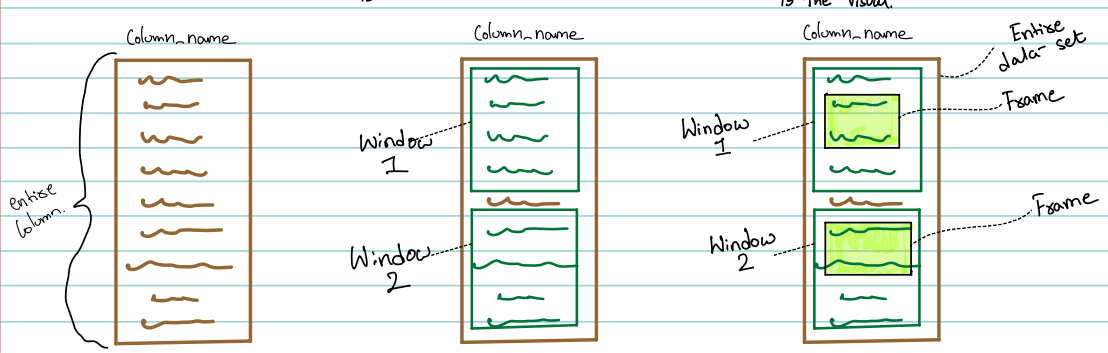
ROWS UNBOUNDED PRECEDING: Defines a subset of rows within each window that is relevant for the calculation

- Lets understand frame clause.

- In Window function if we don't use PARTITION BY clause then the entire column will be considered

if we only mention PARTITION BY (in Window function) then below is the visual of how it works at row level.

if we mention frame clause then below is the visual



- Lets understand the Syntax

AVG(Sales) OVER (PARTITION BY Category ORDER BY OrderDate

Frame Clause

ROWS BETWEEN CURRENT ROW AND UNBOUNDED PRECEDING)

Frame types: ROWS, RANGE

Frame Boundary (Lower Value)	Frame Boundary (Higher Value)
CURRENT ROW	CURRENT ROW
N PRECEDING	N PRECEDING
N FOLLOWING	N FOLLOWING
UNBOUNDED PRECEDING	UNBOUNDED PRECEDING
UNBOUNDED FOLLOWING	UNBOUNDED FOLLOWING

Roles: 1) Frame Clause can only be used together with ORDER BY Clause
2) Lower boundry value must be before the higher value.

Note: Watch /chatgpt the frame clause to understand deeply.

- Rules/Limitation of Window Clause

1) Window functions can be used only in SELECT & ORDER BY clauses.
2) Nesting Window functions in not allowed
3) SQL execute Window functions after WHERE clause
4) Window function can be used together with GROUP BY in the same query, only if the same columns are use

★ f(x) Window expression

Empty — RANK() OVER (ORDER BY order date)

Column — AVG (Sales) OVER (ORDER BY order date)

Numbers — NTILE (2) OVER (ORDER BY order date)

Multiple Arguments — LEAD(Sales, 2, 10) OVER (ORDER BY order date)

Conditional Logic — SUM (CASE WHEN Sales > 100 THEN 1 ELSE 0 END) OVER (ORDER BY order date)

Window Aggregate Func

Aggregate Functions | Syntax

	Expression	Partition clause	Order clause	Frame clause
COUNT	All data type	optional	optional	optional.
SUM	Numeric Values			
AVG	Numeric Values			
MIN	Numeric Values			
MAX	Numeric Values			

COUNT (exp) - Returns the number of rows in a window
SUM (exp) - Returns the sum of values in a window
AVG (exp) - Returns the average of values in a window
MIN (exp) - Returns the minimum value in a window
MAX (exp) - Returns the maximum value in a window

★ COUNT Window function
- When
- COUNT (*) OVER (PARTITION BY Product)
- Counts all the rows in a table regardless of whether any value is NULL
- To avoid including the count of NULL we basically mention the column names.

COUNT (Sales) OVER (PARTITION BY Product)

- Use Cases
1) Overall Analysis - total orders in the table
2) Category Analysis - orders / customers / product.
3) Quality Checks - Compare COUNT (*) vs COUNT (sales) to quickly detect nulls.
4) Duplicate detection - COUNT (*) OVER (PARTITION BY order_id) flags duplicate primary key values.

★ SUM Window function

- Returns the Sum of values within a window
- NULL values are ignored by default in the Summation
- Used for performance comparison - quickly see which product or region leads in revenue

★ AVG Window function

- Returns averages of values within a window
- NULL values are ignored unless the business treats nulls as zero - then use COALESCE (Sales, 0)

MIN() & MAX()

- Returns the lowest value within a window
- Returns the highest value within a window

Use Cases
1) Top/Bottom performers – pull rows where Sales = MAX(sales) OVER() to find top orders
2) Deviation Analysis - Compute Sales – MIN(Sales) OVER() or MAX(Sales) OVER() - Sales to gauge distance from extremes.

★ Running vs Rolling totals

- Both use SUM() with an ORDER BY inside OVER, but differ by their frame
- It can be used for tracking like – Tracking current Sales with target Sales
- It can be used for trend analysis - Providing insights into historical patterns.

- They aggregate sequence of members & the aggregation is updated each time a new member is added.

1) Running total – Aggregate all values from beginning up to the current point without dropping off older data.

```
SUM(sales) OVER(
    ORDER BY    order_date
    ROW BETWEEN UNBOUNDED PRECEDING AND CURREN ROW)
    AS running_total
```

2) Rolling total – Aggregate all values within a fixed time window (eg. 30 days). As new data is added, the oldest data point will be dropped.

```
SUM(sales) OVER(
    ORDER BY    order_date
    ROW BETWEEN 2 PRECEDING AND CURREN ROW)
    AS running_total
```

- In the window function if the frame is not defined by default it runs as ROW BETWEEN UNBOUNDED PRECEDING AND CURREN ROW), which is running total

- If the the frame is defined as ROW BETWEEN Nth PRECEDING AND CURREN ROW) then it will be Rolling total

★ Moving Average

- A Specialized rolling calculation using AVG instead of SUM

```
AVG(Sales) OVER(
  ORDER BY   Order_date
  ROW BETWEEN  1 PRECEDING AND 1 FOLLOWING)
  AS  Moving_Avg
```

- This gives a Centered 3-point average per product over time

★ Key take aways

1) Window aggregate functions delivers both Summary metrics & full-details rows.
2) Handle nulls explicitly (COUNT(col), COALESCE, filters)
3) Use PARTITION BY for group-specific metrics, ORDER BY + frames for time-Series & running/rolling Computations
4) Leverage MIN/MAX for extreme-value analysis & COUNT for quality checks (nulls duplicates)

★ Use Cases of Window aggregate function

- Overall analysis
- Total per groups Analysis
- Past-to-Whole Analysis
- Comparison Analysis: AVG & Extreme: Highest/Lowest
- Identify Duplicates
- outliers detection
- Running total
- Rolling total
- Moving Average

Window Ranking Function

Types of Ranking functions

	Expression	Partition clause	order clause	Frame clause
ROW_NUMBER () RANK () DENSE_RANK () CUME_DIST () PERCENT_ RANK () NTILE ()	Empty Numbers	optional	Required	Not allowed

ROW_NUMBER ()	Assign a unique numbers to each in a window	ROW_NUMBER () OVER (ORDER BY Sales)
RANK ()	Assign a rank to each row in a window, with gaps	RANK () OVER (ORDER BY Sales)
DENSE_RANK ()	Assign a rank to each row in a window, without gaps	DENSE_RANK () OVER (ORDER BY Sales)
CUME_DIST ()	Calculates the cumulative distribution of a value within a set of values	CUME_DIST () OVER (ORDER BY Sales)
PERCENT_ RANK ()	Returns the percentile ranking numbers of a row	PERCENT_RANK () OVER (ORDER BY Sales)
NTILE ()	Divides the rows into a specified numbers of approximately equal group	NTILE () OVER (ORDER BY Sales)

★ ROW_NUMBER

Definition: Assigns a unique, Sequential integers to each row within a partition of a result set, starting a 1 for the first row of the window.

Ex:-

Sales	Rank
100	1
20	2
30	3
80	4
80	5
33	6

We have a tie here

So basically ROW_Number () assigns a unique rank to each of the row

Use Case

- Imagine you're asked to "Show me the top-selling product in each category
- You'd use ROW_Numbers to rank products within categories & then pick only the row with numbers 1 from each row

★ RANK()

Definition - Rank also numbers the rows - but it handles the ties. If two rows have the same value, they get the same rank & the next rank is skipped

Ex:

Sales	Rank
100	1
20	2
30	3
80	4
80	4
33	6

RANK() assigns a same rank for both identical value

Leaves a gap in Ranking after a tie

- It is useful when ranking competitors, products or scores - Where the tie is important & you want the same position reflected.

★ DENSE_RANK

- Very Similar to RANK(), but it doesn't skip numbers. It still gives tied rows the same rank but the next rank just goes up by one

Ex:

Sales	Rank
100	1
20	2
30	3
80	4
80	4
33	5
27	6

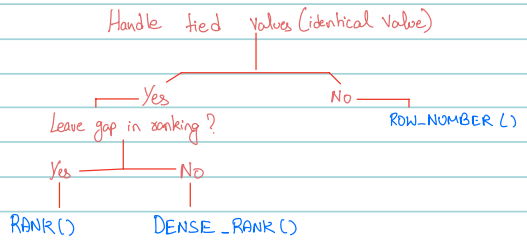
tie

Same ranking

No gaps in rank sequence

- When you want to preserve ranks for ties, but also avoid gaps in the numbering - like in product tiers or level-based systems.

- Among the ranking & Row numbering functions - Which one to use?



★ NTILE(n)

- This divides your result into "n" equal-sized groups (or "tiles") & labels each row with number of the group it falls into

- Data Analyst use it for Data Segmentation
- Data Engineer use it for equilizing load processing

NTILE(2) OVER (ORDER BY Sales DESC)

Sales	NTILE
100	1
300	1
450	1
71	1
93	2
21	2
62	2

Bucket size = $\frac{\text{Numbers of Rows}}{\text{Numbers of Buckets}} = \frac{7}{2}$

$3.5 \approx 4$

- Here if the No. of rows was even numbers then the divisio would be either 3 or 4 rows
- Since its odd numbers the easlies groups gets 1 extra row

- Used for grouping customers or products into quantiles like
  1) Top 25%.
  2) Bottom 25%.
- The groups are based on row count, not on the values themselves. So its not the same as dividing by actual percentages (like PERCENT_RANK)

★ CUME_DIST (cumulative distribution)

- This tells you what percentage of rows have values less than or equal to the current row.
- It gives a sense of how far along the row is in the dataset

CUME_DIST() - Cumulative distribution calculates the distribution of data points within a Window

$$\frac{\text{Position Nr}}{\text{Numbers of rows}}$$

CUME_DIST Calculates the relative position of a specified value in a group of values.

Sales	Dist
→ 100	0.2
80	
90	
20	
50	

CUME_DIST = $\frac{\text{Numbers of rows} \le X}{\text{Total Numbers of rows}} = \frac{1}{5} = 0.2$

Sales	Dist
100	0.2
80	0.6
→ 90	0.6
20	
50	

CUME_DIST = $\frac{\text{Numbers of rows} \le X}{\text{Total Numbers of rows}} = \frac{3}{5} = 0.2$

Sales	Dist
100	0.2
80	0.6
90	0.6
20	0.8
→ 50	1

CUME_DIST = $\frac{\text{Numbers of rows} \le X}{\text{Total Numbers of rows}} = \frac{1}{5} = 0.2$

- Useful in analysis for understanding percentile position or creating dynamic scoring system.
- Great in grading systems, benchmarking or financial performance.

★ PERCENT_RANK

- This shows the relative rank of row compared to the rest, normalized b/w 0 & 1.
- Unlike CUME_DIST, it doesn't include the current row's position when calculating the percentage

$\frac{\text{Position Nr} - 1}{\text{Numbers of rows} - 1}$ (Returns the percentile ranking numbers of a row)

	Sales	Rank	Dist
Current row →	20	1	0
	50	2	
	60	3	
	80	4	
	100	5	

$\frac{\text{Position Nr} - 1}{\text{Numbers of rows} - 1} = \frac{0}{4} = 0$

	Sales	Rank	Dist
	20	1	0
	50	2	0.25
→	60	3	0.5
	80	4	
	100	5	

$\frac{\text{Position Nr} - 1}{\text{Numbers of rows} - 1} = \frac{2}{4} = 0.5$

	Sales	Rank	Dist
	20	1	0
	50	2	0.25
	60	3	0.5
	80	4	0.75
→	100	5	1

$\frac{\text{Position Nr} - 1}{\text{Numbers of rows} - 1} = \frac{4}{4} = 1$

- Perfect for converting ranks into percentiles
- Used when visualizing ranks as percentages in dashboards or reports – e.g. "this product is in the top 10%"

★ Window RANK Func Summary

Types
- Integers
  - ROW_NUMBER
  - RANK
  - DENSE_RANK
  - NTILE
- Percentage based Ranking
  - CUME_DIST()
  - PERCENT_RANK()

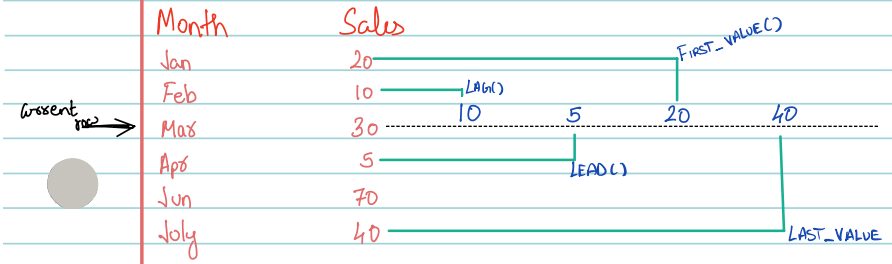
Rules
- Expression — Empty
- ORDER BY — Required
- FRAME — Not allowed

Use case
- Top N Analysis
- Bottom N Analysis
- Identify & Remove Duplicates
- Assign Unique ID & Pagination
- Data Segmentation
- Data Distribution Analysis
- Equlizing load processing

Window Value Function

LAG (exp, offset, default)	Returns the value from a previous row	LEAD (sales, 2, 0) OVER (ORDER BY orderdate)
LEAD (exp, offset, default)	Returns the value from a subsequent row	LAG (sales, 2, 0) OVER (ORDER BY orderdate)
FIRST_VALUE (exp, offset, default)	Returns the first value in a window	FIRST_VALUE (Sales) OVER (ORDER BY orderdate)
LAST_VALUE (exp, offset, default)	Returns the last value in a window	LAST_VALUE (Sales) OVER (ORDER BY orderdate)

	Expression	Partition	Order	Frame
LAG (exp, offset, default) LEAD (exp, offset, default) FIRST_VALUE (exp, offset, default) LAST_VALUE (exp, offset, default)	All data types	optional	Required	Not allowed optional Should used



LEAD() & LAG()

LEAD() function lets you to access a value from the next row within a window.

LAG() function is opposite to LEAD() which access a value from the previous row within a window

optional Required

LEAD (Sales, 2, 10) OVER (PARTITION BY Prod_ID ORDER BY order_Date)

- Expression is required (Any data type)
- offset (optional): Number of rows forward or backward from current row. Default = 1
- Default value (optional): Returns default value if next/previous is not available! Default = NULL

`LEAD(Sales) OVER (ORDER BY Month)`

Ex: Find Sales of the next month

Month	Sales	LEAD
Jan	20	10
Feb	10	30
Mar	30	5
Apr	5	NULL

`LAG(Sales) OVER (ORDER BY month)`

Ex: Find Sales of previous month

Month	Sales	LAG
Jan	20	NULL
Feb	10	20
Mar	30	10
Apr	5	30

`LEAD(Sales,2,0) OVER (ORDER BY Month)`

Ex: Find the Sales for the two months ahead

Month	Sales	LEAD
Jan	20	30
Feb	10	5
Mar	30	0
Apr	5	0

`LAG(Sales,2,0) OVER (ORDER BY month)`

Find the Sales for the two months ago

Month	Sales	LAG
Jan	20	0
Feb	10	0
Mar	30	20
Apr	5	10

- Used for "Time Series analysis" - Years over Years, analyze the overall growth or decline of the business's performance over time
- Month-Over-Month, analyze short-term trends & discovers patterns in seasonality

★ `FIRST_VALUE` & `LAST_VALUE`

`FIRST_VALUE` - Access a value from the first row within a window
`LAST_VALUE` - Access a value from the last row within a window

Note: These two window function will use default frame clause i.e.
`RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`
Let's Look at an example

`FIRST_VALUE(Sales) OVER (ORDER BY Month)`

Here, since frame is not mentioned the default frame clause will be used. Due to range of unbounded preceeding the returned result will be 20.

UNBOUNDED PRECEDING → CURRENT ROW

Month	Sales	FIRST_VALUE
Jan	20	20
Feb	10	20
Mar	30	20
Apr	5	20

`LAST_VALUE(Sales) OVER (ORDER BY Month)`

CURRENT ROW, UNBOUNDED PRECEDING

Month	Sales	LAST_VALUE
Jan	20	20
Feb	10	
Mar	30	
Apr	5	

Frame

UNBOUNDED PRECEDING → Current row →

Month	Sales	LAST_VALUE
Jan	20	20
Feb	10	10
Mar	30	
Apr	5	

- It's useless to use LAST_VALUE with default frame clause.
- The only way to effectively use is to define the frame clause as per the query objective
- If you did not notice - the sales column & LAST_VALUE column is same

★ SQL Window use Cases

Top N analysis
Bottom N analysis
Identify & Remove Duplicates
Assign unique ID's & pagination
Data Segmentation
Data Distribution Analysis
Equalizing Load processing
Overall Analysis
Total / Groups Analysis

- Past-to-Whole Analysis
- Time Series Analysis: MOM & YOY
- Time Gaps Analysis: Customer Retention
- Comparison Analysis: Extreme ? Highest / Lowest
- Outlier Detection
- Running total
- Rolling total
- Moving average.

★ Pivoting with CROSSTAB()

What is it?
- Pivoting means turning rows into columns. Imagine taking a table that tracks medal wins by year & transforming it so that each year becomes a column.

Ex: Before Pivoting

Country	Year	Awards
CHN	2008	74
CHN	2012	56
USA	2008	125

After Pivoting

Country	2008	2012
CHN	74	56
USA	125	NULL

Let's Look at the Syntax

```sql
CREATE EXTENSION IF NOT EXISTS tablefunc;

SELECT * FROM CROSSTAB($$
  SELECT Country, Year, COUNT(*) AS Awards
  FROM Summer_Medals
  WHERE Medal = 'Gold'
  GROUP BY Country, Year
  ORDER BY Country, Year
$$)
AS ct (Country VARCHAR, "2008" INT, "2012" INT);
```

Use case: Make a matrix-like report for easier comparison.

CREATE EXTENSION IF NOT EXISTS tablefunc
- This enables the tablefunc extension in postgresql, which provide the CROSSTAB function
- Required to use pivot like operations.

SELECT * FROM CROSSTAB(...)
- This uses the CROSSTAB function to convert row into columns
- It transforms the summer_medals table into matrix with Countries as rows & years (2008, 2012) as columns.

Subquery inside CROSSTAB
- Filters for only gold medal
- Groups country & years, counting how many gold medals each year country won per year
- Ensures results are ordered by Country & years which is required for CROSSTAB.

AS ct (Country VARCHAR, "2008" INT, "2012" INT);
- Defines the output structure:
1) One row/country
2) One column each for gold medals in 2008 & 2012.

🔢 3. `ROLLUP()` — Group Totals and Grand Totals

🧠 What is it?

ROLLUP is like a calculator that gives **totals at different levels**.

Say you're counting medals by country and medal type (Gold, Silver, Bronze). You want:

- Total medals by type
- Total medals by country
- Grand total

🗒️ Sample Output:

Country	Medal	Awards
CHN	Gold	74
CHN	NULL	184 ← subtotal for CHN
NULL	NULL	327 ← grand total

🔧 Query:

```sql
SELECT Country, Medal, COUNT(*) AS Awards
FROM Summer_Medals
WHERE Year = 2008
GROUP BY ROLLUP(Country, Medal);
```

> NULL means "this row is a total, not an actual value."

🔁 4. `CUBE()` — All Possible Totals

🧠 What is it?

CUBE is like **ROLLUP on steroids**—it gives **totals for all combinations**.

🧾 Example:

It will give:

- Totals by country
- Totals by medal
- Grand total
- Everything that ROLLUP gives — plus more

🔧 Query:

```sql
SELECT Country, Medal, COUNT(*) AS Awards
FROM Summer_Medals
WHERE Year = 2008
GROUP BY CUBE(Country, Medal);
```

Use when you want to analyze **from multiple angles**.

🧼 5. `COALESCE()` — Cleaning NULLs

🧠 What is it?

ROLLUP and CUBE generate `NULL`s to represent totals. But NULL looks confusing.

COALESCE(x, 'default') says:

> If x is NULL, show 'default' instead.

🧾 Example:

```sql
SELECT
  COALESCE(Country, 'Total for all countries') AS Country,
  COALESCE(Medal, 'All medals') AS Medal,
  COUNT(*) AS Awards
FROM Summer_Medals
GROUP BY ROLLUP(Country, Medal);
```

Now your output is readable:

Country	Medal	Awards
CHN	Gold	74
CHN	All medals	184
Total for all countries	All medals	327

🧵 6. `STRING_AGG()` — Combine Rows into a String

🧠 What is it?

Instead of listing results in rows, we can **combine them into a sentence**.

📝 Example:

You want to list all countries that got gold medals in gymnastics:

```sql
SELECT STRING_AGG(Country, ', ') AS CountryList
FROM Country_Awards;
```

Output:

CHN, RUS, USA

Great for reports, titles, dashboards!