MySQL

SQL is declarative, meaning wecwrite what we want, and the database optimizer decides how to execute it. While **aliases are assigned in the query's definition**, they are effectively available for referencing during execution because they represent the current row being processed in the main query

A classic SQL query works by evaluating the rows in the database and in most cases applying a filter so that only some of the rows, the desired ones, are returned and/or performing operations on the rows such as combine them, multiplying them or for a scalar

- **FROM Clause**:
  - SQL starts by identifying the tables (or derived tables/subqueries) that will provide the data.
  - If joins are present, they are processed here, combining rows from multiple tables based on the join conditions.
- **WHERE Clause**:

- Filters rows based on the conditions specified.
- This step applies before any grouping or aggregation, ensuring only relevant rows are included in the next steps.

 GROUP BY Clause (if present):

- Groups the filtered rows into subsets based on one or more columns.
- Aggregation functions (e.g., MAX, SUM) are applied to each group in the next step.

 HAVING Clause (if present):

- Filters the grouped subsets based on conditions applied to the aggregated values.
- Only groups that meet the HAVING condition are passed to the next step.

 SELECT Clause:

- Retrieves the specified columns, calculated values, or expressions.
- Aliases assigned in the SELECT clause are applied here but are not available earlier in the query.

 ORDER BY Clause (if present):

Sorts the result set based on the specified columns or expressions.

Column aliases from the SELECT clause can be used here.

 LIMIT Clause (if present):

- Restricts the number of rows returned in the final result.

The tables and their relationships are exectued immediately while the select and the columns are used at the end of the query, after the filtering has taken place

GENERAL COMMANDS

1. Basic Commands (select, from, distinct)

```
SHOW DATABASES;

USE tenants;

DESCRIBE ten_details;

SELECT * FROM ten_details;

SELECT name FROM ten_details;

SELECT Rent, Room FROM ten_details;

SELECT DISTINCT * FROM ten_details

SELECT DISTINCT(name) FROM ten_details;
```

2. Logical Operators (and , or, xor)

LOGICAL OPERATORS are the most used operators to filter and combine conditions in sequel. They are commonly used in the order where we find the WHERE clause to start the filtering process in the query, then a condition often represented by one or more comparison operator combined through the AND, OR and XOR clauses. While the functionality of the WHERE is exhaustively explained by the short above sentence, it can be helpful to spend a few words about the other logical operators. These operators are used to filter the database so that:

- AND: the query returns a result only when all the conditions combined through the various AND are met. In other words, if even one of the conditions evaluates to False, the result will not include that row.
- OR: the query returns a result when at least one of the conditions out of the ones combined through the OR is met. Even if other conditions are false, as long as one condition is true, the row will be included in the result
- XOR: the query returns a result only when one condition is evaluates to True and the other one evaluates to False.

Given the nature of these operators, the best way to introduce them is by using them in combination with comparison operators, hence the examples of their syntax and application is going to be included in the next paragraph.

3. Comparison Operators (<, >, =, !=, >=, <=, between, in)

COMPARISON OPERATORS are also among the most common and useful operators in the SQL environment, allowing users to perform comparisons between values in the columns of a database. They are intimately connected to logical operators.
An important aspect of these operators is that the input for the comparison must always be a specific value or a set of specific values. It is not possible to compare entire columns directly in a single comparison. Even when using operators like BETWEEN or IN to work with ranges or lists of values, the comparison must still involve a defined value or set of values for it to evaluate correctly. The only exception to this is using the ALL keyword, which allows us to compare a column to a set of values returned by a sub-query (the ALL operator is explained in detail in the sub-query section). The 'not equal to' operator (!=) can also be written as <>

4. Filtering with WHERE

WHERE clause filters **rows before** grouping or aggregation.
It is used to specify conditions **that must be true for a row to be included in the result. It's always used with some sort of comparison/filtering operator (= , is null, between) and often in combination with logical operators to apply multiple filters at once. The opposite/negation of a filter can be done with WHERE NOT CONDITION or with the 'not equal to' operator**

```
SELECT * FROM ten_details
WHERE Sex = 'M';

SELECT name, rent
FROM ten_details
WHERE rent = 350
OR rent = 400;
```

WHERE is a condition either on the columns stated in the select or in other columns. They work on set of rows (set of values) and not with only one row (i.e. AGG).

```
SELECT name, rent
FROM ten_details
WHERE name = 'Santiago'
AND Rent = 350 OR rent = 400;

SELECT name, sex
FROM ten_details
WHERE rent > 400
AND start_date >= 2023-6-19 AND Room = 'Sx';

SELECT *
FROM ten_details
WHERE rent > 400 AND start_date >= 2023-7-19 AND start_date <= 2024-08-30;

SELECT * FROM ten_details
WHERE rent!=415 AND NOT sex = 'M';


SELECT * FROM ten_details
WHERE rent BETWEEN 395 AND 411;

SELECT * FROM ten_details
WHERE Date(start_date) NOT BETWEEN 2016-10-01 AND 2020-11-30;

SELECT name, rent FROM ten_details
WHERE name = 'Davide' AND Rent IN (360, 400);
```

5. NULL

Whenever there's a missing value, SQL assigns to said value the special marker NULL. This special marker is not implemented as a value, therefore using mathematical operators, such as '*', '/' or '=', on it will give back an unknown result. If in a query we have to retrieve NULL elements, the right syntax is performed by using the 'IS' statement instead of the '=' (which would give back zero rows since the result of this query is unknown).

```
SELECT * FROM ten_details WHERE room IS NULL;

        10  / NULL;  → -- result is NULL
```

6. ORDER BY

ORDER BY simply orders the results according to the criteria/column expressed in the clause: if there's more than one column in the ORDER BY statement, then the first it will order by the first column and then, within this column, by order of the other column.
If we don't explicitly state it, the default order will be ascendant (or we could also call it through the ASC keyword). Otherwise, we can opt for a descendent order through the DESC keyword.

```
select name, sex, Rent from ten_details ORDER BY Rent;

select name, Rent from ten_details ORDER BY  Rent,  name DESC LIMIT 15;

select name from ten_details where marks > 75 ORDER BY RIGHT(name,3), Rent;
```

7. LIKE (like, not like, I like)

LIKE is used when we are not exactly sure about the total characters of a value we want to filter the query by: so we can use this command to only give a partial number of characters instead of the full version

```
select * from ten_details WHERE name LIKE 'T%' or name LIKE 'Giuli_';

select * from ten_details WHERE name NOT LIKE 'Giul__' or name LIKE '%fran%';

select * from ten_details WHERE name ILIKE 'T%';
```

$\longrightarrow$ MySQL does not accept ILIKE

8. Aggregated function agg (sum, count, max, avg)

Aggregate functions (AGG) perform calculations on a set of values and return a single result (e.g., SUM, COUNT, AVG, MAX, MIN). For this reason, AGG's cannot be used in a WHERE clause because WHERE filters individual rows, while AGG functions operate on groups of rows (HAVING, as we'll see in the GROUP BY section, is used instead, as it specifically filters aggregated results)
NOTE: Round is used to choose the digits of decimals.

```
SELECT COUNT(rent) FROM ten_details;

SELECT COUNT(DISTINCT rent) FROM ten_details;

SELECT SUM(Rent) FROM ten_details;

SELECT ROUND(AVG(Rent),3) FROM ten_details;

SELECT ROUND(AVG(rent::NUMERIC, 3) FROM ten_details;

SELECT MAX(rent),MIN(rent) FROM ten_details;

SELECT ROUND(max(rent),2) FROM ten_details;
```

COUNT(*), by definition, counts all the rows, including those with null values. This is a useful aspect when searching for missing values (null) in other columns: the difference in the COUNT(*) and COUNT(another_column) is the number of missing values from the another_column

Some SQL environments expect elements in the **ROUND** as actual numbers instead of FLOATS or DOUBLE PRECISION. This is the case also when using AVG, which might return the previous data types. To work around this issue, we can simply convert the value to an actual number by using the '::numeric' expression inside the ROUND itself

9. GROUP BY, HAVING

GROUP BY groups values "per" category: for example, it will group the sum of the rents "per" each sex. In the GROUP BY section, the same columns used in the SELECT must be included and in the same order, except for aggregate functions. Also, either there must be an aggregate function (AGG) in the SELECT, or all the selected columns must be listed in the GROUP BY.

The added benefit of using GROUP BY is that we can further filter the query through a **HAVING** clause, which works similarly to WHERE but on grouped data. This clause allows us to apply aggregate functions outside of the SELECT (or within a sub-query) and filters the grouped values accordingly. As mentioned, it acts as an additional filtering step applied after grouping.

If a GROUP BY is applied to multiple columns, SQL will generate unique groups for every distinct combination of values from those columns. This means that each unique value from one column will be grouped together with each unique value from the other specified columns, forming all possible distinct combinations present in the dataset. So if we group by room (1 or 2), sex (M or F), and rent (100 or 200), SQL will return combinations such as:

Group by room, sex,rent	room	sex	rent
First combination (room 1, sex M)	1	M	200
	1	M	100
Second combination (room 1, sex F)	1	F	200
	1	F	100
Third combination (room 2, sex M)	2	M	200
	2	M	100
Fourth combination (room 2, sex F)	2	F	200
	2	F	100

NOTE: from the previous table we can see tha the order of columns in GROUP BY affects both how they are displayed and how SQL groups the data. SQL first groups by the first column, then by the second within each of the first groups, and so on. This also determines the order in which values change in the output, as grouping follows a hierarchical, left-to-right sequence.

At this point, we can count the occurrences of each combination by using COUNT(*) in the SELECT, and the output will display this count. This is a useful approach for assessing duplicates in a given number of columns because if there is more than one row with the same room, sex, and rent values, those entries must be duplicates.

We can then use HAVING COUNT(*) > 1 to confirm whether multiple entries exist with the same values across all specified columns, meaning they are duplicates.

NOTE: If we are checking for duplicates and want to eliminate them to clean the table, we can create a new table that excludes those duplicate entries.

The WHERE clause operates before GROUP BY and only filters raw data, meaning it does not affect aggregate functions.

ORDER BY and HAVING take action after the GROUP BY, with HAVING specifically applying conditions to aggregated results.

```
SELECT sex, sum(rent) FROM ten_details GROUP BY sex;
SELECT sex, room FROM ten_details GROUP BY sex, room;

SELECT sex, name, count(name) FROM ten_details
GROUP BY sex, name ORDER BY count(name), sex LIMIT 5;

SELECT room, avg(rent) FROM ten_details WHERE rent
BETWEEN 390 AND 425 GROUP BY room ORDER BY avg(rent);

SELECT sex, sum(rent) FROM ten_details GROUP BY sex
HAVING SUM(rent) IN (400,425,460);
```

10.Alias ('AS')

The **AS** clause assigns an alias (another name) to the last column in the SELECT section. The alias of the AS is assigned at the end of the query's execution, such as after the HAVING has been processed. Keep this in mind because if the alias is used elsewhere in the query before it is assigned, it won't be recognized.
It is also possible to assign an alias to a table, which must be done in the FROM section and it can be used in combination with an alias for columns. If the alias is assigned to a table, it will immediately be available for the entire scope of the query (and in the sub-query as well). This difference between how aliases are applied to columns and to tables is due to the order in which SQL processes the different sections of a query.
In most SQL dialect, the AS keyword is optional: however, it's considered good practice to include it to enhance clarity and consistency.

```
SELECT name, rent AS rent_per_room FROM ten_details;

SELECT name AS person, sex, SUM(rent) AS sum_rent FROM ten_details;

SELECT name person , sex,  SUM(rent)  sum_rent FROM ten_details
GROUP BY rent HAVING sum(rent)

SELECT A.name AS person,  A.sex, B.mq FROM ten_details AS A JOIN dati_catastali AS B
ON  A.indirizzo=B.indirizzo ORDER BY rent

SELECT A.name,  A.sex, B.mq FROM ten_details  A  JOIN dati_catastali  B
ON  A.indirizzo=B.indirizzo ORDER BY rent
```

11.Dates

Many of the dates command are pretty straight forward and don't need further explanation

```
SELECT EXTRACT(month from start_date)  FROM ten_details;

SELECT EXTRACT (year from end_date) FROM ten_details;

SELECT AGE (start_date) AS age_now FROM ten_details;

SELECT TIMESTAMPDIFF(YEAR, end_date, CURDATE())
FROM ten_details;

SELECT TO_CHAR(end_date, 'dd-MM-YY') FROM ten_details;
```

$\Longrightarrow$

AGE and **TO_CHAR** are not supported on MySQL.
AGE=TIMESTAMPDIFF;
TO_CHAR=DATE_FORMAT;
TO_CHAR is also used to convert something into a string, on MySQL the command is **CAST**.
TO_CHAR can be used in combination with a DISTINCT to extract the month, or years, or day, etc.
DOW works to calculate the specific day of the week from a date (Sunday=0, Saturday=6).
On MySQL **EXTRACT** works differently and one can use **DAYOFWEEK** (Sunday=1)

```
SELECT DATE_FORMAT(end_date, '%d-%M-%Y') FROM ten_details;
```

```
SELECT count(start_date) FROM ten_details
WHERE EXTRACT(dow FROM start_date) = 1;
```

```
SELECT count(start_date) FROM ten_details
WHERE dayofweek(start_date) = 1;
```

12.Mathematical Functions (+, - , *, /, %)

The SQL environment includes a wide variety of **MATHEMATICAL FUNCTIONS** that help us process operation on numbers-data columns. A detailed list of these functions, along with their functionality, syntax, and examples, can be found on the help page.
Here the case of the difference ' %' of two types of rents and the 5% of a rent column

NOTE: it's worth mentioning the function to find the regression slope (m as in y=mx+b) of a linear regression where the syntax is regr_slope(y_dependent_variable,x_independent_variable) and defines positive trend if positive or negative trends if negative

```
SELECT ROUND(rent / prezzo_max, 2) * 100 FROM canone_concordato;

SELECT 0.5 * rent FROM canone_concordato;

SELECT 5 * rent / 100 FROM canone_concordato;

SELECT rent FROM canone_concordato WHERE rent % 2 = 0;

SELECT REGR_SLOPE(start_date , rent) from ten_details;
```

NOTE: an important **mathematical function** is the remainder (%)!
By setting the reminder of 2 to zero we can retrieve even numbers from

13.String Functions (LENGTH(), ||, CONCAT(), UPPER(), LOWER())

Another useful feature in SQL is represented by the **STRING FUNCTIONS**. These functions allow us to perform many different operations on text-data columns such as edit, combine, alter, analyze (e.g. count the letters) and many more. A detailed list of these functions, along with their functionality, syntax, and examples, can be found on the help page.

```
SELECT LENGHT(name) FROM ten_details;

SELECT name || hometown FROM ten_details;

SELECT name|| ' ' || hometown FROM ten_details;

Select CONCAT(name, ' ' , hometown) from ten_details;

SELECT UPPER(name || '—' || hometown) FROM ten_details;

SELECT LOWER(LEFT(name,1))||LOWER (hometown)|| 'gmail.com' FROM ten_details;

SELECT CONCAT(LOWER(name), ' ', UPPER(hometown), '$') FROM ten_details;
```

NOTE: on MySQL the equivalent of || is **CONCAT**. The ' ' and '—' operators add a space or a dash between strings

14.UNION, UNION ALL

UNION is used to combine the results of two or more **`SELECT`** statements. It removes duplicate rows, meaning that if both **`SELECT`** statements return the same row, it will appear only once in the final result.
UNION ALL also combines the results of two or more **`SELECT`** statements, but keeps all duplicates. If two identical rows appear, both will be included in the final output.

```
SELECT * FROM ten_details
UNION
SELECT name FROM OG
```

```
SELECT * FROM ten_details
UNION ALL
```

15.Creating a copy of a table with SELECT INTO

Whenever we want to create a quick copy of a table based on the result of a query without actually creating a new table (as we'll see in the creating a table section) we can use SELECT INTO.
This method is often used for temporary analysis or to store filtered query results in a new table. In fact, INSERT INTO won't copy the constraint of the original table though, so it's not suited for operations across related tables

```
SELECT * INTO #temporary FROM ten_details
```

If we wanted to insert values/rows/columns in an existing table, we should use INSERT INTO as we'll see in the related section

JOIN COMMAND

1. INNER JOIN/ JOIN

A **JOIN** (typically an **INNER JOIN)** is used to combine rows from two or more tables based on a common column, usually a primary key. This allows you to retrieve related data across different tables in a single query. The JOIN works by matching values from a column in one table with values in a column from another. When the JOIN involves two different tables, the connection is typically made through primary key–foreign key relationships.

The syntax for specifying the relationship is: **ON** table1.column = table2.column

This tells SQL how the rows in the two tables are related. If the two tables have columns with the same name, you must use the table prefix in the SELECT clause to clarify which column you're referring to.

When we use a JOIN, we can retrieve any or all columns from both tables by specifying them in the SELECT clause.

```
SELECT name,
        sex,
        mq
FROM dati_catastali
JOIN ten_details
ON dati_catastali.indirizzo=ten_details.indirizzo
ORDER BY rent
```

2. SELF JOIN

A **SELF-JOIN** happens when joining a table with itself. In this case, any chosen column can be used as long as the logic makes sense. This is useful for comparing values within the same table, such as employees and their managers

```
SELECT E2.FirstName, E2.LastName
FROM Employees E1
JOIN Employees E2
ON E1.DepartmentID = E2.DepartmentID
WHERE E1.FirstName = 'Jane' AND E1.LastName = 'Doe'
AND E2.EmployeeID <> E1.EmployeeID;
```

3. LEFT JOIN

A **LEFT JOIN** returns all the rows from the left (first) table, along with the matched rows from the right table. If there's no match, the columns from the right table will contain NULL

```
SELECT DISTINCT customername,orderid
FROM customers c
LEFT JOIN orders o
ON o.customerid = c.customerid
```

4. RIGHT JOIN

A **RIGHT JOIN** is the opposite of a LEFT JOIN. It returns all rows from the right (second) table, and any matched rows from the left table. If there is no match, the left-side columns are filled with NULL

```
SELECT DISTINCT customername, orderid
FROM orders o
RIGHT JOIN customers c
ON o.customerid = c.customerid;
```

5. FULL OUTER JOIN

A **FULL OUTER JOIN** returns all rows from both tables, regardless of whether a match exists. If a row from one table doesn't have a match in the other, the unmatched side will contain NULL.
NOTE: not all SQL environments (e.g., MySQL) support FULL OUTER JOIN natively without workarounds

```
SELECT DISTINCT customername, orderid
FROM customers c
FULL OUTER JOIN orders o
ON o.customerid = c.customerid;
```

6. Multiple JOIN

Multiple joins let you combine data from **more than two tables** by chaining multiple `JOIN` clauses in a single query.
Each `JOIN` connects two tables using a common key, and you can **mix different types** (`INNER`, `LEFT`, `RIGHT`, etc.).

```
SELECT name,
        sex,
       og,
       mq
FROM OG
JOIN OG
ON OG.name=ten_details.name
JOIN dati_catastali
ON dati_catastali.casa= ten_details.casa
GROUP BY sex;
```

```
SELECT canone_concordato.indirizzo,
       canone_concordato.Room,
       ten_details.*
FROM ten_details
JOIN canone_concordato
ON canone_concordato.Room=ten_details.Room
JOIN dati_catastali
ON canone_concordato.casa= dati_catastali.casa
WHERE canone_concordato.mq_condivisi = 8.5;
```

SUB-QUERY

A **SUB-QUERY** , as the name suggests, is a query nested inside another query. It's implemented within parentheses (()) at some point in the main query, using the standard syntax of a normal query.
The primary purpose of a subquery is to retrieve certain data from a column or set of columns by applying the necessary conditions. The main query can now use the new revised column(s) and its adjusted data as a foundation to perform further filtering, comparisons and calculations. To achieve this, the sub-query is executed first, providing the main query with the desired dataset to work with.

This technique offers additional flexibility, such as:

- Enabling the use of aggregate functions (AGG) in the WHERE clause for filtering.
- Supporting joins to retrieve columns from other tables using the IN condition, as long as the specified column exists in both tables.

While the above provides a more general definition, it is helpful to outline the various positions where subqueries can be implemented in a query and how the position determines the type of data returned and its purpose. Below are the main places where subqueries can be used:

1) In the SELECT Clause

- **Purpose**: Calculates or retrieves a single value for each row or for all rows in the main query.
- **Outcome**:
  - **Correlated Subquery**: Returns a value that depends on the current row being processed in the main query.
  - **Non-Correlated Subquery**: Returns a single constant value that applies to all rows.

2) In the FROM Clause

- **Purpose**: Acts as a derived (temporary) table that the main query uses as a data source.
- **Outcome**: Returns multiple rows and columns, treated as a table by the main query.

3) In the WHERE Clause

- **Purpose**: Filters data in the main query based on the subquery's result.
- **Outcome**:
  - **Scalar Value**: If used with a comparison operator (e.g., =, <).
  - **List of Values**: If used with IN or NOT IN.

4) In a JOIN Clause

- **Purpose**: Acts as a derived table (similar to FROM) to enrich data by joining it with other tables. In this case, an alias must be assigned to the derived table (just like any other table, it must have a name so that SQL can refer to it)

Outcome: Returns multiple rows and columns, treated as a table.

While the above provides a more general definition, it is helpful to outline how the structure and position of a subquery determine the type of data it returns. The following points describe some of the possible outcomes in terms of data:
1) A scalar (single data): when executed in WHERE clause and in combination with a COMPARISON operator, because these operators compare only one value at a time, the sub-query filters the original column returning a single value. The main query will then use this value to compare it with the object in the WHERE condition.
2) A list of value (single column): when executed in a WHERE clause and in combination with an "IN" or "NOT IN" operator, because of the nature of these operators, the sub-query filters the original column returning the required list of values from that column. The main query will then use these values as a foundational dataset for its further operations.

3) A table (multiple rows/columns): when executed directly in a FROM clause as a derived (temporary) table, the sub-query will return a filtered table (based on the conditions applied). The main query will then use this table as a foundational dataset for its further operations. NOTE: a sub-query can be executed directly in the SELECT statement. In this case, it will return only one value which will be applied to all rows in one of two ways
1) Correlated sub-query: if the subquery is correlated to the main query (i.e., there is a dependency between the data in the subquery and the current row being processed in the main query), it will return a value that can vary for each row. The subquery is re-evaluated for every row in the main query.
2) Non correlated sub-query: if the subquery is not correlated to the main query (i.e., it does not depend on the value of the current row in the main query), it will return a single value that is constant across all rows. The subquery is executed once, and the result is applied uniformly to all rows in the main query. An important addition to the previous discussion is that the sub-query can include more than one column in the SELECT only under certain conditions:
1) When the sub-query starts in the FROM statement. In this case, the main query treats the columns from the sub-query as a derived table with the selected attributes.
2) When multiple columns are used in comparison. In certain SQL database, if the condition before the main query involves multiple attributes, then the SELECT in the sub-query can also include multiple attributes. This will return the columns as pairs for comparison.
NOTE: a situation in which this is not possible is when the comparison between the sub-query and the main query is performed through an 'IN' or 'NOT IN' condition, as per the nature of these operators

1) Correlated sub-query: if the subquery is correlated to the main query (i.e., there is a dependency between the data in the subquery and the current row being processed in the main query), it will return a value that can vary for each row. The subquery is re-evaluated for every row in the main query.
2) Non correlated sub-query: if the subquery is not correlated to the main query (i.e., it does not depend on the value of the current row in the main query), it will return a single value that is constant across all rows. The subquery is executed once, and the result is applied uniformly to all rows in the main query.

An important addition to the previous discussion is that the sub-query can include more than one column in the SELECT only under certain conditions:

1) When the sub-query starts in the FROM statement. In this case, the main query treats the columns from the sub-query as a derived table with the selected attributes.
2) When multiple columns are used in comparison. In certain SQL database, if the condition before the main query involves multiple attributes, then the SELECT in the sub-query can also include multiple attributes. This will return the columns as pairs for comparison.

NOTE: a situation in which this is not possible is when the comparison between the sub-query and the main query is performed through an 'IN' or 'NOT IN' condition, as per the nature of these operators

NOTE: when using a join with the other table being expressed as a sub-query, there must always be an alias for the sub-query.

NOTE: a **SUBQUERY** and a **DOUBLE JOIN** are often interchangeable when dealing with 3 tables. This is particularly true when the relationships between the tables are straightforward, allowing us to directly use **JOINS** to relate them to each other. In such cases, either approach can generally be used.
However, there are few situations in which this interchangeability becomes less true so it's useful to summarize which approach is better suited according to different situations.
SUBQUERIES are better suited when:

- we need to **filter data using an aggregation** (AGG) and compare it in a WHERE or HAVING statement with the outer query or before a JOIN
- dealing with **tables that contain one-to-many relationship** (one row in one column relates to multiple rows in another column, e.g. one student can follow multiple classes). A JOIN could in this case lead to row duplication

- **filtering a subset of data** before the JOIN could reduce the numbers of rows processed in the main query

JOINS are better suited when:

- the **performance** of a query is important. This is the case for very large dataset since in modern SQL environments **JOINS** are better optimized than subqueries
- **readability** is our focus, since they provide a more intuitive structure when working with multiple tables or the code is going to be shared with other coders
- we want to avoid correlated SUBQUERIES, which can be significantly slower because of the dependency with the outer query

```
SELECT name, og FROM (SELECT name, og, SUM(rent) FROM OG WHERE og='YES')
AS name_og;

SELECT name, sex FROM ten_details
WHERE rent  >  ( SELECT AVG(rent) FROM ten_details);

SELECT name, sex FROM ten_details
WHERE name  IN (SELECT name FROM OG WHERE og = 'YES');

SELECT name, sex FROM ten_details
WHERE name IN (SELECT mq FROM dati_catastali JOIN ten_details
ON dati_catastali.indirizzo=ten_details.indirizzo);

SELECT OG.name, og, OG.hometown FROM OG WHERE name IN
(SELECT ten_details.name FROM ten_details
JOIN canone_concordato ON
canone_concordato.Room=ten_details.Room
where canone_concordato.mq_condivisi = 80) and og='YES'
ORDER BY name;

SELECT name, sex  FROM ten_details AS c WHERE EXISTS
(SELECT * from OG AS p WHERE c.name=p.name AND og = 'Yes');

SELECT name, og  FROM OG AS c WHERE EXISTS
(SELECT * FROM ten_details AS p WHERE c.name=p.name
AND name LIKE 'T%');
```

The **EXISTS** keyword is used to check whether a sub-query "makes sense". It evaluates to TRUE if there's at least one row that validates the query. Then, after finding the first one, it stops executing further. While it might seem redundant in some cases (as one could run the subquery directly), it's particularly useful where we only need to verify the presence of data rather than retrieve it

WINDOW FUNCTIONS

WINDOW FUNCTIONS are a powerful feature in SQL that allow us to perform **calculations across a set of rows related to the current row**—without grouping or collapsing the result set. They are sometimes referred to as **analytic functions**.
Unlike aggregate functions (such as AVG, SUM, COUNT) which return a single result per group, **window functions return a value for each row** in the original table, based on a defined **"window" of rows**.

As for the syntax, the generic structure is as follows:

```
function_name(expression) OVER (
   PARTITION BY column_to_group
   ORDER BY column_to_order)
```

```
select country,
        averagetemp,
        (averagetemp - AVG(averagetemp) OVER (PARTITION BY country)) /
        STDDEV(averagetemp) OVER (PARTITION BY country) AS z_score
from global_1850;
```

Key Elements
- function_name(...) is the operation being applied, like AVG(), ROW_NUMBER(), RANK(), etc.
- The OVER clause defines the **window**, which tells SQL how to group and sort the data for the function.
- PARTITION BY divides the data into **logical groups** (like GROUP BY, but without collapsing rows).
- ORDER BY determines the **order of rows** within each partition (required for ranking or time-based functions).

Common Use Cases
- Calculating **running totals** or **moving averages.**
- Assigning **row numbers** or **ranks** within each group.
- Comparing values across rows using **LAG** and **LEAD.**
- Standardizing or normalizing data with **Z-scores.**

NOTE:
- **Window functions never reduce the number of rows** — they return a new value **for each row**.
- They can be used **with or without GROUP BY**, often **together with CTEs** or subqueries.
- If you omit the PARTITION BY clause, the function is applied over the **entire result set**.

FUNCTION	FUNCTION DESCRIPTION
ROW_NUMBER()	Assigns a unique number to each row
RANK()	Assigns ranks with possible gaps
DENSE_RANK()	Like RANK() but without gaps
LAG()	Returns a value from the **previous row**
LEAD()	Returns a value from the **next row**
NTILE()	Divides rows into n equal buckets
FIRST_VALUE()	Applies the smallest value of a column
LAST_VALUES	Applies the biggest value of a column

1. ROW_NUMBER

In case of a tie (same marks), ROW_NUMBER() will assign a unique row number to each tied row, but the order between them is arbitrary unless a secondary sorting column is specified.

```
SELECT *, ROW_NUMBER() OVER(ORDER BY marks DESC) AS row_num
FROM students
```

PARTITION BY is used when we want to **apply the window function to every group of values** (in this case the subject of the classes). The idea is similar to GROUP BY with the difference that group by **collapses rows** into one row per group

```
SELECT *, ROW_NUMBER() OVER(PARTITION BY subject ORDER BY marks DESC) AS row_num
FROM students
```

2. RANK()

Missing ranks occur because the RANK() function assigns the same rank to rows with equal values, and then skips as many subsequent ranks as there are tied rows.

```
SELECT *, RANK() OVER(ORDER BY marks DESC) AS row_num
FROM students
```

3. DENSE_RANK()

Missing ranks occur because the RANK() function assigns the same rank to rows with equal values, and then skips as many subsequent ranks as there are tied rows.

4. LEAD()

LEAD(profit) creates a new column where each row shows the profit of the *next* row in sequence. The last row returns NULL by default. Works as a window function, optionally within PARTITION BY groups.

```
SELECT *, LEAD(profit) OVER(PARTITION BY product ORDER BY monthnumber DESC)
AS nxt_month_profit FROM profitdata
```

5. LAG()

LAG(profit) is complementary to lead as it creates a new column where each row shows the profit of the previous month with the first row being NULL

```
SELECT *, LAG(profit) OVER(PARTITION BY product ORDER BY monthnumber DESC)
AS prv_month_profit FROM profitdata
```

6. FIRST_VALUE()

FIRST_VALUE() creates a new column and applies to all rows the minimum value of the column specified inside the brackets. if we wanted to create two columns, we just need to apply it twice

```
SELECT *,
FIRST_VALUE(salary) OVER(ORDER BY salary) AS minimum_salary
FROM employeesalaries
```

```
SELECT *,
FIRST_VALUE(employeename) OVER(ORDER BY salary) AS minimum_salary,
FIRST_VALUE(salary) OVER(ORDER BY salary) AS minimum_salary
FROM employeesalaries
```

7. LAST_VALUE()

Same as FIRST_VALUE() but with the maximum values

CREATING/ALTERATING A TABLE

In this section we are going to describe a few main commands involved in the creation and management of a database.
For every concept related to this topic, we'll present a couple of coding examples which will practically show how to build a database, how to perform the changes to the tables and columns, etc.

1. CREATE TABLE, ALTER, DROP

- **TABLE CREATION:** the first step is to use the CREATE TABLE command, so that SQL knows what we are going to do, followed by the name we chose for the table.
  Then, inside parentheses, we must enter the name of the column, the data type (INT, VARCHAR, etc.) and the constraint (UNIQUE, NOT NULL) in this exact order.

```
CREATE TABLE ten_deatils (
tenant_id SERIAL PRIMARY KEY,
ten_name VARCHAR(50) UNIQUE NOT NULL,
sex VARCHAR(1) NOT NULL,
hometown VARCHAR(50) NOT NULL,
start_date TIMESTAMP NOT NULL,
end_date TIMESTAMP,
room VARCHAR(50) NOT NULL,
 rent FLOAT,
zona VARCHAR(50) NOT NULL,
address VARCHAR(100) NOT NULL,
area VARCHAR(100) NOT NULL
people INT);
```

When we implement the NOT NULL constraint, we can avoid stating that column in the INSERT but if we do write that column in the INSERT, then in the list of value we must explicitly write null for the blank values. The NOT NULL constraint is the only one we can directly drop with the ALTER-DROP command while in order to drop any of the other constraint types we must know their name (name that is assigned automatically by SQL and can be retrieved with a code state later on

- **DEFINITION OF PRIMARY KEY:** if the column is a PRIMARY KEY (see note), we must explicitly address this property through the related command. Moreover, if we want the PRIMARY KEY to be a sequence of numbers, we can conveniently use the SERIAL command, which generates said sequence where the numbers are automatically incremented at every row (and this naturally implies the constraints of unicity and non-nullity).
  Alternatively, if we want the elements from the PRIMARY KEY to be something other than numbers (though the requirement remain for the elements to be UNIQUE and NOT-NULL), we can manually express the desired data type.

- **ALTERATION OF A TABLE:** by using the ALTER TABLE/COLUMN command (often in combination with other keywords) we can modified a pre-created table. This is a very powerful tool that helps us fix errors made during the implementation of the structure and/or when facing any sort of request for adjustments so to achieve the complete description of the dataset in later stages of the data collection, without having to recreate the table from scratch: we can change name (for both table and columns, RENAME TO), data type (TYPE-USING), constraints, etc. or add/delete a column (DROP COLUMN).

```
ALTER TABLE ten_deatils RENAME TO ten_details;

ALTER TABLE ten_details
RENAME COLUMN tennanant_id TO tenant_id;

ALTER TABLE ten_details DROP COLUMN zona;
```

NOTE: In SQL, a PRIMARY KEY or pk is a constraint applied to a column (or a set of columns) in a table that uniquely identifies each row in that table. It ensures that no two rows have the same value for the primary key column(s), and it also prevents null values in the primary key column(s).
Even though there's only one PRIMARY KEY per table, this can include multiple columns ("composite primary key") with the request of their combination being unique, meaning that they don't have to be necessarily unique on their own.
Here are some key characteristics of a primary key:

a. **Uniqueness**: Each value in the primary key column(s) must be unique. This guarantees that every row in the table can be uniquely identified.
b. **Not NULL**: A primary key column cannot contain NULL values because it must uniquely identify each record in the table.
c. **Immutability**: Once set, the value of a primary key should not change. Changing a primary key can disrupt relationships between tables in a database.

A SECONDARY KEY or sk is a column with (usually but niot necessarily) unique values used for faster searching or indexing data in a more efficient way (like a username or email which are often unique to one customer). Values belonging to this constraint can be NULL and are often non auto-incremented (username, email). Unlike the previous one, there can be multiple SECONDARY KEY in one table

Example 1: creation of a table, alteration of a misspell in the name of both table and columns(ALTER TABLE/COLUMN-RENAME) and deletion of a column (DROP COLUMN)

2. ALTER TABLE – ADD/DROP CONSTRAINT/COLUMN, - FOREIGN KEY

Since we've already explored the concept of ALTER TABLE, let's jump straight into the remaining topics for this section. Let’s start by creating the table

```
CREATE TABLE OG (
og_id INTEGER PRIMARY KEY,
name VARCHAR(50) UNIQUE NOT NULL,
sex CHAR(1) NOT NULL ,
hometown VARCHAR(20) NOT NULL,
room VARCHAR(15) NOT NULL,
og CAHR(3) NOT NULL
);
```

- **ADDING A COSTRAINT**: the ADD CONSTRAINT/COLUMN command, just like the name suggests, helps us adding a constraint or a column to a table that has already been implemented. In the case of a column, we can add the data type or the constraint altogether.

```
ALTER TABLE OG
ADD CONSTRAINT pk_og_id PRIMARY KEY (og_id);
```

```
ALTER TABLE OG DROP ten_name;
```

```
ALTER NAME OG ADD ten_name VARCHAR(50) PRIMARY KEY;
```

- **ADDING A FOREIGN KEY**: as an additional functionality to the previous point, ADD CONSTRAINT – FOREIGN KEY – REFERENCES allows us to apply this specific constraint to an existing column in table which consist of referencing (REFERENCES = creating a connection in terms of) a column in another table. (see note for an explanation about foreign keys)

```
ALTER TABLE OG
ADD CONSTRAINT fk_ten_name
FOREIGN KEY (ten_name) REFERENCES ten_details (ten_name);


ALTER TABLE OG
ADD CONSTRAINT fk_name
FOREIGN KEY (name) REFERENCES ten_details (ten_name);
```

These codes serve to add a foreign key in one table from another table and to add the constraint of primary key

- **DELETING A CONSTRAINT**: the DROP CONSTRAINT/COLUMN is used whenever we want to drop a constraint from an existing table

```
ALTER TABLE Thursday DROP CONSTRAINT pk_hour;

ALTER TABLE ten_details ALTER COLUMN ten_name DROP NOT NULL;
```

NOTE: in many SQL environment the keyword COLUMN after ADD/DROP is not mandatory. This is the case for MySQL, PostgreSQL, SQL Server.

NOTE: a FOREIGN KEY or fk is a constraint applied to a column (or a set of columns) one table that references a PRIMARY KEY from another table.

a. It ensures data integrity, preventing invalid data from being inserted and orphan records (records that reference a non-existent value in the parent table): this avoids accidentally inserting in the fk a value that it's not present in the pk.

b. It maintains referential integrity**,** ensuring that relationships between tables stay valid so that child tables always reference valid parent records: this avoids deletion of a value present in a pk if that value is also present in the fk, unless cascade deletion is enforced.

c. It allows linking data/columns across tables by creating a relationship between tables and making it easy to retrieve related data. This is essential for JOIN operations: a fk links columns in different table, allowing to JOIN those tables.

Example 2: creation of a second table with a reference to the previous one via a FOREIGN KEY, correction of the mistakes in the name of a column, few attempts to create the REFERENCE to the first table and deletion of generic constraints and NOT NULL constraint (DROP CONSTRAINT/NOT NULL)

```
SELECT constraint_name FROM information_schema.table_constraints
WHERE table_name = 'Thursday'
AND constraint_type= 'primary key';
```

These are the codes to find the type and the name of a constraint of a column and to drop the primary key

3. DROP TABLE-CASCADE

The DROP TABLE – CASCADE is used when a column in a table ("child table") references a column in another table ("parent table") through a FOREIGN KEY constraint. In such cases it's not possible to simply delete the parent table: SQL would raise an error because of the relationship between the two tables.
To delete the parent table along with all its dependencies, we use CASCADE, which removes the parent table and all related CONTRAINTS, FOREIGN KEYS and dependent objects.

Example 3: deleting one table that presents a CASCADE effect on a second one

```
DROP TABLE ten_details CASCADE;
```

4. CREATION OF A THIRD TABLE, CONVERTING A DATA TYPE (USING)

In this section we are just going to retain ideas by creating a third table and performing a few actions on it. The only new information here is about how to convert one data type to a different one in a column through the USING command: if we wanted to convert data types, we don't have to explicitly express the old data type in the line of code but instead we must always refer to the new one (both after the TYPE keyword and after the USING column_name::)

NOTE: USING keyword is only necessary if the new data type and the old one are not part of the same family (string ⇒ numeric, etc). So, if the old data type is FLOAT and the new one is INTEGER or (VARCHAR ⇒ TEXT), we don't have to use USING, but if the old data type is VARCHAR and the new one is TIMESTAMP (string ⇒ data time) we must use it. In order to avoid issues when changing the datatype, it is recommendable to always add the USING keyword.

<u>Example 4</u>: creation of a third table with an alteration of the COSTRAINT of one column, of the name of a column (two consecutive times) and of the type of the data

```
CREATE TABLE canone_concordato (
Casa VARCHAR(15) NOT NULL,
Room VARCHAR(15) NOT NULL,
address VARCHAR(40) NOT NULL,
zona VARCHAR(25) NOT NULL,
mq_esclusivi FLOAT NOT NULL,
mq_condivisi FLOAT NOT NUL,
mq_accessori FLOAT,
parametri_tot SMALLINT NOT NULL,
fascia VARCHAR(6) NOT NULL,
valore_max FLOAT NOT NULL,
prezzo_max FLOAT UNIQUE,
Rent FLOAT NOT NULL
);

ALTER table canone_concordato ALTER column_mq_condivisi DROP not null;

ALTER table canone_concordato RENAME indirizzo TO address
ALETR table canone_concordato RENAME address TO address;

ALTER TABLE ten_details
ALTER COLUMN start_date TYPE TIMESTAMP USING start_date::TIMESTAMP;

ALTER TABLE ten_details
ALTER COLUMN ten_name TYPE VARCHAR(50); (if the old data type is TEXT)

ALTER TABLE employees
ALTER COLUMN Rent TYPE FLOAT USING Rent::FLOAT;  (if the old data type is VARCHAR)

SELECT constraint_name
FROM information_schema.table_constraints
WHERE table_name = 'ten_details' AND constraint_type = 'UNIQUE';
```

5. INSERT INTO, SET, UPDATE, TEMPORARY TABLE

INSERT INTO allows us to add new rows of data in a table when use in combination with VALUES.

```
INSERT INTO ten_details (ten_name, sex, hometown,
start_date, end_date, room, rent, address, area)
VALUES(   …..
);
```

```
INSERT INTO ten_details (room, rent)
VALUES(‘big’, 450),
(‘small’, 320),
(‘medium’, 400),
…
```

We can also use it to add the result of a SELECT to insert values from another table or filtered values

```
INSERT INTO ten_details (name, sex)
SELECT name, sex
FROM OG
WHERE name LIKE ‘T%’ AND sex = ‘m’;
```

Or to add a single default value in a series, this works **as long as the other columns are nullable** or have **default values**.

```
INSERT INTO users (name) VALUES ('Bob');
```

```
INSERT INTO OG ( hometown)
VALUES('Rome'),  ('Ostia'),  (‘Fiumicino')…;
```

If we want to manually insert some column values, while leaving others to be filled in later (possibly using data from another table), we can:

- Leave those columns as `NULL`, or use the `DEFAULT` keyword in the `INSERT INTO` statement — as long as the column definition allows `NULL` or has a default value defined in the `CREATE TABLE` command.
- Later, we can use an `UPDATE` statement to fill in the missing values by referencing another table

TEMPORARY TABLE is a special type of table that exists **only for the duration of a database session or transaction**. It is used to store intermediate results without affecting the main database schema.

```
CREATE TEMPORARY TABLE temp_hometown
(hometown VARCHAR(255) );
```

The **UPDATE ... SET** statement is used to modify values in one or more columns of a table. You can assign fixed values or use values from another table (via JOIN or FROM). To target specific rows, you can use WHERE clauses or conditional logic with CASE WHEN.
Note: SET must always be used with UPDATE

In this example we’ll use the temporary table hometown from before

```
UPDATE ten_details
SET rent=1
WHERE rent is null
```

UPDATE OG SET hometown=temp_hometown.hometown
from temp_hometown;

In the first case we can first recall the other table in the insert table and set the remnant columns as 'default'.
At this point we can either recall the columns we have the list of values for from a temporary table/ normal table or use a case in which the command when is used for every rows (very tedious). A WHERE can be add at the end of the INSERT INTO if we want a condition on the values recalled

UPDATE new_table
SET hometown = CASE name
WHEN 'Manolo' THEN 'Fiumicino su'
WHEN 'John Doe' THEN 'New York'
WHEN 'Jane Smith' THEN 'Los Angeles'
ELSE 'unkown_hometown'
END;

In an `INSERT INTO` using values from another table, we can set the remaining columns as `DEFAULT` (if allowed). We can:
Pull values from a temp or regular table,
- Use `CASE WHEN` per row (tedious for many rows),
- Add a `WHERE` clause to filter source rows.

The following code updates a column in the `OG` table by using matching data from the `ten_details` table. Let's break it down:

UPDATE OG SET ten_name=ten_details.ten_name
from ten_details
Where ten_details.tennant_id=og_id

6. CHECK

sixth example: use the **CHECK** constraint while creating a table so that we make sure that a certain column cannot have values of a certain type. CHECK is used to customize CONSTRAINTS that adhere to a certain condition

CREATE TABLE dati_catastali (Casa varchar(15),
foglio int,
particella int,
 sub int,
cat_catastale char (2),
classe int, zona_catastale int,
classe_energetica char (1),
vani float,
rendita_cat float,
mq float CHECK mq > 80);

In the CHECK command it's worth noting that if we at first insert a value that is not allowed by the check constraint, when we after insert the right values, SQL will keep track of the wrong attempts by not reporting the id's of the failed attempts

7. DELETE

DELETE is used to remove all rows from a table (if we wanted to delete only certain columns, we can use ALTER TABLE DROP). If we want to target only specific rows to delete, we can use where to filter such rows

DELETE FROM ten_details;

DELETE FROM users
WHERE name = 'Tyron';

COMMON TABLE EXPRESSION (CTE)

1. Single CTE

A **Common Table Expression (CTE)** is a **temporary named result set** defined using the **WITH** keyword. It can be used within a **larger SQL query** to simplify complex logic, avoid repetition, and improve readability. Unlike subqueries, which are embedded directly inside a main query, CTEs allow you to structure queries in multiple steps, almost like **creating temporary virtual tables** that only exist during the execution of that query.

One of the main advantages of a CTE is that it allows you to **reuse calculated values or aggregated data without having to rewrite or duplicate the logic**.
This is especially useful when a complex query — such as an average per country, a filtered join, or a grouped aggregation — serves as the **basis for multiple follow-up queries**.
By placing that logic inside a WITH clause, you can then **build on it with different conditions or operations**, much like referencing a virtual table. This **saves time, avoids errors, and keeps your code clean and modular**.

```
WITH avg_rent_per room AS (
   SELECT Room, AVG(rent) AS avg_rent
   FROM ten_details
   GROUP BY Room)
SELECT Room,  avg_rent
FROM avg_rent_per room
WHERE avg_rent > 350;
```

Through CTE's, we can also perform other actions if combined with other keywords like, INSERT INTO, UPDATE-SET, DELETE. For this purpose, let's create a copy of ten_details and call #temp1

```
SELECT * INTO temp1 FROM ten_details

WITH cte_1 AS (
SELECT * FROM temp1 WHERE name LIKE 'T&'
)
INSERT INTO #temp1 SELECT * FROM cte_1;


WITH cte_1 AS (
SELECT * FROM temp1 WHERE tennant_id IN (2,3,4)
)
UPDATE  temp1 SET tennant_id = 101 WHERE tennant_id IN (SELECT DISTINCT tennant_id FROM
cte_1)

WITH cte_2 AS (
SELECT * FROM temp1 WHERE tennant_id = 1
)
DELETE FROM temp1 WHERE tennant_id IN (SELECT DISTINCT tennant_id FROM cte_2)
```

How it works:

- The WITH clause defines a **CTE named cte_name**.
- This CTE can then be **queried as if it were a regular table** in the main query that follows.
- You can reference it **once or multiple times** within the same query.
- CTEs can also be **chained or nested** by defining multiple ones in the same WITH clause, separated by commas.

Use Cases:

- **Breaking down complex queries** into manageable steps.
- **Reusing calculated values** or aggregated data without repeating the logic.
- **Improving readability and maintainability** of SQL scripts.
- **Creating recursive queries**, though this is an advanced use case.

NOTE:

- CTEs exist **only during the execution of the main query**.
- You **cannot index** or **persist** a CTE — they are **temporary**.
- CTEs are especially helpful when used **with complex joins**, **aggregations**, or **window functions**

2. Multiple CTE

We can perform the same actions as before also when we set multiple CTE by using UNION ALL

```
WITH cte_1 AS (
SELECT * FROM temp1 WHERE tennant_id IN (1,2)
), cte_2 AS (
SELECT * FROM temp1 WHERE tennant_id IN (3,4)
)
SELECT * FROM cte1
UNION ALL
SELECT * FROM cte2

WITH cte_1 AS (
SELECT * FROM temp1 WHERE tennant_id = 1
), cte_2 AS (
SELECT * FROM temp1 WHERE tennant_id = 3
)
SELECT * INTO temp2 FROM(
SELECT * FROM cte1
UNION ALL
SELECT * FROM cte2);

WITH cte_1 AS (
SELECT * FROM temp1 WHERE tennant_id = 1
), cte_2 AS (
SELECT * FROM temp1 WHERE tennant_id = 3
)
INSERT INTO temp2  SELECT * FROM(
(SELECT * FROM cte1
UNION ALL
SELECT * FROM cte2);

WITH cte_1 AS (
SELECT * FROM temp1 WHERE tennant_id = 1
), cte_2 AS (
SELECT * FROM temp1 WHERE tennant_id = 3
)
UPDATE temp2 SET tennant_id =100 WHERE tennant_id IN (SELECT DISTINCT tennant_id FROM cte1
UNION ALL
SELECT * FROM cte2);
```

```
WITH cte_1 AS (
SELECT * FROM temp1 WHERE tennant_id = 1
), cte_2 AS (
SELECT * FROM temp1 WHERE tennant_id = 3
)
DELETE FROM temp2 WHERE tennant_id =100 WHERE tennant_id  NOT IN (
SELECT DISTINCT tennant_id FROM cte1
UNION ALL
SELECT * FROM cte2);
```

3. Recursive CTE

A **recursive CTE** is a Common Table Expression that refers to itself in order to repeatedly execute a query and build up a result set — typically used for **hierarchical data** (like org charts or folder trees) **or** iterative calculations (like factorials or sequences)

It has two components

- **Anchor member** – the base case (initial result set).
- **Recursive member –** the query that references the CTE itself and runs repeatedly until a stopping condition is met.

The anchor member is the starting point from where the first iteration takes place. Then every result of an iteration becomes the starting point for the next

The following code produces a column called n with 5 rows containing numbers from 1 to 5

```
WITH RECURSIVE cte AS (
   SELECT 1 AS n
   UNION ALL
   SELECT n + 1 FROM cte WHERE n < 5
)
SELECT * FROM cte;
```

The following code is based on the previous and calculates the factorial of 5 (or any other number). The logic here is that exp(sum(ln…))) = exp (ln(1) +…+ln(n)) = exp(ln(1·…·n)=1·…·n. With n= 5. We have exp(sum(ln…))) = exp (ln(1) +…+ln(5)) = exp(ln(1· 2 · 3 · 4 · 5)= 1· 2 · 3 · 4 · 5 = 120

```
WITH RECURSIVE cte AS (
   SELECT 1 AS n
   UNION ALL
   SELECT n + 1 FROM cte WHERE n < 5
)
SELECT round(exp(sum(ln(n)))) AS factorial FROM cte;
```

CONDITIONAL EXPRESSION AND PROCEDURES

1. CASE

The **CASE** statement is used to perform conditional logic in SQL — similar to if-else using WHEN-THEN or switch in programming languages. It checks conditions **in order**, and returns the result for the first condition that is true. If none match, the ELSE clause is returned (optional).

Use case: Categorizing data or creating derived columns based on conditions.

```
SELECT ten_name, sex
CASE WHEN (ten_name like 'Ty%') THEN 'landlord'
ELSE 'tenant'
END AS landlord_tenants
FROM ten_details;

SELECT ten_name, sex,
CASE WHEN rent BETWEEN 350 AND 390 then 'centre'
         WHEN rent >= 600 THEN 'fiumicino'
ELSE 'Rome'
END AS landlord_tennants
FROM ten_details;
```

This is the **GENERAL CASE** in which one gives a condition after the when and for this reason there more freedom when choosing what it's going to be evaluated. This operator works like an IF/ELSE

```
SELECT *,
CASE rent WHEN 360 THEN 'centre'
       WHEN 350 THEN 'centre'
ELSE 'other rooms'
END AS rooms_rents
FROM ten_details;

SELECT ten_name,
CASE ten_name WHEN 'Tyron' THEN 'landlord'
WHEN 'Tyron' 1 THEN 'landlord'
WHEN 'Tyron 2' THEN 'landlord'
ELSE 'tenants'
END AS landlord_tenants
FROM ten_details;
```

This is the **CASE EXPRESSION** and here the condition is sort of stated in the expression so that the when just process the action, so when the user already knows what they want to find this situation is more straightforward and quicker, therefore this scenario is mostly used to check equalities

```
SELECT
SUM (CASE room WHEN 'Dx' THEN 1
                          WHEN 'Sx' THEN 1
ELSE 0
END) AS Dx_number
FROM OG

SELECT
SUM (CASE room WHEN 'Dx' THEN 1
ELSE 0
END) AS Dx_number,
SUM (CASE room WHEN 'Sx' THEN 1
ELSE 0
END) AS Sx_number
FROM OG;
```

The first code allows us to count the numbers of a category (Dx and Sx) and return them in a column, or two as in this case. If the WHEN is stated in the same CASE it will just show it as a SUM of both categories

2. COALESCE()

The COALESCE() function takes **multiple arguments** and returns the **first non-NULL** value in the list. It's very useful for **replacing NULLs** with fallback values, or for checking multiple optional columns.

Use case: Handling missing data by specifying default values.

```
SELECT COLAESCE(null, null, 'C')  ⟶  returns C
```

```
SELECT rent, end_date,
COALESCE(end_date , '1987-12-12 11:00:00'),
COALESCE(rent, 0)
FROM ten_details
ORDER BY tenant_id;
```

The **COALESCE** command checks for null values in the column (the first object) stated in the parentheses row by row. When the first null is found, it replaces it with the next object we wrote in the parentheses, which might be the value on the same row of another column or whatever string (timestamp etc.) is next in the command. This command is useful when we want to change a null value with something else without modifying the table

```
SELECT CAST ('5' as integer) AS new_num;

SELECT '5' :: integer

SELECT end_date, COALESCE(CAST(end_date as text), 'present')
FROM ten_details;

SELECT rent, end_date, COALESCE(CAST(end_date as text), 'present'),
COALESE(CAST(rent as varchar), 'zero')
FROM ten_details
ORDER BY tenant_id;

SELECT end_date, COALESCE (end_date :: text, 'present')
FROM ten_details;

SELECT CHAR_LENGTH(rent :: varchar) FROM ten_details;
```

The **CAST** (or ::) can be used to change the type of a data into another type. This command must "make sense", i.e. cannot change 'five' into integer since SQL won't understand five as a number. It can be used in combination with **CHAR_LENGTH** to calculate the number of character/digits used to store a number in a row. It can be used multiple times in the same select

3. NULLIF()

The **NULLIF(a, b)** function compares two expressions:

- If `a = b`, it returns **NULL**
- If they're different, it returns the **first value `a`**

This is useful to **avoid errors like division by zero** by intentionally returning NULL when two values match.

Use case: Preventing divide-by-zero errors or ignoring repeated values.

```
SELECT rent,
NULLIF (CASE WHEN rent-460<0 THEN 0 ELSE rent-460 END, 0 )
AS difference
FROM ten_details;

SELECT rent, SUM(rent) /
NULLIF(
(CASE WHEN rent - 460 < 0 THEN 0 ELSE rent - 460 END), 0)
AS difference
FROM ten_details group by rent;
```

NULLIF returns NULL if the two arguments inside the parentheses are equal. Otherwise, it returns the first argument.
Here's an example of how to combine it with a CASE statement.
In the second line, the usefulness of NULLIF is more evident: dividing by zero would normally cause an error. But by using NULLIF, if the denominator is zero, the expression returns NULL instead of attempting the division — avoiding the error and making the whole result NULL.

4. ISNULL

ISNULL() is a function that replaces **NULL with a specific value**. It's commonly used in **SQL Server**. Other databases (like PostgreSQL, MySQL) don’t have isnull() and use **COALESCE()** or similar functions instead.

Use case: Showing a default value when a column is NULL

SELECT ISNULL(null, ‘1st value is null’) $\longrightarrow$ Will return ‘1st value is null’ when there is a NULL

```
SELECT customerid, ISNULL(email, 'mail is NA'), ISNULL (phonenumber, 'phone is NA')
FROM customers
```

On postgree, we’d use for the same purpose COALESCE

```
SELECT customerid, COALESCE(email, phonenumber, 'contact NA')
FROM customers
-- This line will display email if email is NOT NULL, phone number if email is NULL and ‘contact NA’ when phonenumber is also NULL
```

\

STORED PROCEDURES and VIEW

A **STORED PROCEDURE** is a **pre-written block of SQL code** that we save in the database and can **reuse/call** whenever needed — instead of writing the same queries over and over again.
We can think of it like a **recipe**: we define it once (with all the steps and logic inside), and then just call it by name to execute all the steps at once.
It uses the keywords BEGIN and END. To reuse the block of code we must call through **EXECUTE** (or EXEC)command and it can be altered the same we would with a table

```
CREATE PROCEDURE sp_test AS
BEGIN
SELECT * FROM ten_details
END
```

```
CREATE PROC sp_test AS
BEGIN
SELECT name, rent FROM ten_details
END
```

```
EXECUTE sp_test
```

```
ALTER PROC sp_test AS
BEGIN
SELECT name, rent, sex FROM ten_details
END
```

A**VIEW** is a virtual table used to store a SQL query. It usually used for two main reasons

- **To reduce the complexity of the code**: if we know we will be using the same complex code multiple times in the future, we can create a VIEW containing such code to avoid writing every time. Additionally, if a non-technical colleague needs to retrieve the result of a query but doesn't have the competence to write it on their own, we can store the query in a VIEW so that they can access the result by just running the VIEW.
- **To implement security**: If there's sensitive information that shouldn't be accessible to the whole team, we can create a `VIEW` that includes only the general columns — excluding the sensitive ones. This way, team members can access and work with the general data through the `VIEW`, without ever seeing the restricted information.

It's of course possible to operate distinct actions in the code of the view or modify the VIEW itself at any time with UPDATE-SET. To reuse the code in the VIEW we can simply select it's columns as if it was a normal table.

```
CREATE VIEW view_1 AS(
select rent, og from ten_details join OG
on ten_details.ten_name=OG.ten_name);
```

```
select * from view_1;
```

```
CREATE OR REPLACE VIEW view_1 AS(
select rent, og, area, address,sex
 from ten_details join OG
on ten_details.ten_name=OG.ten_name);
```

```
UPDATE view_1
SET ten_name = 'Tanno'
```

The **VIEW** is a database object that creates virtual (they are not physically stored as data) tables in which are stored combinations of table and conditions, more specifically queries, that are used often and can be recalled anytime instead of having to write the code/query every single time.
To recall the query just use select and it will show the columns use for the ioin in the VIEW

WHERE name = ‘Tyron’

DROP VIEW IF EXISTS view_1;

NOTE – Difference between STORED PROCEDURES and VIEWS:

- STORED PROCEDURE: a **reusable block of code** that can **perform actions** (insert, update, delete) and also **return data** if needed.
- VIEW: a **read-only structure** based on a SQL query — it only **returns data**, and nothing else

INDEX IN SQL

Whenever we run a query to retrieve specific rows — for example, by filtering on some values — the SQL engine normally scans the entire table row by row, checking each value. This is called a **sequential scan**, and on large datasets, it can significantly slow down performance.
To make this process more efficient, we can create an **index** on one or more columns. An index acts like a **lookup table** that maps values to the physical locations of their corresponding rows. This allows the SQL engine to **jump directly** to the matching rows without scanning everything.
For instance, if we want to find all the rows where `name = 'Tyron'`, without an index, SQL checks every row one by one. But with an index on the `name` column, the engine uses the index it internally assigned to such values in the column and quickly find the position of `'Tyron'` and retrieve the corresponding rows much faster — like using a book's index to find a topic instead of reading every page.

```
CREATE INDEX idx_1 ON ten_details(ten_name)
```

```
CREATE INDEX idx_1 ON ten_details(ten_name, hometows DESC)
```

Then, if we want to drop the index we can use

```
DROP INDEX ten_details.idx_1
```

We can search for index in the 'object explorer' by going to the dataset>schemas> public> tables> column_name>indexes

NOTE- how to use INDEX:
- Indexes **speed up reads** but **slow down writes** (inserts, updates, deletes), because the index must be updated too.
- **Too many indexes** = bloated storage + slower writes. Use them **strategically**.

1. CLUSTERED INDEX
A CLUSTERED INDEX determines the physical order of the rows in a table, which means that the table's data is stored on disk in the same order as the index. In practice, it defines how the table is organized by the column(s) being**.**
There can be only one clustered index per table, because data can only be physically sorted in one way and if we want to define a new clustered index we must first drop the first index: the primary is usually (but not always) the clustered index by default.

```
CREATE CLUSTERED INDEX idx_1 ON ten_details(sex DESC, rent ASC)
```

2. NON _CLUSTERED INDEX
NON-CLUSTRED INDEX does not change the physical order of the data in the table.
Instead, it creates a separate structure that stores the indexed column(s) along with pointers to the actual rows in the table.
So we can have multiple clustered index on the same column.

```
CREATE CLUSTERED INDEX idx_1 ON ten_details(sex DESC, rent ASC)
```

IMPORTING/EXPORTING A DATABASE

This is a summary on how to import (or export) a dataset file from (or to) another program like excel, google spreadsheet or sublime text editor to the SQL environment in the form of a correct .cvs file.

The first step is to create a table cause the import command doesn't do it automatically. The columns in the CREATE must be consistent with the types of data, same goes for the CONSTRAINTS that are going to be implemented.

After creating the table that's going to hold the dataset, the procedure to import the file goes through the following steps:

1. Save the spreadsheet as .CSV and make sure to remember where the file is stored in the computer (when creating the .CVS file it is possible to check the exact location by right-clicking on the file, then properties and it will show the location.
2. Create a database on PGadmin4 by right-clicking on "databases" and choose "create" and on the general tab insert the name of the new table.
3. Create the new table via SQL query with columns that have the same data types and consistent constraints with the spreadsheet (it doesn't matter if the columns on the CSV have capital letters).
4. Refresh the table just in case, then go to schemas>public>table_name and right-click on table_name to open the import/export window
5. On the import/export window, general tab, toggle the option for either import or export, select the full path of the file by either typing it or chose from the list.
6. Go on options tab toggle the header in case the columns on the spreadsheet have a column name, choose the right delimiter and quote.
7. Go to the "columns" tab and in the list of columns make sure that there is consistency with the spreadsheet by possibly eliminating columns.
8. Click OK and the import should start. If there's any error during the actual importing process a small window will open notifying the error, then click on "view process" and on the new "process" tab click on the paper symbol to see what kind of error happened. It might be useful to then report the error to ChatGPT and get information about the solution.

NOTE: Excel, specifically the European version, uses commas instead of periods for decimals but PGadmin4 uses decimals with commas, which might create a discrepancy during the import; Excel uses delimiters as semi columns (;) while PGadmin uses comma (,), then change the delimiter to the one that suits the better case; check the consistencies with between the columns in the table on PGadmin4 and the ones in Excel; if a CONSTRAINT implies NOT NULL, then in the spreadsheet one has to put zero or other placeholder values where the data are not present; if TIMESTAMP is present then the time format must be in the same format; it is required to recall the exact position in the computer of where the spreadsheet was saved.

1 **First, import the data without constraints** (except maybe `PRIMARY KEY` if you're confident about it).
2 **Then, analyze the dataset** to understand its structure, unique values, missing values, and patterns.
3 **Finally, add constraints (`NOT NULL`, `UNIQUE`, `FOREIGN KEY`, etc.)** based on what you observe.

Avoid Import Errors: If constraints (like `NOT NULL` or `UNIQUE`) are applied before importing, the import could fail due to unexpected data inconsistencies.
✓ **More Flexibility:** Without constraints, you can clean or transform data as needed **before enforcing rules**.
✓ **Better Understanding of Data:** If you are **not familiar with the dataset**, you can analyze it first and apply appropriate constraints **only when necessary**

ADDITIONAL USEFUL CODES

- **MODE:**

```
SELECT duration, COUNT(*) AS freq
FROM flight_prices
GROUP BY duration
ORDER BY freq DESC
LIMIT 1;
```

```
SELECT conname AS constraint_name,
contype AS constraint_type,
a.attname AS column_name
FROM pg_constraint c
JOIN pg_class t ON c.conrelid = t.oid
JOIN pg_attribute a ON a.attnum = ANY(c.conkey)
AND a.attrelid = t.oid
WHERE t.relname = 'your_table_name';
```

This code is useful to retrieve a list of all the columns of a table with their constraint types:
P=primary
U=unique
F=foreign
C=check

```
SELECT  continent, name, area FROM world x
  WHERE area >= ALL
    (SELECT area FROM world y
        WHERE y.continent=x.continent). (correlated  or synchronized subquery)
```

I

```
UPDATE global_land_temp
SET averagetemp = LEFT(averagetemp, POSITION('.' IN averagetemp) + 3)
WHERE averagetemp ~ '\\..*\\.';
```

1 UPDATE global_land_temp

- Updates the global_land_temp table.

2 SET averagetemp = ...

- This modifies the averagetemp column.

3 LEFT(averagetemp, POSITION('.' IN averagetemp) + 3)

- LEFT(string, n):
  - Extracts the first n characters from the string.
- POSITION('.' IN averagetemp):
  - Finds the position of the first dot (.) in averagetemp.
  - Example: If averagetemp = '24.418.000.000', POSITION('.') returns 3 (because the first dot is in the 3rd position).
- + 3:
  - Ensures that 3 digits after the first dot are kept.
- Example Transformation:
  - averagetemp = '24.418.000.000'

  - POSITION('.') = 3, so 3 + 3 = 6
  - LEFT('24.418.000.000', 6) → '24.418' (keeps only the first dot and 3 decimals).

4 WHERE averagetemp ~ '\\..*\\.'

- This filters only the rows that need fixing.
- ~ is PostgreSQL's regex match operator.
- '\\..*\\.':
  - \\. → Matches a literal dot (.) (double backslash is needed in SQL regex).
  - .* → Matches any number of characters after the first dot.
  - \\. → Ensures there's another dot after that.

]

NOMENCLATURE

- strings

timestamp

STRUCTURE ANY

OTHER BUSINESS

```
select first_name, last_name, title
from actor
join film_actor on actor.actor_id = film_actor.actor_id
join film on film.film_id=film_actor.film_id
where first_name = 'Nick'
and last_name = 'Wahlberg'
order by title
```

(Explore the commands REGEXP,

In SQL, a **primary key** is a column (or a set of columns) in a table that uniquely identifies each row in that table. It ensures that no two rows have the same value for the primary key column(s), and it also prevents null values in the primary key column(s).
Here are some key characteristics of a primary key:

d. **Uniqueness**: Each value in the primary key column(s) must be unique. This guarantees that every row in the table can be uniquely identified.

e. **Not NULL**: A primary key column cannot contain NULL values because it must uniquely identify each record in the table.
f. **Immutability**: Once set, the value of a primary key should not change. Changing a primary key can disrupt relationships between tables in a database.