* ***SQL and their types ?***

**SQL (Structured Query Language)** is the standard language used to interact with relational databases. It allows us to perform operations like retrieving, inserting, updating, and deleting data from tables.

**🔹 1. DDL (Data Definition Language)**

Used to define and modify the structure of database objects like tables and schemas.

| **Command** | **Description** |
| --- | --- |
| CREATE | Creates a new table, view, or database |
| ALTER | Modifies an existing object (like adding a column) |
| DROP | Deletes an object like a table or database |
| TRUNCATE | Deletes all rows from a table without logging individual row deletions |

**🔹 2. DML (Data Manipulation Language)**

Used to manipulate data inside database objects.

| **Command** | **Description** |
| --- | --- |
| SELECT | Retrieves data |
| INSERT | Adds new data |
| UPDATE | Modifies existing data |
| DELETE | Removes data |

**🔹 3. DCL (Data Control Language)**

Used to control access to data.

| **Command** | **Description** |
| --- | --- |
| GRANT | Gives user access privileges |
| REVOKE | Takes back permissions |

**🔹 4. TCL (Transaction Control Language)**

Used to manage transactions and ensure data integrity.

| **Command** | **Description** |
| --- | --- |
| BEGIN or START TRANSACTION | Begins a transaction |
| COMMIT | Saves the transaction |
| ROLLBACK | Reverts changes if there’s an error |

**🔹 5. DQL (Data Query Language)**

Some classify SELECT alone under **DQL**, which is specifically used to **query data** from the database. While it's technically part of DML in some classifications, many separate it for clarity.

**"What are constraints in SQL? Can you explain different types of constraints with examples?"**

In SQL, constraints are rules enforced on data columns to maintain data integrity, accuracy, and reliability in the database.

They ensure that only valid data gets inserted, updated, or stored in the database. Constraints can be defined at the column level or table level when creating or altering a table.

**🔹 1. PRIMARY KEY**

* Uniquely identifies each record in a table.
* Cannot contain NULL values.
* Each table can have only one primary key.

**🔹 2. FOREIGN KEY**

* Establishes a link between the data in two tables.
* Ensures **referential integrity**.

**🔹 3. UNIQUE**

* Ensures that all values in a column are different.
* Unlike PRIMARY KEY, it **can contain NULLs**.

**🔹 4. NOT NULL**

* Prevents null values from being inserted into a column.

**🔹 5. CHECK**

* Validates values against a condition before inserting or updating.

**🔹 6. DEFAULT**

* Sets a default value if none is provided during INSERT.

"***Can you explain what normalization and denormalization?***

**Normalization** is the process of organizing data in a database to minimize redundancy and improve data integrity. The main goals of normalization are to ensure that data is stored logically and that relationships between data are clear.

1. **Purpose**: The primary purpose of normalization is to eliminate duplicate data and ensure that data dependencies are logical. This helps maintain data integrity and reduces the risk of anomalies during data operations.
2. **Normal Forms**: Normalization is typically achieved through a series of steps known as normal forms. The most commonly referenced normal forms are:
   * **First Normal Form (1NF)**: Ensures that each column contains atomic values and that each entry is unique.
   * **Second Normal Form (2NF)**: Builds on 1NF by ensuring that all non-key attributes are fully dependent on the primary key, eliminating partial dependencies.
   * **Third Normal Form (3NF)**: Further refines the structure by ensuring that there are no transitive dependencies among non-key attributes.
   * **BCNF** aims to eliminate redundancy and prevent anomalies that can occur in 3NF by ensuring that all functional dependencies are based on superkeys.

**Denormalization**, on the other hand, is the process of intentionally introducing redundancy into a database design. This is often done to improve performance, especially in read-heavy applications.

1. **Purpose**: The main goal of denormalization is to optimize read operations by reducing the number of joins required in queries. This can lead to faster data retrieval and simpler query structures.
2. **When to Use**: Denormalization is typically considered when the performance benefits outweigh the drawbacks of increased redundancy. It’s common in data warehousing and reporting scenarios where read performance is critical.

***Aggregate Functions***

Aggregate functions in SQL are used to perform calculations on a set of values and return a single result.  
These functions help summarize data, and they’re commonly used in **SELECT** queries with the GROUP BY clause or even without it for global summarization.

**Common Aggregate Functions:**

**1. COUNT()**

"The COUNT() function returns the number of rows in a result set, or the number of non-NULL values in a column."

**2. SUM()**

"The SUM() function adds up all the values in a column."

**3. AVG()**

"The AVG() function calculates the average of a numeric column."

**4. MIN()**

"The MIN() function returns the smallest value in a column."

**5. MAX()**

"The MAX() function returns the largest value in a column."

**Using Aggregate Functions with GROUP BY**

"Aggregate functions become even more powerful when combined with GROUP BY. This lets us summarize data based on categories or groups."

**Using Aggregate Functions with HAVING**

"The HAVING clause filters the results of an aggregate query. It's used with GROUP BY to filter groups based on aggregate conditions."

By default, **COUNT()** does not count NULL values, but **SUM()**, **AVG()**, **MIN()**, and **MAX()** will ignore NULL values.  
However, if I want to **include NULL values** in the calculation, I could use a **conditional aggregation** or a function like COALESCE() to replace NULL with a default value."

-- Replace NULL values with 0 in the SUM calculation

SELECT SUM(COALESCE(sales\_amount, 0)) FROM sales;

"What’s the difference between COUNT(\*) and COUNT(column\_name) ?

**COUNT(\*)** counts all rows, including rows with NULLs.

**COUNT(column\_name)** counts only the rows where the specified column has a non-NULL value.

key difference **between WHERE and HAVING is:**  
WHERE filters rows before aggregation, while  
 HAVING filters groups after aggregation.

"Can you explain the different types of SQL joins and when you'd use them?"

SQL JOINs are used to **combine rows from two or more tables** based on a related column between them — usually a foreign key.

**1. INNER JOIN**

Returns only the **matching rows** from both tables.

**2. LEFT JOIN (or LEFT OUTER JOIN)**

Returns **all rows from the left table**, and matching rows from the right table. If there’s no match, returns NULLs from the right side.

**3. RIGHT JOIN (or RIGHT OUTER JOIN)**

Opposite of LEFT JOIN — returns **all rows from the right table**, and matching rows from the left table. If there’s no match, returns NULLs from the left side.

**4. FULL OUTER JOIN**

Returns **all records** from both tables. When there’s no match, shows NULLs for the missing side.

**5. CROSS JOIN**

Returns the **Cartesian product** — every combination of rows from both tables.

**6. SELF JOIN**

A table joined with **itself** — useful for hierarchical or comparative data.

**INNER JOIN** only returns rows where **there is a match in both tables**.( Only matching rows)

**OUTER JOIN** returns **all rows from one or both tables**, and fills in NULLs where there is no match.( Includes unmatched rows too)

**JOIN and UNION both combine data from two or more tables, but they work in very different ways:**

🔹 A JOIN combines **columns** from different tables based on a related key.( Horizontal combination)  
🔹 A UNION combines **rows** from two result sets that have the **same structure**.( Vertical combination)

**UNION vs UNION ALL**

UNION removes duplicates.

UNION ALL **includes duplicates**, and is faster because it skips the deduplication step.

**what Set Operations are in SQL**

Set Operations are used to **combine the results of two or more SELECT statements.** **and operate on entire result sets — not individual columns.**

**UNION** – Combine and remove duplicates(Returns all **unique names** from both tables — no duplicates.)

**UNION ALL** – Combine **with** duplicates.( Returns all names from both tables, including duplicates (faster performance)

INTERSECT – Common rows in both result sets.( *Useful when we want to find overlapping data.)*

EXCEPT / MINUS – Subtract one result set from another

**How is this different from using JOINs?**

**JOINs** are **used to combine rows side-by-side based on related keys,** like foreign keys.  
But **Set Operations** **combine rows top-to-bottom, as long as the column structures match** — no relationships needed.

**what a subquery ?**

subquery is a SQL query that's nested inside another query.

it’s used to perform intermediate steps, like filtering, aggregation, or preparing a temporary result set that the outer query can use.

**-Subquery in WHERE Clause =** *To filter data based on another query.*

***-Subquery in SELECT Clause =*** *To bring in extra info like a single value or count.*

*-***Subquery in FROM Clause =** *To create a temporary derived table (inline view).*

***Correlated vs Non-Correlated Subqueries***

**"**Subqueries can be either correlated or non-correlated:

**🔹** A non-correlated subquery runs once and its result is used by the outer query.

**🔹** A correlated subquery runs once per row of the outer query and references columns from it.

**Can you explain what a View is in SQL and when you'd use it?**

A View in SQL is a virtual table based on the result of a SELECT query.  
It doesn’t store data itself — instead, it pulls live data from underlying tables every time it’s queried. Think of it like a saved query that looks and behaves like a table.

Why Use Views?

**Simplify complex queries** → Write it once, reuse it multiple times.

**Improve readability & maintainability** → Abstract away joins and filters so the end user sees only what they need.

**Enhance security** → Expose only selected columns (e.g., hide salary, SSN) to users who don’t need access to full tables.

**Support data abstraction** → Allows business users or analysts to query complex datasets without knowing the schema.

**Enforce consistent business logic** → For example, always using the same filters for "active customers" or "latest transactions.

There are a few limitations to keep in mind:

* Views don’t store data → each query re-runs the underlying SQL.
* You can’t always INSERT, UPDATE, or DELETE from a view — especially if it includes aggregates, joins, or GROUP BY.
* Performance can be an issue for very complex views if overused or nested too deeply.

**Can you explain what an index is in SQL, and how it works?**

An index in SQL is a data structure that improves the speed of data retrieval operations on a table at the cost of additional space and slower updates.  
You can think of it as a lookup table that allows the database to find rows more efficiently without scanning the entire table. Indexes are especially useful when you’re working with large tables and need fast searches, joins, and sorts.

| **Index Type** | **Description** |
| --- | --- |
| **Single-Column Index** | Index on a single column, improving searches and filtering. |
| **Composite Index** | Index on multiple columns, useful when filtering on more than one column. |
| **Unique Index** | Ensures no duplicate values in the indexed column(s). |
| **Full-Text Index** | Optimized for full-text searches on large text fields. |
| **Clustered Index** | Data is stored in the order of the index (the table is sorted). |
| **Non-Clustered Index** | The index is stored separately, and the table contains pointers to the rows. |

**Clustered vs Non-Clustered Indexes**

**Clustered Index**:

* The data in the table is physically organized **according to the index**.
* A table can have **only one clustered index**.
* By default, the **primary key** creates a clustered index.

**Can you explain what a window function is in SQL, and how it's different from aggregate functions?**

A window function performs a calculation across a set of table rows that are related to the current row — called a window — without collapsing the rows.

This is different from aggregate functions like SUM() or AVG() in a GROUP BY query, which return one row per group.

Window functions return a value for every row, while still giving access to aggregate-like operations.

| Function | Description |
| --- | --- |
| ROW\_NUMBER() | Assigns a unique sequential number per partition |
| RANK() | Ranks with gaps in case of ties |
| DENSE\_RANK() | Ranks without gaps |
| NTILE(n) | Distributes rows into n buckets |
| SUM(), AVG() | Running totals or averages |
| LEAD() / LAG() | Access next or previous row’s value |
| FIRST\_VALUE() / LAST\_VALUE() | Get first/last value in window |

**Why use a window function instead of GROUP BY**?

GROUP BY reduces the number of rows — you only get **one row per group**.

But with window functions, you **keep the original row structure** while still performing aggregate-like operations across a window.