1. **Types of commands and their examples.**

In SQL, commands categorized into four types:

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

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

**3. Data Query Language (DQL)**

**4. Data Control Language (DCL).**

**1. Data Definition Language (DDL):** DDL commands are used to define, modify, and delete the structure of database objects like tables, indexes, views, etc.

**a. CREATE TABLE**:

CREATE TABLE employees (

id INT PRIMARY KEY,

name VARCHAR(100),

salary DECIMAL(10, 2)

);

**b. ALTER TABLE**:

**ALTER TABLE** employees ADD COLUMN department VARCHAR(50);

**c. DROP TABLE:** Deletes an existing table.

**DROP TABLE employees;**

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

DML commands are used to manipulate data within database objects like tables.

**a. INSERT INTO:** Inserts new records into a table.

**INSERT INTO** employees (id, name, salary) **VALUES** (1, 'John Doe', 50000);

**b. UPDATE:** Updates existing records in a table.

**UPDATE** employees **SET** salary = 55000 **WHERE** id = 1;

**c. DELETE** **FROM**: Deletes records from a table.

**DELETE** **FROM** employees **WHERE** id = 1;

**3. Data Query Language (DQL):**

DQL commands are used to retrieve data from the database.

1. **SELECT**: Retrieves data from one or more tables.

**SELECT** \* FROM employees;

**4. Data Control Language (DCL):**

DCL commands are used to control access to data within the database.

1. **GRANT**: Grants permissions to users or roles.

GRANT SELECT ON employees TO user1;

b. **REVOKE**: Revokes permissions from users or roles.

REVOKE SELECT ON employees FROM user1;

1. **What is Normalization and denormalization?**

Normalization and denormalization are two complementary database design techniques used to optimize database structure for efficiency, maintainability, and scalability.

Normalization:

Normalization is the process of organizing data in a database to reduce redundancy and dependency. The primary goal of normalization is to eliminate data anomalies and ensure that each piece of data is stored in only one place. This is achieved by dividing large tables into smaller, related tables and defining relationships between them. Normalization typically involves decomposing tables into smaller, more specialized tables and ensuring that each table represents a single entity or concept.

Normalization is usually divided into several normal forms, each with specific rules to guide the decomposition process. The most commonly used normal forms are:

First Normal Form (1NF)

Second Normal Form (2NF)

Third Normal Form (3NF)

Boyce-Codd Normal Form (BCNF)

Fourth Normal Form (4NF)

Fifth Normal Form (5NF)

By normalizing a database, you can reduce redundancy, improve data integrity, and simplify data maintenance. However, normalization may result in an increase in the number of tables and joins required for querying the data, which can impact performance in some cases.

Denormalization:

Denormalization is the process of intentionally introducing redundancy into a database design to improve performance or simplify data retrieval. While normalization aims to reduce redundancy, denormalization trades off redundancy for improved performance by storing redundant data in multiple places. This can involve combining related tables, duplicating data, or precomputing and storing aggregated values.

Denormalization is often used in scenarios where read performance is critical, such as in data warehousing or reporting applications. By storing redundant data, denormalization reduces the need for joins and simplifies queries, leading to faster data retrieval.

However, denormalization comes with trade-offs. It can increase storage requirements, complicate data maintenance, and introduce the risk of data inconsistency if the redundant data is not properly synchronized. Therefore, denormalization should be used judiciously, balancing the benefits of improved performance against the drawbacks of increased complexity and maintenance overhead.

**3. Explain 1NF, 2NF, 3NF.**

**1. First Normal Form (1NF):**

In 1NF, each column in a table must hold atomic (indivisible) values.

This means that each column should contain only a single value, not a list of values or a combination of values.

There should be no repeating groups of columns.

Each row must be unique, meaning there should be a primary key that uniquely identifies each row.

**2. Second Normal Form (2NF):**

To be in 2NF, a table must first fulfill the requirements of 1NF.

Additionally, all non-key attributes (columns) must be fully functionally dependent on the entire primary key.

This means that each non-key attribute should depend on the entire primary key, not just a part of it.

If there are partial dependencies, where non-key attributes depend on only a portion of the primary key, the table is not in 2NF.

**3. Third Normal Form (3NF):**

A table is in 3NF if it is in 2NF and all transitive dependencies have been removed.

Transitive dependency occurs when a non-key attribute is dependent on another non-key attribute, which is dependent on the primary key.

In simpler terms, if A → B → C, where A is the primary key, B is a non-key attribute, and C is another non-key attribute, then C is transitively dependent on A through B.

To remove transitive dependencies, we break down the table into multiple tables, ensuring that each non-key attribute is functionally dependent only on the primary key.

In summary, 1NF ensures that data is stored atomically, 2NF eliminates partial dependencies, and 3NF removes transitive dependencies. Each normal form builds upon the previous one, aiming to reduce redundancy and dependency in database tables.

**4. Share use case where you had to do denormalization in database.**

One common use case for denormalization in a database is in a reporting or analytics system where read performance is critical. Consider a scenario where you have a relational database containing transactional data for an e-commerce platform. The normalized schema might have separate tables for customers, orders, order items, products, and categories. While this normalized schema is efficient for transactional operations, it may not be optimal for generating complex reports or analytics queries that involve aggregations across multiple tables.

In such a scenario, denormalization can be used to improve query performance. For example, you might denormalize the data by creating a single, flattened table that combines information from multiple normalized tables. This denormalized table could include attributes like customer details, order details, product details, and category details all in one place, reducing the need for joins and simplifying complex queries.

By denormalizing the data, you can significantly improve the performance of analytical queries, especially those involving aggregations or data across multiple entities. This approach can be particularly useful in data warehousing environments where the focus is on generating reports and performing analysis rather than on transactional operations.

However, it's essential to carefully consider the trade-offs of denormalization, such as increased storage requirements and the need for data synchronization mechanisms to ensure consistency between the denormalized data and the original normalized tables. Additionally, denormalization should be used judiciously and only when necessary to address specific performance requirements.

**5. What is primary key and foreign key?**

**Primary Key:** A primary key is a column or a set of columns in a relational database table that uniquely identifies each record or row in that table. It serves as a unique identifier for the data stored in the table, ensuring that no two rows can have the same primary key value. Primary keys play a crucial role in enforcing entity integrity, as they guarantee the uniqueness and integrity of the data within a table.

Primary keys are typically defined when creating a table and are often based on one or more attributes that uniquely identify each entity or record. Common examples of primary keys include a single auto-incrementing integer column, a combination of multiple columns, or a globally unique identifier (GUID). Primary keys are essential for establishing relationships between tables, as they are referenced by foreign keys in related tables.

**Foreign Key:** A foreign key is a column or a set of columns in a database table that establishes a link or a relationship between that table and another table in the same database. The foreign key in one table refers to the primary key or a unique key in another table, creating a logical association between the data in the two tables.

Foreign keys are used to enforce referential integrity, ensuring that data stored in related tables remains consistent and accurate. They prevent actions that would create orphaned records or violate the relationships defined between tables. When a foreign key is defined in a table, it typically restricts the values that can be inserted into the foreign key column to those values that exist in the corresponding primary key column of the referenced table.

Foreign keys play a crucial role in defining relationships between tables in a relational database model. They enable the establishment of one-to-one, one-to-many, or many-to-many relationships between tables, allowing for the efficient retrieval and manipulation of related data across multiple tables within the database.

**6. .what is alternate and candidate key?**

**Candidate Key:** A candidate key is a set of one or more columns in a relational database table that can uniquely identify each record or row within that table. Unlike a primary key, which is chosen as the main identifier for a table, candidate keys are all potential choices for the primary key. Each candidate key must satisfy the following conditions:

Uniqueness: Each candidate key must uniquely identify each record in the table.

Irreducibility: No subset of the candidate key should be able to uniquely identify records; removing any column from the candidate key would result in a loss of uniqueness.

While a table can have multiple candidate keys, one of them is selected as the primary key. The other candidate keys, which are not chosen as the primary key, are known as alternate keys.

**Alternate Key:** An alternate key, also known as a secondary key, is any candidate key that is not chosen as the primary key for a table. In other words, an alternate key is any candidate key that is not used to uniquely identify records in the table. Although only one candidate key can be designated as the primary key, the remaining candidate keys still have unique constraints applied to them, ensuring that they also uniquely identify records.

While the primary key serves as the main identifier for a table, alternate keys provide additional options for uniquely identifying records if needed. They play a role in maintaining data integrity and ensuring that each record in the table remains uniquely identifiable, even if the primary key changes or is unavailable.

**7. What are window functions?**

Perform calculations across a set of rows related to the current row within a query result set. Unlike aggregate functions like SUM() or AVG(), which collapse multiple rows into a single result, window functions operate on a "window" of rows defined by a specific partition or ordering.

Key characteristics of window functions include:

Operate on Window of Rows: Window functions perform calculations across a set of rows defined by a window frame, which can be specified using the OVER() clause. This window frame can be based on a partition (group of rows) or an ordering (sequential arrangement of rows).

Return Individual Result for Each Row: Window functions return a value for each row in the result set, rather than collapsing the rows into a single result as aggregate functions do. Each row's calculation is based on the data in its window frame.

Flexible Usage: Window functions can be used with various analytical calculations, such as ranking, aggregation, moving averages, cumulative sums, and more. They provide powerful capabilities for analyzing and manipulating data within a query.

**Common window functions include:**

ROW\_NUMBER(): Assigns a unique sequential integer to each row based on a specified ordering.

RANK(): Assigns a rank to each row based on a specified ordering, with ties resulting in the same rank and leaving gaps between ranks.

DENSE\_RANK(): Similar to RANK(), but assigns consecutive ranks without gaps, even when there are ties.

NTILE(n): Divides the rows into n equally-sized buckets and assigns a bucket number to each row.

SUM(), AVG(), MAX(), MIN(): Aggregate functions can be used as window functions when combined with the OVER() clause to calculate aggregates over a window of rows.

Window functions are commonly used in analytical queries to gain insights into data distributions, trends, and patterns. They offer a powerful and flexible toolset for performing complex calculations within SQL queries.

**8. Explain Ranking Functions? Given a small table , write the output.**

Ranking functions in SQL are used to assign a rank to each row in a result set based on a specified ordering. There are several ranking functions available, including ROW\_NUMBER(), RANK(), and DENSE\_RANK().

Here's a brief explanation of each:

ROW\_NUMBER(): Assigns a unique sequential integer to each row based on the specified ordering. If there are ties, each tied row receives a distinct row number.

RANK(): Assigns a unique rank to each row based on the specified ordering. If there are ties, rows with the same value receive the same rank, and the next rank is skipped. For example, if two rows tie for the first position, both will receive a rank of 1, and the next rank will be 3.

DENSE\_RANK(): Similar to RANK(), but assigns consecutive ranks without gaps, even when there are ties. If there are ties, rows with the same value receive the same rank, and the next rank is the next consecutive integer.

Here's a small example table and the output for each ranking function:

ID Name Score

1 Alice 80

2 Bob 90

3 Charlie 80

4 David 95

**ROW\_NUMBER():**

ID Name Score Row\_Number

1 Alice 80 1

2 Bob 90 2

3 Charlie 80 3

4 David 95 4

**RANK():**

ID Name Score Rank

1 Alice 80 1

2 Bob 90 2

3 Charlie 80 1

4 David 95 4

**DENSE\_RANK ():**

ID Name Score Dense\_Rank

1 Alice 80 1

2 Bob 90 2

3 Charlie 80 1

4 David 95 3

These examples demonstrate how each ranking function assigns ranks to the rows based on the specified ordering (in this case, the score).

**9. Types of Joins? With example and usecase. All the number of records return and exact records.**

Types of Joins:

INNER JOIN: Returns records that have matching values in both tables based on the specified join condition.

Syntax:

SELECT \* FROM table1 INNER JOIN table2 ON table1.id = table2.id;

Use case: Suppose you have two tables, orders and customers. You want to retrieve all orders along with their corresponding customer information where there's a match between the customer\_id column in the orders table and the id column in the customers table.

LEFT JOIN (or LEFT OUTER JOIN):

Returns all records from the left table and matching records from the right table. If there's no match, it returns NULL values for the columns from the right table.

Syntax:

SELECT \* FROM table1 LEFT JOIN table2 ON table1.id = table2.id;

Use case: You want to retrieve all employees along with their corresponding department information. Even if some employees don't belong to any department, you still want to include them in the result set.

RIGHT JOIN (or RIGHT OUTER JOIN):

Returns all records from the right table and matching records from the left table. If there's no match, it returns NULL values for the columns from the left table.

Syntax: SELECT \* FROM table1 RIGHT JOIN table2 ON table1.id = table2.id;

Use case: Similar to LEFT JOIN, but you want to ensure all records from the right table are included in the result set, even if there are no matches in the left table.

FULL JOIN (or FULL OUTER JOIN):

Returns all records when there is a match in either left or right table. If there's no match, it returns NULL values for the columns from the opposite table.

Syntax:

SELECT \* FROM table1 FULL JOIN table2 ON table1.id = table2.id;

Use case: You want to retrieve all students along with their corresponding course information. You want to include students who are enrolled in courses and courses that have enrolled students, regardless of whether there's a match between students and courses.

CROSS JOIN:

Returns the Cartesian product of the two tables, i.e., all possible combinations of rows from both tables.

SELECT \* FROM table1 CROSS JOIN table2;

Use case: You want to generate a list of all possible combinations of products and categories. Each product should be paired with each category.

For the number of records returned and exact records, it's essential to consider the actual data in the tables and the join conditions used in the query. The specific records returned and the number of records in the result set can vary based on the relationships between the tables and the data in them. Each join type behaves differently and has different implications for the result set.

**10. Use case when self-join is required.**

A self-join is required when you need to join a table with itself. One common use case for a self-join is when dealing with hierarchical data stored within a single table.

Example Use Case: Managing Hierarchical Data

Consider a scenario where you have an "employees" table in a company database. Each employee has a unique ID (employee\_id) and a manager ID (manager\_id) indicating who their direct manager is. The manager\_id is a foreign key that references another employee's employee\_id.

In this scenario, you might need to perform a self-join to retrieve information about employees and their managers. For example, you may want to generate a report that lists each employee along with their manager's details.

Here's how you can use a self-join to achieve this:

SELECT

e.employee\_id,

e.employee\_name,

m.employee\_id AS manager\_id,

m.employee\_name AS manager\_name

FROM

employees e

LEFT JOIN

employees m ON e.manager\_id = m.employee\_id;

In this query:

We're selecting columns from the employees table, referring to it twice as e (for employees) and m (for managers).

We're using a LEFT JOIN to join the employees table to itself based on the manager\_id relationship.

We're selecting the employee's ID and name (employee\_id and employee\_name), as well as their manager's ID and name (manager\_id and manager\_name).

This query will return a list of employees with details about their managers. It allows you to navigate the hierarchical structure of employees reporting to managers within the same table.

In summary, a self-join is required when you need to establish relationships between records within the same table, such as in hierarchical data structures like organizational charts, bill of materials, network graphs, or any other scenario where records have parent-child relationships within the same entity.

**11. What is subquery?**

A subquery, also known as a nested query or inner query, is a SQL query that is embedded within another SQL query. Subqueries are enclosed within parentheses and can appear in various parts of a SQL statement, including the SELECT, FROM, WHERE, HAVING, and INSERT INTO clauses.

Types of Subqueries:

Scalar Subquery: Returns a single value.

Row Subquery: Returns a single row of values.

Column Subquery: Returns a single column of values.

Table Subquery: Returns multiple rows and columns.

Purpose of Subqueries:

Filtering Data: Subqueries are commonly used in the WHERE clause to filter rows based on specific criteria. For example, you might use a subquery to find all employees who earn more than the average salary.

Performing Calculations: Subqueries can be used to perform calculations and return derived values. For instance, you might use a subquery to calculate the total sales for each product category.

Nested Conditions: Subqueries can be nested within other subqueries, allowing for complex conditions and logic. This is particularly useful for handling multiple levels of filtering or comparison.

Subquery in INSERT, UPDATE, DELETE: Subqueries can also be used in INSERT, UPDATE, and DELETE statements to manipulate data based on the results of another query.

Syntax:

The basic syntax of a subquery is as follows:

SELECT column1, column2, ...

FROM table\_name

WHERE column1 = (SELECT column1 FROM another\_table WHERE condition);

Example:

Consider the following scenario where you have two tables, "employees" and "departments." You want to find all employees who work in the Sales department.

SELECT employee\_id, employee\_name

FROM employees

WHERE department\_id = (SELECT department\_id FROM departments WHERE department\_name = 'Sales');

In this example:

The subquery (SELECT department\_id FROM departments WHERE department\_name = 'Sales') returns the department ID for the Sales department.

This department ID is then used in the main query's WHERE clause to filter the employees who work in the Sales department.

Conclusion:

Subqueries provide a powerful tool for writing complex SQL queries and performing various operations based on the results of other queries. They allow for dynamic filtering, calculations, and conditional logic, making them a fundamental aspect of SQL programming.

12. What is corelated subquery?

A correlated subquery is a type of subquery in SQL where the inner query references columns from the outer query. Unlike a non-correlated subquery, which can be executed independently of the outer query, a correlated subquery is dependent on the outer query for its execution.

Here's a detailed explanation of correlated subqueries:

Structure of a Correlated Subquery:

A correlated subquery consists of an inner query (subquery) and an outer query.

The inner query references columns from the tables in the outer query.

The inner query is executed repeatedly for each row processed by the outer query.

The result of the inner query is used to determine the result of the outer query.

Execution of a Correlated Subquery:

The outer query retrieves a row from the main table.

For each row fetched by the outer query, the inner query is executed.

The inner query uses values from the current row of the outer query to filter or calculate results.

The result of the inner query is used in the outer query's condition or calculation.

Steps 1-4 are repeated for each row processed by the outer query.

Use Cases of Correlated Subqueries:

Filtering Results: Correlated subqueries can be used to filter rows in the outer query based on conditions calculated in the inner query.

SELECT employee\_name

FROM employees e

WHERE EXISTS (

SELECT 1

FROM salaries s

WHERE s.employee\_id = e.employee\_id

AND s.salary > 50000

);

Calculating Aggregates: Correlated subqueries can be used to calculate aggregates for each row in the outer query.

SELECT employee\_name,

(SELECT AVG(salary)

FROM salaries s

WHERE s.employee\_id = e.employee\_id) AS avg\_salary

FROM employees e;

Advantages of Correlated Subqueries:

Flexibility: Correlated subqueries allow complex logic and conditions based on values from the outer query.

Granular Control: They provide granular control over filtering and calculations at the row level.

Limitations of Correlated Subqueries:

Performance: Correlated subqueries can be less efficient compared to non-correlated subqueries, especially for large datasets.

Complexity: They can make the SQL query harder to read and understand, particularly when used with multiple levels of nesting.

In summary, correlated subqueries are a powerful tool in SQL that allows for dynamic filtering and calculations based on values from the outer query. While they offer flexibility and granular control, they should be used judiciously considering their potential impact on performance and query complexity.

**13. What is CTE?**

A Common Table Expression (CTE) is a temporary named result set that can be defined within the scope of a SQL query. It allows you to create a named subquery that can be referenced within the main query, making complex queries more readable, modular, and easier to maintain.

Here's a detailed explanation of CTEs:

Syntax:

A CTE is defined using the WITH keyword followed by the CTE name and its definition. The CTE definition includes a SELECT statement that produces the result set for the CTE.

WITH cte\_name (column1, column2, ...) AS

(

SELECT column1, column2, ...

FROM table\_name

WHERE conditions

)

Main Characteristics:

Readability: CTEs improve the readability of SQL queries by breaking them down into smaller, logical parts.

Modularity: They allow you to define reusable subqueries that can be referenced multiple times within a query or across multiple queries.

Scope: CTEs are scoped to the query in which they are defined, meaning they are only accessible within that specific query.

Usage:

WITH RecursiveCTE (column1, column2, ...)

AS

( -- Anchor member

SELECT initial\_columns

FROM initial\_table

UNION ALL

-- Recursive member

SELECT recursive\_columns

FROM RecursiveCTE

WHERE condition )

Data Transformation: CTEs can be used for data transformation tasks, such as aggregations, filtering, or joining multiple tables.

WITH AggregatedData AS (

SELECT category, SUM(sales\_amount) AS total\_sales

FROM sales

GROUP BY category

)

SELECT \* FROM AggregatedData WHERE total\_sales > 1000;

Code Reusability: They can simplify complex queries by breaking them down into smaller, more manageable parts, improving code maintainability and reusability.

Advantages:

Improved Readability: CTEs make complex queries easier to understand by organizing them into smaller, logical units.

Code Reusability: They allow you to reuse subqueries within the same query or across multiple queries.

Simplified Recursive Queries: CTEs provide a structured way to write recursive queries without using temporary tables or stored procedures.

Limitations:

Scope: CTEs are only accessible within the query in which they are defined and cannot be referenced in subsequent queries.

Performance: While CTEs can improve code readability, they may not always offer the best performance compared to alternative query structures.

In summary, CTEs are a powerful SQL feature that enhances the readability, modularity, and maintainability of queries, especially in cases involving recursive queries, data transformation tasks, and complex analytical operations. They provide a structured and efficient way to break down complex queries into smaller, more manageable parts.

**14. What is derived table?**

A derived table, also known as an inline view or subquery, is a temporary result set that is created within the scope of a SQL query and exists only for the duration of that query. It is similar to a view in that it allows you to define a virtual table based on the result of a subquery, but unlike a view, a derived table does not persist in the database schema and is used only within the context of the query in which it is defined.

Here's a detailed explanation of derived tables:

Structure of a Derived Table:

• A derived table is defined within the FROM clause of a SQL query.

• It consists of a subquery enclosed within parentheses and aliased with a table name.

• The subquery can be any valid SQL query that produces a result set, including SELECT, JOIN, UNION, and other clauses.

• The result of the subquery serves as the data source for the derived table.

Syntax:

SELECT column1, column2, ... FROM (subquery) AS alias;

Example: Suppose we have two tables, orders and customers, and we want to retrieve orders along with the corresponding customer details. We can achieve this using a derived table as follows:

SELECT o.order\_id, o.order\_date, c.customer\_name FROM orders o INNER JOIN (SELECT customer\_id, customer\_name FROM customers) AS c ON o.customer\_id = c.customer\_id;

In this example:

• The subquery (SELECT customer\_id, customer\_name FROM customers) retrieves the customer\_id and customer name from the customers table.

• The result of this subquery is aliased as c and used as a derived table within the main query.

• The main query selects order\_id and order\_date from the orders table and joins with the derived table c on customer\_id to retrieve the customer name.

Advantages of Derived Tables:

1. Simplicity: Derived tables allow you to encapsulate complex logic or calculations within the query itself, making the query easier to understand and maintain.

2. Flexibility: They enable you to use the result of a subquery as a virtual table, which can then be further manipulated or joined with other tables in the query.

3. Performance: Derived tables can sometimes improve query performance by allowing the database optimizer to optimize the execution plan based on the subquery's result set.

Limitations of Derived Tables:

1. Readability: Overuse of derived tables can make the query harder to read and understand, especially if there are multiple levels of nesting.

2. Performance: In some cases, using derived tables may lead to suboptimal query performance, particularly if the subquery is complex or returns a large result set.

In summary, derived tables are a useful feature in SQL that allows you to create temporary result sets within a query, providing flexibility and simplicity in query construction. However, they should be used judiciously to maintain query readability and performance.

**15. Find third highest employee based on salary?**

To find the third highest employee based on salary, you can use a SQL query with the ROW\_NUMBER() window function. Here's how you can do it:

SELECT employee\_name, salary

FROM (

SELECT employee\_name, salary, ROW\_NUMBER() OVER (ORDER BY salary DESC) AS rn

FROM employees

) AS ranked\_employees

WHERE rn = 3;

In this query:

We use a subquery to assign row numbers to each employee based on their salary, ordered in descending order (ORDER BY salary DESC).

We use the ROW\_NUMBER() window function to generate row numbers.

We select only those rows where the row number is equal to 3, indicating the third highest salary.

This query will return the name and salary of the employee with the third highest salary. If there are multiple employees with the same salary, the query will return one of them arbitrarily. Adjust the ORDER BY clause as needed if you have specific requirements for handling ties.Top of Form

**16. .Find third highest employee based on salary per department?**

To find the third highest employee based on salary within each department, you can use a Common Table Expression (CTE) along with the ROW\_NUMBER() window function partitioned by the department. Here's how you can do it:

WITH ranked\_employees AS (

SELECT

employee\_name, salary, department\_id,

ROW\_NUMBER() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rn

FROM employees

)

SELECT

employee\_name, salary, department\_id

FROM ranked\_employees

WHERE rn = 3;

In this query:

We first create a Common Table Expression (CTE) named ranked\_employees to assign row numbers to each employee within each department based on their salary, ordered in descending order within each department (ORDER BY salary DESC).

We use the ROW\_NUMBER () window function partitioned by the department\_id to generate row numbers within each department.

We select only those rows where the row number is equal to 3, indicating the third highest salary within each department.

This query will return the name, salary, and department ID of the employee with the third highest salary within each department. Adjust the ORDER BY clause as needed if you have specific requirements for handling ties within each department.

**17. .How to find duplicate values in a single column?**

To find duplicate values in a single column in SQL, you can use a GROUP BY clause along with the HAVING clause to filter the results. Here's how you can do it:

SELECT column\_name

FROM table\_name

GROUP BY column\_name

HAVING COUNT(\*) > 1;

Replace column\_name with the name of the column you want to check for duplicates, and table\_name with the name of the table containing that column.

Explanation:

The GROUP BY clause groups the rows based on the values in the specified column.

The HAVING clause filters the grouped rows to include only those where the COUNT(\*) (i.e., the number of occurrences) is greater than 1, indicating duplicates.

This query will return the values in the specified column that appear more than once in the table. If you want to retrieve the count of duplicates as well, you can modify the query accordingly.

18. How to find duplicate values in a multiple column?

SELECT column1, column2, ..., columnN

FROM table\_name

GROUP BY column1, column2, ..., columnN

HAVING COUNT(\*) > 1;

Replace column1, column2, ..., columnN with the names of the columns you want to check for duplicates, and table\_name with the name of the table containing those columns.

Explanation:

The GROUP BY clause groups the rows based on the values in the specified columns.

The HAVING clause filters the grouped rows to include only those where the COUNT(\*) (i.e., the number of occurrences) is greater than 1, indicating duplicates across the specified columns.

This query will return rows where the combination of values across the specified columns appears more than once in the table, indicating duplicates. Adjust the column names and table name according to your specific scenario.

**19. What are ACID properties? give example for each property**

ACID (Atomicity, Consistency, Isolation, Durability) properties are a set of properties that ensure the reliability and integrity of transactions in a database system.

Here's an explanation of each property with an example:

Atomicity:

Atomicity ensures that a transaction is treated as a single unit of work, either fully completed or fully aborted.

If any part of the transaction fails, the entire transaction is rolled back to its original state.

Example: Consider a bank transfer where money is withdrawn from one account and deposited into another. Atomicity ensures that if the withdrawal succeeds but the deposit fails (or vice versa), the entire transfer is aborted, and the money remains unchanged in both accounts.

Consistency:

Consistency ensures that a transaction brings the database from one valid state to another.

Each transaction must preserve the integrity constraints and invariants of the database.

Example: In a banking system, if a withdrawal transaction reduces an account balance, the resulting balance must still adhere to the constraints (e.g., non-negative balance) defined for the account.

Isolation:

Isolation ensures that the concurrent execution of multiple transactions does not interfere with each other.

Each transaction appears to execute in isolation from others, as if it were the only transaction in the system.

Example: Suppose two transactions are withdrawing money from the same account concurrently. Isolation ensures that the first transaction's changes are not visible to the second transaction until the first transaction commits or rolls back, preventing dirty reads, non-repeatable reads, and phantom reads.

Durability:

Durability ensures that once a transaction is committed, its changes are permanent and survive system failures.

Committed transactions persist even in the event of power loss, crashes, or other failures.

Example: After a user transfers money between accounts and receives a confirmation message, the transaction's changes are durably stored in the database. Even if the system crashes immediately after, the transferred amount remains intact and can be retrieved upon system recovery.

These ACID properties collectively ensure the reliability, consistency, and integrity of transactions in a database system, supporting robust data management and preserving data correctness.

**20**. **Diff between union and union all**

UNION:

The UNION operator is used to combine the results of two or more SELECT statements and returns only distinct rows.

Duplicate rows are automatically removed from the combined result set.

The number of columns and their data types must be the same in all SELECT statements.

The result set is sorted in ascending order by default.

Example:

SELECT column1, column2 FROM table1

UNION

SELECT column1, column2 FROM table2;

UNION ALL:

The UNION ALL operator also combines the results of two or more SELECT statements but returns all rows, including duplicates.

Duplicate rows are not removed; they are retained in the combined result set.

The number of columns and their data types must be the same in all SELECT statements.

The result set is not sorted; it preserves the order of rows from each SELECT statement.

Example:

SELECT column1, column2 FROM table1

UNION ALL

SELECT column1, column2 FROM table2;

In summary, UNION removes duplicate rows from the combined result set, while UNION ALL retains all rows, including duplicates. Use UNION when you want to eliminate duplicates, and UNION ALL when you want to include duplicates in the result set.

**21. Diff between primary key and unique key**

In SQL, both primary keys and unique keys are used to enforce uniqueness constraints on columns in a table, but they have some differences:

**Primary Key:**

A primary key is a column or a set of columns that uniquely identifies each row in a table.

There can be only one primary key constraint defined per table.

Primary key columns do not allow NULL values.

By default, a primary key also creates a clustered index in most database management systems (DBMS).

Used to establish relationships between tables (foreign key constraints).

Automatically creates a unique index to enforce uniqueness and to speed up data retrieval.

Typically used to define the primary means of accessing data in a table.

**Unique Key:**

A unique key is a column or a set of columns that ensures that all values in the column(s) are unique.

There can be multiple unique key constraints defined per table.

Unique key columns allow NULL values, but only one NULL value is allowed (unless the column is part of a composite unique key).

Does not automatically create a clustered index.

Used to enforce uniqueness but does not define the primary means of accessing data.

Can be used to define alternate keys in a table, aside from the primary key.

In summary, a primary key uniquely identifies each row in a table and is typically used as the primary means of accessing data, while a unique key ensures uniqueness but does not necessarily define the primary access mechanism. Additionally, primary key columns do not allow NULL values, whereas unique key columns allow NULL values with certain restrictions.

**22. Diff between truncate and delete**

The main differences between TRUNCATE and DELETE in SQL are as follows:

Operation:

**TRUNCATE** is a DDL (Data Definition Language) operation.

**DELETE** is a DML (Data Manipulation Language) operation.

Functionality:

**TRUNCATE** removes all rows from a table, effectively resetting the table to its original state.

**DELETE** removes specific rows from a table based on a condition or deletes all rows if no condition is specified.

Transaction Logging:

**TRUNCATE** operations are not logged, or they are minimally logged, meaning they do not generate individual log entries for each deleted row.

**DELETE** operations are fully logged, meaning they generate a log entry for each deleted row, which can impact performance for large datasets.

Rollback:

**TRUNCATE** cannot be rolled back. Once executed, the data is permanently removed from the table.

**DELETE** can be rolled back using a ROLLBACK statement, reverting the table to its previous state before the DELETE operation.

Locking:

TRUNCATE typically obtains an exclusive table lock, preventing other users from accessing the table until the operation completes.

DELETE can use row-level locks, allowing other users to access unaffected rows during the operation.

Speed:

TRUNCATE is generally faster than DELETE, especially for large tables, because it deallocates the data pages, while DELETE removes rows one by one.

In summary, TRUNCATE is a faster and more efficient way to remove all rows from a table, but it is less flexible and cannot be rolled back. DELETE, on the other hand, allows for selective row deletion and can be rolled back, but it is slower and generates more transaction log entries. Use TRUNCATE when you need to quickly remove all rows from a table and DELETE when you need more control over the deletion process or when you need to rollback the operation.

**23. Diff between having and where**

The **WHERE** clause is used to filter rows from the result set before any grouping is performed.

The **HAVING** clause is used to filter rows from the result set after the grouping has been performed.

**Application**:

The **WHERE** clause is applied to individual rows in the table before they are grouped.

The **HAVING** clause is applied to groups of rows after they have been grouped using the GROUP BY clause.

**Aggregates**:

The **WHERE** clause cannot be used with aggregate functions.

The **HAVING** clause is specifically designed to filter rows based on aggregate functions, such as COUNT, SUM, AVG, etc.

**Usage with Grouping:**

The **WHERE** clause is used with individual rows and does not require grouping.

The **HAVING** clause is used with grouped rows and requires the GROUP BY clause.

**Syntax**:

The **WHERE** clause is typically used before the GROUP BY clause in a query.

The **HAVING** clause is used after the GROUP BY clause in a query.

In summary, the WHERE clause is used to filter individual rows before any grouping is performed, while the HAVING clause is used to filter grouped rows based on aggregate functions after the grouping has been performed.

**24. SQL query execution order.**

Tables specified in the FROM clause are accessed.

Joins or subqueries are performed to generate the initial result set.

WHERE clause:

Rows from the result set are filtered based on the conditions specified in the WHERE clause.

Rows that satisfy the conditions are retained, while those that do not are discarded.

GROUP BY clause:

If specified, the result set is then grouped into sets of rows based on the columns specified in the GROUP BY clause.

Rows with the same values in the specified columns are grouped together.

HAVING clause:

If specified, grouped rows are filtered based on the conditions specified in the HAVING clause.

Groups that satisfy the conditions are retained, while those that do not are discarded.

SELECT clause:

Columns specified in the SELECT clause are evaluated for each row or group of rows in the result set.

Expressions, functions, and aliases specified in the SELECT clause are also processed at this stage.

DISTINCT clause:

If specified, duplicate rows are removed from the result set to produce a unique set of rows.

ORDER BY clause:

If specified, rows in the result set are then sorted based on the columns specified in the ORDER BY clause.

LIMIT/OFFSET clause:

If specified, the result set may be limited to a certain number of rows or offset by a certain number of rows.

It's important to note that not all queries will include every clause, and the actual execution order may vary based on the specific components of the query. Additionally, some database engines may optimize the execution order based on factors such as indexes, statistics, and query complexity.

**25. What are indexes? Types of Indexes and their differences.**

In databases, indexes are data structures that improve the speed of data retrieval operations by providing quick access to specific rows in a table. They work like the index section of a book, allowing the database engine to quickly locate the desired data without having to scan the entire table.

Types of Indexes:

**Primary Index:**

A primary index is created automatically when a primary key constraint is defined on a column or a set of columns.

It uniquely identifies each row in the table and enforces uniqueness.

Typically implemented as a clustered index, where the physical order of rows in the table is based on the index key.

**Unique Index:**

A unique index ensures that all values in the indexed column(s) are unique.

Unlike primary indexes, unique indexes allow NULL values (except for columns included in the primary key constraint).

Can be created on one or multiple columns.

Does not necessarily define the primary access mechanism to the table.

Clustered Index:

A clustered index physically orders the rows in the table based on the indexed column(s).

There can be only one clustered index per table because the physical order of rows can only be based on one key.

Typically used with primary keys, but can also be created separately.

Non-Clustered Index:

A non-clustered index creates a separate index structure from the table data.

Does not affect the physical order of rows in the table.

Can be created on one or multiple columns.

Allows for faster retrieval of data based on the indexed columns.

Composite Index:

A composite index is created on multiple columns.

Useful for queries that involve multiple columns in the WHERE clause or for covering queries.

Improves query performance for multi-column queries.

Bitmap Index:

A bitmap index stores bitmaps for each possible value in the indexed column(s).

Suitable for low-cardinality columns with few distinct values.

Provides fast performance for equality and range queries.

Differences:

Primary Index vs. Unique Index: Primary index is typically used to enforce the primary key constraint and uniquely identify rows, while a unique index enforces uniqueness but does not necessarily define the primary access mechanism.

Clustered Index vs. Non-Clustered Index: Clustered index physically orders the rows in the table, while non-clustered index creates a separate index structure.

Composite Index vs. Bitmap Index: Composite index is created on multiple columns, while a bitmap index stores bitmaps for each possible value in the indexed column(s).

In summary, indexes are essential for improving the performance of data retrieval operations, and the choice of index type depends on the specific requirements of the database schema and the types of queries being executed.

**26. What is surrogate key? Give example where you used it and how**

A surrogate key is an artificial primary key used to uniquely identify each row in a database table. Unlike natural keys, which are based on real-world attributes of the data (such as a person's social security number or a product's serial number), surrogate keys have no inherent meaning and are typically generated automatically by the database system.

Here's an example of using a surrogate key:

Let's say you have a table named "Employees" to store information about employees in a company. Each employee is uniquely identified by an employee ID, but you also want to ensure that the employee ID remains stable even if other attributes (such as name or department) change over time. In this case, you can introduce a surrogate key as the primary key of the table.

Here's how you might implement it:

CREATE TABLE Employees (

employee\_id INT PRIMARY KEY,

surrogate\_key INT AUTO\_INCREMENT,

name VARCHAR(50),

department VARCHAR(50)

);

In this example:

employee\_id is the natural key, representing the unique identifier for each employee.

surrogate key is the surrogate key, an artificially created identifier generated automatically by the database system.

The AUTO\_INCREMENT attribute ensures that each new record in the table is assigned a unique value for the surrogate key, incrementing by 1 for each new record.

By using a surrogate key in this way, you ensure that the database has a stable, unique identifier for each record, regardless of changes to other attributes. This can simplify database operations and provide better performance for queries and joins. Additionally, it helps maintain data integrity and consistency within the database.