**SQL Interview Question**

**Answers:**

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

**SQL is a domain-specific language used for managing and manipulating relational databases. SQL commands can be categorised into several types based on their purpose and usage. Here are some common types of SQL commands with examples:**

1. **Data Querying Commands**

**These commands are used to retrieve data from databases.**

**/\*Retrieve all columns and rows from table\*/**

1. **Data Modification Commands**

**These commands are used to insert, delete or update data in the database.**

Examples:

* INSERT INTO table\_name (column1, column2) VALUES (value1, value2);: Insert a new row into a table.
* UPDATE table\_name SET column1 = value1 WHERE condition;: Update existing data in a table.
* DELETE FROM table\_name WHERE condition;: Delete rows from a table based on specified conditions.

**2. What is** Normalisation **and denormalization?**

Normalisation and denormalization are two opposing database design techniques used in SQL to organise and structure data efficiently within relational databases. They serve different purposes and have specific use cases:

Normalisation:

Normalisation is the process of organising data in a database to reduce data redundancy and improve data integrity. It involves breaking down large tables into smaller, related tables and establishing relationships between them. Normalisation follows a set of rules called normal forms, including First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), and so on.

The primary goals of normalisation are:

* Minimise data redundancy: Reducing duplicated data ensures consistency and saves storage space.
* Maintain data integrity: Avoiding update anomalies (such as insert, update, or delete anomalies) helps maintain the accuracy and consistency of data.

Here's a simple example of normalisation:

Suppose you have a denormalized table called CustomerOrders with columns for both customer and order information. Normalisation would involve breaking this into two separate tables, Customers and Orders, connected by a foreign key.

Denormalization:

Denormalization, on the other hand, is the process of intentionally introducing redundancy into a database design to improve query performance. In some situations, normalised tables can lead to complex and slow queries, especially when dealing with joins across multiple tables. Denormalization simplifies querying by reducing the need for joins and aggregations.

The primary goals of denormalization are:

* Improve query performance: By reducing the number of joins and simplifying data retrieval.
* Reduce the complexity of queries: Easier-to-understand queries can lead to faster development and maintenance.

Here's a simple example of denormalization:

Suppose you have a normalised database with separate tables for Customers and Orders. To improve query performance, you might create a denormalized view or table that combines customer and order information into a single table. This can be particularly useful for reporting or analytical purposes.

In summary, normalisation and denormalization are two opposite techniques used in SQL database design. Normalisation aims to minimise data redundancy and maintain data integrity, while denormalization focuses on improving query performance by introducing some level of data redundancy. The choice between the two depends on the specific requirements of the application and the trade-offs between storage efficiency and query performance.

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

First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF) are stages of normalisation in SQL database design. These normal forms are used to ensure that a database schema is well-structured, minimises redundancy, and maintains data integrity. Each normal form has specific rules that tables must meet to achieve that level of normalisation:

* First Normal Form (1NF):
  + A table is in 1NF if it meets the following criteria:
    - All columns must contain atomic (indivisible) values. This means that a column should not contain arrays, nested records, or other non-atomic types.
    - Each column in a table must have a unique name.
    - The order in which data is stored does not matter.
* Achieving 1NF ensures that data is stored in a tabular format without repeating groups of data. It is the most basic level of normalisation.  
  Example: Consider a table called StudentCourses that combines student names and their enrolled courses. To bring it to 1NF, you would create separate tables for students and courses, eliminating the repeating data.
* Second Normal Form (2NF):
  + A table is in 2NF if it meets the following criteria:
    - It is in 1NF.
    - It has a primary key (a unique identifier), and all non-key columns are fully functionally dependent on the entire primary key.
* In simple terms, 2NF eliminates partial dependencies, where some non-key columns depend on only a part of the primary key.  
  Example: Consider a table Orders with columns (order\_id, product\_id, quantity), where (order\_id, product\_id) is the composite primary key. If quantity depends only on product\_id and not on the entire primary key, you would split the table into two: one for orders and another for order details.
* Third Normal Form (3NF):
  + A table is in 3NF if it meets the following criteria:
    - It is in 2NF.
    - It has no transitive dependencies. In other words, non-key columns should not depend on other non-key columns.
* 3NF eliminates dependencies between non-key columns, ensuring that each column depends only on the primary key.  
  Example: Consider a table Employees with columns (employee\_id, department\_id, manager\_id). If manager\_id depends on employee\_id, which in turn depends on department\_id, you would split the table into three separate tables: Employees, Departments, and Managers.

Achieving these normal forms helps ensure that a database schema is organised efficiently, minimises data redundancy, and reduces the risk of data anomalies, making it easier to maintain and query the database. Depending on the complexity of the data and the specific use case, higher normal forms like Boyce-Codd Normal Form (BCNF) and Fourth Normal Form (4NF) may also be considered for further refinement of the schema.

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

Use Case: Social Media Feed

Consider a social media platform like Facebook or Twitter, which handles a massive amount of user-generated content, such as posts, comments, likes, and shares. In the database design for such a platform, data is typically normalised to ensure data consistency and reduce redundancy. You might have tables like Users, Posts, Comments, Likes, and Shares.

In this scenario, normalising the data means keeping each type of interaction (post, comment, like, share) in separate tables and using foreign keys to establish relationships. While this approach ensures data integrity, it can lead to complex and slow queries when generating a user's personalised feed. The feed typically contains a mixture of posts, comments on those posts, likes on posts, and shares of posts.

To improve query performance and provide a more responsive user experience, denormalization can be employed:

* Create a Denormalized Feed Table: You can create a denormalized table called UserFeed that contains a user's personalised feed data. This table might include columns like user\_id, post\_id, post\_content, comment\_id, comment\_content, like\_id, share\_id, and timestamps for sorting.
* Update the Denormalized Table: Regularly update the UserFeed table to reflect the user's activity and the activity of their connections. This can be done using triggers, batch processes, or real-time updates. As new posts, comments, likes, and shares occur, relevant entries are added to the UserFeed table.
* Query the Denormalized Table: When a user accesses their feed, you can query the UserFeed table directly, fetching all the necessary data for display. This eliminates the need for complex joins and aggregations, making feed retrieval faster and more efficient.
* Maintain Data Consistency: You need to ensure that updates to the normalised data (e.g., editing a post or deleting a comment) are correctly reflected in the denormalized UserFeed table to maintain data consistency.

While denormalization improves query performance for feed generation, it does introduce some level of data redundancy and requires careful synchronisation with the normalised data. Additionally, considerations for data retention, privacy, and scalability need to be taken into account. Nonetheless, denormalization is a common technique in social media platforms and other systems where personalised, real-time data retrieval is essential for user engagement.

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

In SQL, a primary key and a foreign key are two important concepts used to establish and maintain relationships between tables in a relational database. They play a fundamental role in ensuring data integrity and organizing data within the database. Here's what each of these keys means in SQL:

Primary Key:

* A primary key is a column or a combination of columns in a database table that uniquely identifies each row (record) in that table.
* It enforces the entity integrity constraint, ensuring that each row has a unique identifier within the table.
* Primary keys are used to uniquely identify records and establish relationships with other tables through foreign keys.
* By default, primary keys are indexed, which means that they can be used to efficiently search for specific rows in the table.
* Common data types used for primary keys include integers, GUIDs, and strings, among others.

Example:

sql

Copy code

CREATE TABLE Students (

student\_id INT PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

date\_of\_birth DATE

);

In this example, student\_id is the primary key, and it ensures that each student in the Students table has a unique identifier.

Foreign Key:

* A foreign key is a column or a combination of columns in one table that establishes a link to the primary key in another table.
* It enforces referential integrity, ensuring that the values in the foreign key column(s) match the values in the primary key column(s) of the referenced table.
* Foreign keys create relationships between tables, allowing you to maintain data consistency and integrity.
* They are used to implement one-to-many or many-to-one relationships between tables.
* Foreign keys are used to define relationships between tables and enable cascading actions, such as updates and deletes, when referenced data is modified.

Example:

sql

Copy code

CREATE TABLE Orders (

order\_id INT PRIMARY KEY,

customer\_id INT,

order\_date DATE,

FOREIGN KEY (customer\_id) REFERENCES Customers(customer\_id)

);

In this example, customer\_id in the Orders table is a foreign key that references the customer\_id primary key in the Customers table. This establishes a relationship between orders and customers, ensuring that each order is associated with a valid customer.

In summary, primary keys are used to uniquely identify rows within a table, while foreign keys establish relationships between tables by linking to the primary key of another table. Together, they play a vital role in maintaining data integrity and structuring relational databases effectively.

**6.** Whatis an alternate **and candidate key?**

In SQL, candidate keys and alternate keys are concepts related to the design and modelling of database tables, specifically when it comes to enforcing uniqueness constraints. Here's how these concepts are defined in SQL:

Candidate Key:

* A candidate key in SQL is a set of one or more columns in a database table that can uniquely identify each row (record) within that table.
* Each candidate key satisfies the uniqueness constraint, meaning that the values in the key columns must be unique across all rows in the table.
* A table can have multiple candidate keys.
* Typically, one of the candidate keys is chosen as the primary key of the table, and the others are considered alternate keys.

Alternate Key:

* An alternate key in SQL is a candidate key that is not selected as the primary key for the table.
* While it is not designated as the primary key, an alternate key is still used to enforce the uniqueness constraint on the table.
* Alternate keys provide an alternative means of identifying rows uniquely, apart from the primary key.
* They are valuable for ensuring data integrity and for enabling unique constraints on columns or sets of columns that are not primary keys.

Example: Let's say you have a table called Employees, which contains employee information. In this table, you have the following columns: employee\_id, email, and employee\_code.

* If employee\_id is chosen as the primary key because it uniquely identifies each employee, then both email and employee\_code become alternate keys. These columns can still be used to enforce uniqueness constraints within the table, even though they are not the primary means of identifying employees.

In SQL, you can enforce uniqueness constraints using indexes or unique constraints on the columns that make up candidate and alternate keys. This ensures that duplicate values are not allowed in those columns, maintaining data integrity within the database.

In summary, candidate keys are sets of columns that can uniquely identify rows in a SQL table, and one of them is typically chosen as the primary key. Alternate keys are candidate keys that are not designated as the primary key but are still used to enforce uniqueness constraints within the table.

**7. What are window functions?**

Window functions, also known as windowing or analytic functions, are a category of SQL functions that perform calculations across a set of rows that are related to the current row within a result set. Unlike traditional aggregate functions like SUM or AVG, which aggregate rows into a single value, window functions allow you to perform calculations on a "window" of rows around each row in the result set. These functions are particularly useful for tasks such as ranking, calculating running totals, and finding percentiles within a dataset.

Key characteristics of window functions:

* Operate on a Window of Rows: Window functions work on a specified window or subset of rows within the result set. This window is defined based on an ordering or partitioning criteria.
* No Grouping of Rows: Unlike aggregate functions like GROUP BY, window functions do not group rows but maintain individual rows while performing calculations.
* Return Values for Each Row: Window functions return a value for each row in the result set, and this value is calculated based on the window of rows related to that row.
* Syntax: Window functions are typically used in conjunction with the OVER clause, which specifies the windowing criteria, including the partitioning and ordering of rows.

Commonly used window functions in SQL include:

* ROW\_NUMBER(): Assigns a unique integer to each row within a window, typically used for ranking rows.
* RANK(): Similar to ROW\_NUMBER(), but assigns the same rank to rows with equal values, leaving gaps in the ranking when there are ties.
* DENSE\_RANK(): Similar to RANK(), but assigns the same rank to rows with equal values without leaving gaps in the ranking when there are ties.
* LEAD() and LAG(): Retrieve values from the next or previous row within the window, respectively.
* SUM(), AVG(), MIN(), MAX(): Calculate cumulative or running aggregates within the window.
* FIRST\_VALUE() and LAST\_VALUE(): Retrieve the first or last value within the window, respectively.

Here's an example of using the ROW\_NUMBER() window function to assign row numbers to rows in a table:

sql

Copy code

SELECT

employee\_name,

department,

salary,

ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row\_num

FROM

employees;

In this example, ROW\_NUMBER() assigns a unique row number to each employee within their department, based on their salary in descending order. The PARTITION BY clause defines the partitioning of rows by department, and the ORDER BY clause specifies the ordering criteria within each partition.

Window functions are a powerful tool for performing complex calculations and analyses within SQL queries while maintaining the granularity of individual rows. They are supported in most modern relational database management systems, including PostgreSQL, SQL Server, Oracle, and MySQL.

**8. Explain Ranking Functions?**

Ranking functions in SQL are a category of window functions that assign a ranking or ordinal number to each row within a result set based on specified criteria. These functions are particularly useful when you need to identify the relative position or rank of rows in a dataset, such as finding the top performers, ranking products by sales, or ordering students by exam scores.

Common ranking functions in SQL include:

* ROW\_NUMBER():
  + The ROW\_NUMBER() function assigns a unique integer value to each row within a result set.
  + Rows with the same values in the ordering criteria will receive different row numbers.
  + This function is suitable for ranking rows without considering ties, resulting in gaps in the ranking when there are ties in the data.
* Example:
* sql
* Copy code

SELECT

employee\_name,

department,

salary,

ROW\_NUMBER() OVER (ORDER BY salary DESC) AS row\_num

FROM

employees;

* RANK():
  + The RANK() function assigns a rank to each row within a result set, taking into account rows with equal values in the ordering criteria.
  + Rows with equal values receive the same rank, and the next rank is skipped.
  + This function is suitable when you want to assign the same rank to rows with equal values.
* Example:
* sql
* Copy code

SELECT

product\_name,

sales\_amount,

RANK() OVER (ORDER BY sales\_amount DESC) AS sales\_rank

FROM

products;

* DENSE\_RANK():
  + The DENSE\_RANK() function assigns a rank to each row within a result set, considering rows with equal values in the ordering criteria.
  + Rows with equal values receive the same rank, and there are no gaps in the ranking.
  + This function is useful when you want to assign consecutive ranks without skipping any.
* Example:
* sql
* Copy code

SELECT

student\_name,

exam\_score,

DENSE\_RANK() OVER (ORDER BY exam\_score DESC) AS rank\_in\_class

FROM

exam\_results;

Ranking functions typically use the OVER clause, which specifies the windowing criteria, including the ordering of rows. You can partition the result set using the PARTITION BY clause to reset the ranking for each partition, allowing you to rank rows within groups.

Ranking functions are widely used for various analytical and reporting tasks, such as finding top performers, identifying outliers, and ranking products or competitors by performance. They are particularly valuable in situations where you need to assess the relative position of data points within a dataset.

**9. Types of Joins?**

In SQL, there are several types of joins that allow you to combine rows from two or more tables based on a related column between them. Each type of join serves a specific purpose and produces different results. Here are the most common types of joins in SQL:

* INNER JOIN:
  + An INNER JOIN retrieves rows from both tables that have matching values in the specified columns.
  + Rows from either table that do not have a match in the other table are excluded from the result set.
* Example:
* sql
* Copy code

SELECT employees.employee\_id, employees.first\_name, departments.department\_name

FROM employees

INNER JOIN departments ON employees.department\_id = departments.department\_id;

* LEFT JOIN (or LEFT OUTER JOIN):
  + A LEFT JOIN returns all rows from the left table and the matching rows from the right table. If there is no match, NULL values are returned for columns from the right table.
  + The left table's rows that have no matching rows in the right table still appear in the result set.
* Example:
* sql
* Copy code

SELECT customers.customer\_id, orders.order\_id

FROM customers

LEFT JOIN orders ON customers.customer\_id = orders.customer\_id;

* RIGHT JOIN (or RIGHT OUTER JOIN):
  + A RIGHT JOIN is similar to a LEFT JOIN but returns all rows from the right table and the matching rows from the left table.
  + Rows from the right table that do not have a match in the left table result in NULL values for columns from the left table.
  + This type of join is less commonly used than LEFT JOIN.
* Example:
* sql
* Copy code

SELECT orders.order\_id, order\_details.product\_id

FROM orders

RIGHT JOIN order\_details ON orders.order\_id = order\_details.order\_id;

* FULL JOIN (or FULL OUTER JOIN):
  + A FULL JOIN returns all rows when there is a match in either the left or the right table.
  + Rows from both tables that do not have a match in the other table result in NULL values for columns from the non-matching table.
  + FULL JOIN is less common and is typically used when you want to include all data from both tables.
* Example:
* sql
* Copy code

SELECT customers.customer\_id, orders.order\_id

FROM customers

FULL JOIN orders ON customers.customer\_id = orders.customer\_id;

* CROSS JOIN (or Cartesian Join):
  + A CROSS JOIN returns the Cartesian product of two tables, resulting in every possible combination of rows from both tables.
  + This type of join can produce a very large result set, so it should be used with caution.
* Example:
* sql
* Copy code

SELECT employees.first\_name, departments.department\_name

FROM employees

CROSS JOIN departments;

* SELF JOIN:
  + A SELF JOIN is a join in which a table is joined with itself. It is useful when you have hierarchical or recursive data within a single table.
  + It involves using table aliases to distinguish between the two instances of the same table.
* Example:
* sql
* Copy code

SELECT e1.employee\_id, e1.first\_name, e2.manager\_id

FROM employees e1

LEFT JOIN employees e2 ON e1.employee\_id = e2.manager\_id;

These are the primary types of joins in SQL, each serving different purposes for combining data from multiple tables based on relationships between them. The choice of which join to use depends on the specific requirements of your query and the desired results.

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

A self join in SQL is used when you want to join a table with itself. This may seem a bit unusual at first, but it becomes necessary in various scenarios, especially when dealing with hierarchical or recursive data structures within a single table. Here are a few common use cases for self joins:

* Employee Hierarchy:
  + One of the most common use cases for self joins is to represent organizational hierarchies, such as an employee reporting structure.
  + In this scenario, a table of employees may have a foreign key that references the employee's manager by their employee ID.
  + A self join can be used to retrieve information about employees and their respective managers.
* Example:
* sql
* Copy code

SELECT e1.employee\_name AS employee, e2.employee\_name AS manager

FROM employees e1

LEFT JOIN employees e2 ON e1.manager\_id = e2.employee\_id;

* Bill of Materials (BOM):
  + In manufacturing or product management databases, a self join can be used to represent the hierarchical structure of a product's components.
  + Each row represents a component, and a self join is used to find the subcomponents or materials required for a larger assembly.
* Example:
* sql
* Copy code

SELECT a.assembly\_name, b.component\_name

FROM products a

JOIN products b ON a.product\_id = b.parent\_product\_id;

* Comments and Replies:
  + In a discussion forum or comment system, where comments can have replies, a self join can be used to represent the relationship between comments and their replies.
  + This allows you to retrieve comment threads or nested replies.
* Example:
* sql
* Copy code

SELECT c1.comment\_text, c2.reply\_text

FROM comments c1

LEFT JOIN comments c2 ON c1.comment\_id = c2.parent\_comment\_id;

* File Systems:
  + When modelling file systems, each file or directory may have a parent directory. A self join can be used to traverse the file hierarchy and retrieve the structure.
* Example:
* sql
* Copy code

SELECT f1.file\_name, f2.directory\_name

FROM files f1

LEFT JOIN directories f2 ON f1.parent\_directory\_id = f2.directory\_id;

* User Friendships or Relationships:
  + In social networking applications, you may want to represent user connections or friendships.
  + A self join can be used to find connections between users within the same table.
* Example:
* sql
* Copy code

SELECT u1.username AS user, u2.username AS friend

FROM users u1

LEFT JOIN friendships f ON u1.user\_id = f.user\_id

LEFT JOIN users u2 ON f.friend\_id = u2.user\_id;

These are just a few examples of when a self join can be useful. Self joins are versatile and help represent hierarchical or recursive relationships within a single table, allowing you to query and analyse such data structures effectively.

**11. What** is a subquery**?**

A subquery, also known as a nested query or inner query, is a query that is embedded within another SQL query. Subqueries are used to retrieve data that will be used as a part of the main query's condition, filter, or result. They are a powerful tool in SQL for performing complex operations and making queries more dynamic and flexible.

Here are some key points about subqueries in SQL:

* Embedded Query: A subquery is a query statement that is enclosed within parentheses and placed inside another SQL query, typically within a SELECT, FROM, WHERE, or HAVING clause.
* Usage:
  + In the SELECT clause: Subqueries can be used to retrieve a single value that becomes part of the result set, for example, to calculate a column based on the result of the subquery.
  + In the FROM clause: Subqueries can be used as a derived table, allowing you to treat the result of the subquery as a temporary table to be used in the main query.
  + In the WHERE clause: Subqueries can be used to filter rows based on conditions that involve data from another table.
  + In the HAVING clause: Subqueries can be used to filter the result of a grouped query.
* Purpose:
  + Subqueries are used to break down complex problems into smaller, more manageable parts.
  + They allow you to perform operations on one set of data based on values or conditions derived from another set of data.
  + Subqueries are often used for tasks like filtering rows, performing calculations, checking existence, and retrieving related data.
* Types:
  + Scalar Subqueries: Subqueries that return a single value and can be used wherever a single value is expected, such as in the SELECT clause.
  + Row Subqueries: Subqueries that return multiple rows and are typically used in comparison operators like IN, ANY, or ALL.
  + Table Subqueries: Subqueries that return a result set that can be treated as a table and used in the FROM clause.

Here's an example of a subquery in SQL:

sql

Copy code

SELECT employee\_name, salary

FROM employees

WHERE department\_id = (

SELECT department\_id

FROM departments

WHERE department\_name = 'Engineering'

);

In this example, the subquery retrieves the department\_id for the 'Engineering' department from the departments table. The main query then uses this value to filter rows in the employees table, selecting only employees in the Engineering department.

Subqueries add flexibility to SQL queries, enabling you to perform complex operations and make queries more dynamic by using the results of one query as input to another.

**12. What** is a correlated **subquery?**

A correlated subquery is a type of subquery in SQL where the inner query (the subquery) references columns from the outer query. Unlike a regular (non-correlated) subquery, which is executed independently, a correlated subquery is executed for each row processed by the outer query. This means that the subquery's results depend on the values of the current row being processed in the outer query.

Key characteristics of correlated subqueries:

* Reference to Outer Query: In a correlated subquery, columns from the outer query are referenced within the subquery's WHERE or HAVING clause. This creates a relationship between the inner and outer queries.
* Execution for Each Row: The subquery is executed once for each row returned by the outer query. The result of the subquery is calculated based on the specific values of the current row in the outer query.
* Dynamic Behavior: Because a correlated subquery depends on the context of the outer query, it provides a dynamic way to filter or calculate results based on the data in the current row.
* Typical Use Cases: Correlated subqueries are often used when you need to perform row-level comparisons, filtering, or calculations based on related data within the same or other tables.

Here's a simple example of a correlated subquery:

sql

Copy code

SELECT employee\_name, salary

FROM employees e1

WHERE salary > (

SELECT AVG(salary)

FROM employees e2

WHERE e2.department\_id = e1.department\_id

);

In this query, the correlated subquery calculates the average salary (AVG(salary)) for employees in the same department as the current row being processed in the outer query (e1.department\_id). The outer query then retrieves the names and salaries of employees whose salary is higher than the department's average salary.

It's important to note that correlated subqueries can be computationally expensive, especially when dealing with large datasets, because they involve executing the subquery multiple times. Therefore, they should be used judiciously, and alternatives like JOIN operations or window functions should be considered when possible to improve query performance.

Despite their potential performance considerations, correlated subqueries are a powerful tool in SQL for solving complex problems that require context-aware comparisons or calculations based on related data within a dataset.

**13. What is CTE?**

A Common Table Expression (CTE) is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement in SQL. CTEs provide a way to create a named, structured, and reusable query that simplifies complex queries and makes them more readable. CTEs are especially useful when you need to perform recursive queries or when you want to break down a complex query into smaller, more manageable parts.

Key characteristics and usage of CTEs in SQL:

* Definition: A CTE is defined using the WITH keyword, followed by a query enclosed within parentheses. The CTE is given a name, and you can reference this name in subsequent queries.
* Reusability: CTEs allow you to define a query once and use it multiple times within a larger query or across different queries, promoting code reuse and maintainability.
* Recursive Queries: CTEs are commonly used for recursive queries, where a query references its own output to build a hierarchical or interconnected result set.
* Improved Readability: CTEs make SQL queries more readable by breaking them into smaller, self-contained parts, each with a meaningful name.
* Scoped to a Query: CTEs are scoped to the query in which they are defined. They cannot be referenced outside of the query where they are declared.

Here's a basic example of a CTE:

sql

Copy code

WITH SalesCTE AS (

SELECT

employee\_id,

SUM(sales\_amount) AS total\_sales

FROM

sales

GROUP BY

employee\_id

)

SELECT

employee\_id,

total\_sales

FROM

SalesCTE

WHERE

total\_sales > 10000;

In this example, the CTE named SalesCTE calculates the total sales for each employee by grouping the sales data. The main query then selects employees with total sales greater than $10,000 from the CTE.

CTEs are particularly useful in scenarios such as hierarchical data (e.g., organisational structures), recursive operations (e.g., finding all descendants of a node in a hierarchical tree), and when you want to simplify complex queries by breaking them into more manageable parts. They are supported by most modern relational database management systems, including PostgreSQL, SQL Server, Oracle, and MySQL.

**14. What** is a derived **table?**

In SQL, a derived table (also known as an inline view or subselect) is a temporary result set that is created within a SQL query. Derived tables are typically defined within the FROM clause of a query and are used to simplify complex queries or to encapsulate a portion of a query for readability and maintainability. They are especially useful when you need to perform calculations, filtering, or transformations on data before using it in the main query.

Key characteristics and usage of derived tables in SQL:

* Definition in the FROM Clause: Derived tables are defined within the FROM clause of a query. They are treated as if they were regular tables, but they exist only for the duration of the query.
* Encapsulation: Derived tables allow you to encapsulate a portion of a query, making it more self-contained and easier to read. This can be especially helpful for complex calculations or transformations.
* Reusability: While derived tables cannot be referenced outside the query in which they are defined, you can use the same derived table multiple times within the same query.
* Dynamic Data: Derived tables allow you to create dynamic datasets based on the data in your database, which can be particularly useful for generating intermediate results.
* Aliases: Derived tables are typically given aliases (table names) within the query, making it easier to reference them in the main query.

Here's an example of a derived table in SQL:

sql

Copy code

SELECT

customer\_name,

order\_count

FROM

(

SELECT

customer\_id,

COUNT(order\_id) AS order\_count

FROM

orders

GROUP BY

customer\_id

) AS CustomerOrders

WHERE

order\_count >= 5;

In this example, a derived table named CustomerOrders is created to calculate the count of orders (order\_count) for each customer. The main query then selects customer names and their respective order counts from the derived table, filtering for customers with five or more orders.

Derived tables are a valuable tool for improving query readability, encapsulating logic, and breaking down complex queries into more manageable parts. They are widely supported in SQL databases and can be used to simplify queries while maintaining the power and flexibility of SQL.

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

To find the third-highest employee based on salary in SQL, you can use a subquery with the LIMIT or TOP clause (depending on your specific database system) to select the top three salaries and then retrieve the employee(s) associated with the third-highest salary. Here's a generic SQL query that should work in most relational databases:

sql

Copy code

SELECT employee\_id, employee\_name, salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 2;

In this query:

* We select the employee\_id, employee\_name, and salary columns from the employees table.
* We use ORDER BY salary DESC to sort the rows in descending order based on the salary column, which will place the employees with the highest salaries at the top.
* We use LIMIT 1 OFFSET 2 to retrieve only one row (the third-highest salary) starting from the third row. The OFFSET 2 skips the first two rows, which contain the highest and second-highest salaries.

Please note that the specific syntax for limiting and offsetting rows may vary depending on the database system you are using. For example:

* In MySQL, you can use LIMIT 1 OFFSET 2.
* In PostgreSQL, you can also use LIMIT 1 OFFSET 2.
* In SQL Server, you can use TOP 1 with a subquery and the ROW\_NUMBER() function to achieve the same result.
* In Oracle, you can use the ROWNUM pseudocolumn with a subquery to achieve the same result.

Be sure to adapt the query to the syntax and features supported by your particular database system.

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

To find the third-highest employee based on salary per department in SQL, you can use a common table expression (CTE) and the ROW\_NUMBER() window function to rank employees within each department by salary. Then, you can filter the results to only include rows where the rank is 3. Here's a SQL query that accomplishes this task:

sql

Copy code

WITH RankedEmployees AS (

SELECT

employee\_id,

employee\_name,

department\_id,

salary,

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

FROM

employees

)

SELECT

employee\_id,

employee\_name,

department\_id,

salary

FROM

RankedEmployees

WHERE

salary\_rank = 3;

In this query:

* We create a CTE called RankedEmployees, which calculates the rank of each employee's salary within their department using the ROW\_NUMBER() window function. The PARTITION BY clause ensures that ranking is done separately for each department, and the ORDER BY salary DESC sorts salaries in descending order.
* Then, in the main query, we select the employee\_id, employee\_name, department\_id, and salary columns from the RankedEmployees CTE.
* We filter the results to include only rows where the salary\_rank is equal to 3, which represents the third-highest salary within each department.

This query will provide you with a list of employees who have the third-highest salary in their respective departments. Adjust the column names and table names to match your actual database schema.

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

To find duplicate values in a single column in SQL, you can use a SELECT query with a GROUP BY clause and a HAVING clause. Here's a SQL query that will help you identify duplicate values in a specific column:

sql

Copy code

SELECT column\_name, COUNT(\*) AS count\_of\_duplicates

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 where the column is located.

Here's an explanation of how this query works:

* The GROUP BY clause groups the rows in the table based on the values in the specified column.
* The COUNT(\*) function is used to count the number of rows within each group (i.e., for each distinct value in the column).
* The HAVING clause filters the groups to include only those where the count of rows (i.e., the count of duplicates) is greater than 1. This effectively identifies duplicate values in the column.

When you execute this query, it will return a list of the duplicate values in the specified column along with the count of occurrences for each duplicate value.

For example, if you have a table named employees with a column named email, and you want to find duplicate email addresses, you would use the following query:

sql

Copy code

SELECT email, COUNT(\*) AS count\_of\_duplicates

FROM employees

GROUP BY email

HAVING COUNT(\*) > 1;

This query will provide a list of email addresses that appear more than once in the employees table.

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

To find duplicate values across multiple columns in SQL, you can use the combination of the GROUP BY clause and the HAVING clause. You'll group the rows based on the combination of columns you want to check for duplicates and then filter the groups to include only those where the count of rows (i.e., the count of duplicates) is greater than 1. Here's a SQL query that demonstrates how to do this:

sql

Copy code

SELECT column1, column2, ..., columnN, COUNT(\*) AS count\_of\_duplicates

FROM table\_name

GROUP BY column1, column2, ..., columnN

HAVING COUNT(\*) > 1;

Here's an explanation of how this query works:

* Replace column1, column2, ..., columnN with the names of the columns you want to check for duplicates.
* Replace table\_name with the name of the table where the columns are located.

This query will group the rows based on the combination of columns you specified, count the number of rows within each group (i.e., count of duplicates), and filter the groups to include only those where the count is greater than 1. The result will be a list of rows with duplicate values across the specified columns.

Here's an example: Let's say you have a table named employees with columns first\_name, last\_name, and email, and you want to find rows where both the first name and last name are the same:

sql

Copy code

SELECT first\_name, last\_name, COUNT(\*) AS count\_of\_duplicates

FROM employees

GROUP BY first\_name, last\_name

HAVING COUNT(\*) > 1;

This query will identify rows with duplicate combinations of first names and last names in the employees table. Adjust the column names and table name as needed for your specific use case.

**19. What are ACID properties?**

ACID is an acronym that stands for Atomicity, Consistency, Isolation, and Durability. These properties are a set of characteristics that guarantee the reliability and consistency of database transactions in SQL and other database management systems. They ensure that transactions are executed reliably even in the presence of hardware failures, software bugs, or other unexpected events.

Here's an overview of each of the ACID properties:

* Atomicity:
  + Atomicity ensures that a transaction is treated as a single, indivisible unit of work.
  + Either all the changes made by the transaction are committed to the database, or none of them are.
  + If any part of the transaction fails (e.g., due to an error or exception), all changes made by the transaction are rolled back (undone), leaving the database in its previous state.
* Consistency:
  + Consistency ensures that a transaction brings the database from one consistent state to another.
  + The database must satisfy a set of integrity constraints before and after a transaction.
  + If a transaction violates any integrity constraint, it is rolled back, and no changes are applied to the database.
* Isolation:
  + Isolation ensures that concurrent transactions do not interfere with each other.
  + Each transaction should operate as if it is the only transaction executing, even when multiple transactions are running concurrently.
  + Isolation levels (e.g., Read Uncommitted, Read Committed, Repeatable Read, Serializable) specify the level of isolation provided by the database system.
* Durability:
  + Durability ensures that once a transaction is committed, its changes are permanent and will survive any subsequent failures, including system crashes.
  + Committed changes are stored in non-volatile memory (e.g., disk) and are not lost, even if the database system crashes.

These ACID properties are crucial for maintaining data integrity, reliability, and consistency in database systems, especially in scenarios where multiple users or applications access and modify data concurrently. ACID-compliant database management systems (DBMS) provide mechanisms and safeguards to ensure that transactions meet these properties, making them suitable for critical applications where data accuracy and reliability are essential.

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

UNION and UNION ALL are both used to combine the result sets of two or more SELECT queries in SQL, but they differ in how they handle duplicate rows. Here are the key differences between UNION and UNION ALL:

* Duplicate Rows:
  + UNION: UNION eliminates duplicate rows from the combined result set. It returns only distinct rows across all the SELECT queries.
  + UNION ALL: UNION ALL includes all rows from the combined result set, including duplicates. It does not remove duplicate rows; instead, it simply combines the result sets of the SELECT queries.
* Performance:
  + UNION ALL is generally faster than UNION because it doesn't have to perform the additional work of identifying and removing duplicate rows.
* Result Size:
  + Because UNION removes duplicates, it may result in a smaller combined result set compared to UNION ALL, which retains all rows.
* Use Cases:
  + Use UNION when you want to combine result sets and remove duplicate rows to get a list of distinct values. For example, when querying a table of customers and contacts, you might want a list of unique names.
  + Use UNION ALL when you want to combine result sets and keep all rows, including duplicates. This is useful when you need to concatenate data from different sources or need to preserve duplicate entries for some reason.

Examples:

sql

Copy code

-- Using UNION to remove duplicates

SELECT employee\_id FROM employees

UNION

SELECT manager\_id FROM employees;

-- Using UNION ALL to keep duplicates

SELECT employee\_id FROM employees

UNION ALL

SELECT manager\_id FROM employees;

In the first example, UNION would return a list of unique employee IDs, removing duplicates. In the second example, UNION ALL would return all employee and manager IDs, including any duplicates. The choice between UNION and UNION ALL depends on your specific requirements for the result set.

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

In SQL, both the primary key and unique key are used to enforce the uniqueness of values in one or more columns of a table, but they serve slightly different purposes and have distinct characteristics. Here are the key differences between a primary key and a unique key:

Primary Key:

* Uniqueness: A primary key enforces both uniqueness and the absence of NULL values in the specified column(s). Each row must have a unique value in the primary key column(s).
* Null Values: A primary key column(s) cannot contain NULL values. Every row must have a non-null value in the primary key column(s).
* Number of Columns: A table can have only one primary key, which can consist of one or more columns. This is known as a composite primary key.
* Clustered Index (In Some Databases): In some database systems like SQL Server, a primary key also defines the clustered index for the table. The clustered index determines the physical order of data rows in the table.
* Use Case: Primary keys are typically used to uniquely identify each row in a table and establish relationships with other tables (e.g., in foreign keys).

Unique Key:

* Uniqueness: A unique key enforces uniqueness in the specified column(s), similar to a primary key. Each row must have a unique value in the unique key column(s).
* Null Values: A unique key column(s) can contain NULL values. However, if a column contains a NULL value, it doesn't violate the uniqueness constraint, but it allows multiple NULL values.
* Number of Columns: A table can have multiple unique keys, each consisting of one or more columns.
* Clustered Index: A unique key does not define the clustered index for the table. In most database systems, you need to explicitly specify which index should be clustered, and it's not necessarily related to unique keys.
* Use Case: Unique keys are used when you need to ensure the uniqueness of values in a column or combination of columns but don't require the additional constraint of non-null values in the column(s). They are often used to prevent duplicate values in non-primary key columns.

In summary, while both primary keys and unique keys enforce uniqueness, primary keys have additional constraints, such as the absence of NULL values and the limitation of one primary key per table. Unique keys allow for NULL values and support multiple unique constraints in a single table. The choice between them depends on the specific requirements of your data and the constraints you need to enforce.

**22. Diff between truncate and delete**

In SQL, both the TRUNCATE and DELETE statements are used to remove data from a table, but they work differently and have distinct characteristics. Here are the key differences between TRUNCATE and DELETE:

TRUNCATE:

* Operation:
  + TRUNCATE is a DDL (Data Definition Language) statement used to remove all rows from a table quickly.
  + It is not used to selectively delete specific rows but instead empties the entire table.
* Locking:
  + TRUNCATE typically uses less locking than DELETE, making it more efficient for removing all rows from a table.
  + It acquires an exclusive table-level lock briefly during the operation, but it does not lock individual rows, allowing other transactions to access the table simultaneously.
* Rollback:
  + TRUNCATE is non-transactional in most database systems, meaning it cannot be rolled back. Once you execute TRUNCATE, the data is permanently removed from the table.
* Identity Columns:
  + In some database systems, TRUNCATE may reset identity (auto-increment) columns to their initial seed value.
* Performance:
  + TRUNCATE is typically faster than DELETE when you need to remove all rows from a table because it is a minimal logged operation and doesn't generate individual delete statements for each row.

DELETE:

* Operation:
  + DELETE is a DML (Data Manipulation Language) statement used to remove specific rows from a table based on a condition or criteria.
  + It allows for selective deletion of rows that meet specific criteria.
* Locking:
  + DELETE can use row-level locking, which means it locks the rows that are being deleted. This can lead to more significant locking and potential blocking issues in a multi-user environment.
* Rollback:
  + DELETE is transactional and can be rolled back if executed within a transaction. You can undo the effects of a DELETE statement by rolling back the transaction.
* Identity Columns:
  + DELETE does not reset identity (auto-increment) columns. If you delete rows, the identity column values remain unchanged.
* Performance:
  + DELETE can be slower than TRUNCATE, especially when deleting a large number of rows, as it generates individual delete statements and logs each deleted row.

In summary, TRUNCATE is suitable for quickly removing all rows from a table when you want to reset the table's data, and it is more efficient and less logged than DELETE. On the other hand, DELETE is used when you need to selectively remove specific rows based on conditions, and it is a transactional operation that can be rolled back. The choice between them depends on your specific use case and requirements.

**23. SQL query execution order.**

In SQL, the execution order of a query is determined by the logical query processing order, which defines how various clauses and operations within a query are evaluated. Here's the typical logical query processing order in SQL:

* FROM: The first step in query execution is to identify the source tables and views in the FROM clause. This clause specifies the tables from which data will be retrieved.
* JOIN: If the query involves multiple tables and includes JOIN operations, these joins are performed next. Joins combine rows from different tables based on specified join conditions.
* WHERE: After the tables are joined, the WHERE clause is used to filter rows. Rows that do not meet the conditions specified in the WHERE clause are eliminated from further consideration.
* GROUP BY: If the query includes a GROUP BY clause, rows are grouped into sets based on the specified grouping columns. Aggregate functions (e.g., SUM, COUNT, AVG) can be applied to each group.
* HAVING: If the query includes a HAVING clause (which is used in combination with GROUP BY), it filters groups based on specified conditions. Groups that do not meet the conditions are eliminated.
* SELECT: After filtering and grouping, the SELECT clause specifies which columns to include in the result set. Expressions and calculations defined in the SELECT clause are also performed at this stage.
* DISTINCT: If the query includes the DISTINCT keyword, duplicate rows are eliminated from the result set. This step occurs after the SELECT clause.
* ORDER BY: If an ORDER BY clause is present, the result set is sorted based on the specified columns and sort order.
* LIMIT/OFFSET (Optional): In some database systems, you can use LIMIT and OFFSET clauses to limit the number of rows returned or to skip a certain number of rows from the beginning of the result set.
* Final Result: The final result set is returned to the user or application based on the previous steps' processing.

It's important to note that not all queries include every clause listed above, and the order can vary depending on the query's structure and requirements. Additionally, some database systems may optimise query execution by reordering operations or using various optimization techniques.

Understanding the logical query processing order is essential for writing efficient and correct SQL queries, especially when dealing with complex queries involving multiple tables and operations.

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

Indexes in SQL are database structures that improve the retrieval speed of rows from database tables. They work like an index in a book, allowing the database engine to quickly locate rows of data based on the indexed columns. Indexes are critical for optimising query performance, especially in tables with large amounts of data. There are different types of indexes in SQL, each with its own characteristics and use cases. Here are some common types of indexes and their differences:

* Single-Column Index:
  + A single-column index is created on a single column of a table.
  + It speeds up queries that involve filtering or sorting by the indexed column.
  + Commonly used index types for single-column indexing include B-tree indexes and hash indexes.
* Composite Index (Multi-Column Index):
  + A composite index is created on multiple columns of a table.
  + It is used to optimize queries that involve filtering or sorting based on multiple columns in the same query.
  + The order of columns in a composite index matters; it affects how the index is used for different query conditions.
* Unique Index:
  + A unique index enforces the uniqueness constraint on one or more columns, preventing duplicate values in the indexed column(s).
  + It is used to ensure data integrity for columns that should contain unique values.
  + Attempting to insert a duplicate value into a unique indexed column will result in an error.
* Primary Key Index:
  + A primary key index is a type of unique index that enforces the primary key constraint on one or more columns.
  + It ensures that the indexed column(s) contain unique values and do not contain NULL values.
  + A table can have only one primary key.
* Clustered Index:
  + A clustered index determines the physical order of data rows in a table.
  + In some database systems (e.g., SQL Server, MySQL InnoDB), the table's data rows are stored in the order of the clustered index.
  + A table can have only one clustered index.
* Non-Clustered Index:
  + A non-clustered index is a separate structure from the data rows and does not affect the physical order of data.
  + It speeds up queries by providing a quick lookup mechanism to locate rows based on indexed columns.
  + A table can have multiple non-clustered indexes.
* Full-Text Index:
  + A full-text index is used for efficient text-based searches in columns that contain large amounts of textual data (e.g., articles, documents).
  + It enables features like keyword searches and ranking of search results based on relevance.
* Bitmap Index:
  + A bitmap index uses a bitmap for each possible value in the indexed column(s).
  + It is particularly useful for columns with low cardinality (few distinct values).
  + Bitmap indexes are efficient for certain types of queries, such as data warehousing and analytics.

Differences between these index types primarily include their use cases, enforcement of uniqueness, physical storage implications, and how they impact query performance. The choice of which type of index to use depends on the specific requirements of your database and the types of queries you need to optimise.