SQL Interview Questions

1. What is SQL?
SQL (Structured Query Language) is a programming language designed for managing relational databases. It provides a standard way to interact with and manipulate databases, allowing users to create, retrieve, update, and delete data stored in a relational database management system (RDBMS). SQL is widely used in the field of data management and is supported by most relational database systems such as MySQL, Oracle, PostgreSQL, Microsoft SQL Server, and SQLite. It serves as a powerful tool for storing, organizing, and retrieving large amounts of structured data efficiently.

2. What is a database?

A database is an organized collection of structured information or data that is stored and managed on a computer system. It is designed to efficiently store, retrieve, and manipulate large amounts of data. Databases are widely used in various applications, ranging from simple personal record-keeping to complex enterprise systems. A database typically consists of one or more tables, which are organized into rows (also known as records or tuples) and columns (also known as fields). Each row represents a distinct entity or record, while each column represents a specific attribute or characteristic of the entities. The tables in a database are related to each other through keys, which establish relationships between the data. Databases provide a structured way to store and manage data, enabling efficient data retrieval and manipulation. There are various types of databases, including relational databases (such as MySQL, Oracle, and Microsoft SQL Server), NoSQL databases (such as MongoDB and Cassandra), and graph databases (such as Neo4j). Each type has its own strengths and is suitable for different types of applications and data models.

3. What is a table?

A table, in the context of databases, is a collection of related data organized in rows and columns. It is a fundamental structure used to store and represent structured data in a database system. A table consists of columns (also known as fields or attributes) and rows (also known as records or tuples). Each column represents a specific type of data, such as a name, age, or price, and each row represents a distinct set of values or a record.

Employee Table:

Employee ID	Name	Department	Salary
1	John	IT	5000
2	Sarah	HR	4500
3	Michael	Sales	6000

4. What is a view?

a view is a virtual table that is derived from one or more existing tables or views. It does not store any data on its own but presents the data from the underlying tables in a customized or filtered manner. A view acts as a logical representation of data, providing a simplified and controlled access to the underlying data. Views are created based on predefined queries that specify the desired columns, rows, and relationships from the underlying tables. Once a view is created, it can be treated and used like a regular table, allowing users to query, retrieve, and manipulate the data as if it were a physical table.

5. What is a primary key?

A primary key is a unique identifier for each record (or row) in a table. It ensures that each record within a table can be uniquely identified and distinguishes it from other records. The primary key enforces data integrity and provides a means to establish relationships with other tables. The syntax for defining a primary key varies depending on the database management system (DBMS) being used. Here's a general representation of the syntax:

```
CREATE TABLE table_name (
    column1 data_type PRIMARY KEY,
    column2 data_type,
    ...
);
```

In the above syntax: table_name is the name of the table you're creating. column1 is the name of the column that will serve as the primary key. data_type specifies the data type of the primary key column.

6. What is a foreign key?

a foreign key is a column or a set of columns in a table that establishes a link or relationship between the data in that table and the data in another table. It represents a reference to the primary key of another table, creating a relational connection between the two tables. Foreign keys are used to enforce referential integrity, ensuring that the values in the foreign key column(s) correspond to the values in the primary key column(s) of the referenced table. This relationship allows for the establishment of connections and joins between tables in a database. The syntax for defining a foreign key constraint varies depending on the database management system (DBMS) being used. Here's a general representation of the syntax:

```
CREATE TABLE table_name1 (
    column1 data_type PRIMARY KEY,
    column2 data_type,
    ...
    FOREIGN KEY (column3) REFERENCES table_name2(column4)
);
```

In the above syntax: table_name1 is the name of the table in which the foreign key is being defined. column1 and other columns represent the columns in table_name1. column3 is the column in table_name1 that will serve as the foreign key. table_name2 is the name of the table being referenced by the foreign key. column4 is the column in table_name2 that is being referenced.

7. What is a constraint?

A constraint is a rule that is applied to a table to enforce data integrity. Constraints can be used to ensure that data is valid and consistent. constraints are rules or conditions applied to columns or tables to enforce data integrity and maintain consistency in a database. They define restrictions on the data that can be inserted, updated, or deleted in a table. Here are some commonly used constraints in SQL:

a. Primary Key Constraint: Ensures the uniqueness of values in a column or a combination of columns. Each table can have only one primary key.
b. Foreign Key Constraint: Establishes a relationship between two tables based on a column or a set of columns. The values in the foreign key column(s) must correspond to the values in the primary key column(s) of the referenced table.
c. Unique Constraint: Enforces the uniqueness of values in one or more columns. Unlike primary keys, unique constraints allow null values (except in cases where a column is part of a composite unique constraint).
d. Check Constraint: Specifies a condition that the values in a column must satisfy. Allows or restricts data based on the defined condition.
e. Not Null Constraint: Ensures that a column does not contain null values.
f. Default Constraint: Specifies a default value for a column when no explicit value is provided during insertion.

These constraints can be defined when creating a table using the CREATE TABLE statement or added later using the ALTER TABLE statement. Constraints help maintain data integrity, prevent inconsistent or invalid data, and support the relationships between tables in a database.

8. What is a stored procedure?

A stored procedure is a named and pre-compiled collection of SQL statements and procedural logic that is stored in a database. It is designed to perform specific tasks or operations within the database system. Stored procedures are often used to encapsulate complex database

operations, improve performance, and enhance security. Here are a few key characteristics of stored procedures:

a. Reusability: Stored procedures can be executed multiple times and can be reused across different applications or modules within the database system.
b. Parameterization: Stored procedures can accept input parameters, allowing for dynamic and flexible execution based on different values passed to them.
c. Atomicity: Stored procedures are typically executed as a single unit of work, ensuring that all the statements within the procedure are either completed successfully or rolled back in case of an error.
d. Performance Optimization: By storing the procedure's compiled form in the database, execution time can be reduced, especially for complex operations involving multiple SQL statements.

Here's an example of a stored procedure:

```
CREATE PROCEDURE GetEmployeeByDepartment
    @DepartmentID INT
AS
BEGIN
    SELECT EmployeeID, Name, DepartmentID
    FROM Employees
    WHERE DepartmentID = @DepartmentID
END
```

In this example, the stored procedure is named "GetEmployeeByDepartment." It takes an input parameter @DepartmentID and retrieves employee information from the "Employees" table based on the provided department ID. To execute the stored procedure, you would use the EXEC or EXECUTE statement:

```
EXEC GetEmployeeByDepartment @DepartmentID = 1
```

This would execute the stored procedure and retrieve employee records from the "Employees" table where the DepartmentID is 1. Stored procedures can be more complex, involving control flow statements (e.g., IF-ELSE, WHILE loops), transaction management, error handling, and other database-specific functionalities. They provide a powerful mechanism for implementing business logic and data manipulation within the database itself.

9. What is a trigger?

A trigger is a database object in SQL that is associated with a table and automatically executes a set of actions in response to certain database events, such as insertions, updates, or deletions of data in the table. Triggers are used to enforce data integrity, implement business logic, and automate tasks within a database system. Triggers are typically defined to execute either before or after the triggering event occurs. The actions performed by triggers can include

modifying data in the same table or other related tables, validating data, logging changes, or executing additional SQL statements.
Here's an example of a trigger:

```
CREATE TRIGGER UpdateProductStock
ON Products
AFTER UPDATE
AS
BEGIN
    IF UPDATE(Quantity)
    BEGIN
        -- Perform necessary actions based on the updated Quantity column
        -- For example, update the stock level in a separate Stock table
        UPDATE Stock
        SET StockLevel = StockLevel - (SELECT Quantity FROM inserted)
        WHERE ProductID = (SELECT ProductID FROM inserted)
    END
END
```

10. What is a join?

a join is a mechanism that combines rows from two or more tables based on a related column between them. It allows you to retrieve data from multiple tables in a single query, combining information that is spread across different tables based on specified conditions. Joins are used to establish relationships between tables using common columns, enabling you to retrieve data that spans across related tables. The result of a join operation is a new table, often referred to as a result set or a derived table, which includes columns and rows from the joined tables.
There are different types of joins in SQL:

a. Inner Join: Retrieves records where there is a match between the specified columns in both tables.
b. Left Join (or Left Outer Join): Retrieves all records from the left table and the matching records from the right table.
c. Right Join (or Right Outer Join): Retrieves all records from the right table and the matching records from the left table.
d. Full Join (or Full Outer Join): Retrieves all records from both tables, combining unmatched records from each table.
e. Cross Join (or Cartesian Join): Generates the Cartesian product of both tables, resulting in all possible combinations of rows.
f. Self Join: Joins a table with itself, treating it as two separate tables, typically using different aliases.

11. What is normalization?

Normalization in SQL is the process of organizing and designing a relational database schema to eliminate redundancy and dependency issues. It aims to improve data integrity, eliminate data anomalies, and optimize database performance by structuring the tables and relationships in a systematic and efficient manner. There are different levels or forms of normalization, often referred to as normal forms. The most commonly discussed normal forms are:

a. First Normal Form (1NF): Requires that each column in a table holds only atomic values (indivisible) and there are no repeating groups or arrays of values.
b. Second Normal Form (2NF): Requires that a table is in 1NF and each non-key column is fully dependent on the entire primary key.
c. Third Normal Form (3NF): Requires that a table is in 2NF and there are no transitive dependencies between non-key columns.

Further normal forms, such as Boyce-Codd Normal Form (BCNF), Fourth Normal Form (4NF), and Fifth Normal Form (5NF), address more complex dependencies and normalization requirements. The process of normalization involves analyzing the attributes and relationships of a database schema and applying normalization rules to break down larger tables into smaller, more manageable ones. This helps to minimize data duplication, ensure data consistency, and simplify database operations.

12. What is denormalization?

Denormalization in SQL is the process of intentionally adding redundancy to a normalized database schema. It involves relaxing some of the normalization principles in order to improve query performance, simplify data retrieval, and reduce the need for complex joins. Denormalization is often employed in situations where read performance is a high priority, such as in data warehousing or reporting scenarios, where the focus is on efficiently retrieving and aggregating large amounts of data.

13. What is a transaction?

A transaction is a sequence of SQL statements that are executed as a single unit of work. Transactions can be used to ensure data consistency and to prevent data corruption.By using transactions, you can ensure the reliability and integrity of your database operations, enabling consistent and predictable results even in the presence of concurrent access and potential failures.

14. What is indexing?

Indexing in SQL is a technique used to improve the performance of database queries by creating special data structures called indexes. An index is a separate database object that contains a sorted copy of selected columns from a table, along with a pointer to the corresponding row(s) in the table. Indexes allow for faster data retrieval by providing an efficient

way to locate specific rows based on the values in the indexed columns. When a query is executed that involves the indexed columns, the database engine can utilize the index to quickly identify the relevant rows, reducing the need for a full table scan.
Benefits of indexing in SQL:

a. Improved Query Performance: Indexes can significantly speed up query execution, especially for large tables, by minimizing the number of disk reads required to locate data.
b. Efficient Data Retrieval: Indexes enable the database engine to directly access the relevant rows based on the indexed columns, rather than scanning the entire table.
c. Sorting and Ordering: Indexes can be created on columns that are frequently used for sorting or ordering results, allowing for efficient sorting operations.
d. Constraint Enforcement: Indexes can be used to enforce unique constraints, primary key constraints, and referential integrity constraints, ensuring data integrity and consistency.
e. Join Optimization: Indexes facilitate efficient joining of tables by providing a quick lookup mechanism for related rows, reducing the time and resources needed for join operations.

15. What is a subquery?

A subquery, also known as a nested query or inner query, is a query embedded within another query in SQL. It allows you to use the results of one query as a part of another query, enabling you to perform complex and advanced operations on the data. A subquery is enclosed within parentheses and can be used in various parts of a SQL statement, such as the SELECT, FROM, WHERE, or HAVING clauses. The result of the subquery is typically used as a condition or value in the outer query. Here's an example of a subquery used in a SELECT statement:

```
SELECT ProductName, UnitPrice
FROM Products
WHERE UnitPrice > (SELECT AVG(UnitPrice) FROM Products)
```

In this example, the subquery (SELECT AVG(UnitPrice) FROM Products) calculates the average unit price of all products in the "Products" table. The outer query then retrieves the product names and unit prices from the "Products" table where the unit price is greater than the calculated average.

16. What is a union?

In SQL, the UNION operator is used to combine the results of two or more SELECT statements into a single result set. It combines rows from different queries into a unified result set, removing duplicate rows by default.Here's an example to illustrate the usage of UNION: Let's say we have two tables, "Customers" and "Suppliers," with the following data:
Customers Table:

CustomerID	CustomerName
1	Customer A
2	Customer B
3	Customer C

Suppliers table:

SupplierID	SupplierName
1	Supplier X
4	Supplier Y
5	Supplier Z

We can use the UNION operator to combine the results of two SELECT statements, retrieving all distinct customer and supplier names in a single result set:

```
SELECT CustomerName FROM Customers
UNION
SELECT SupplierName FROM Suppliers
```

The result of this query would be:

CustomerName
Customer A
Customer B
Customer C
Supplier X
Supplier Y
Supplier Z

The UNION operator combines the results of the two SELECT statements, removing duplicate rows to produce a unique result set. If you want to include duplicate rows, you can use the UNION ALL operator instead of UNION. UNION is useful when you want to combine the results of multiple queries with similar column structures into a single result set. It allows you to retrieve and consolidate data from different tables or queries based on your requirements.

17. What is a case statement?
The CASE WHEN statement in SQL is a conditional statement that allows you to perform different actions based on specified conditions. It is often used in the SELECT statement to generate calculated or derived columns based on specific criteria.
Here's an example to illustrate the usage of the CASE WHEN statement: Let's say we have a table named "Employees" with the following data:

EmployeeID	FirstName	LastName	Salary
1	John	Smith	50000
2	Jane	Doe	60000
3	David	Johnson	70000

We can use the CASE WHEN statement to categorize the employees based on their salary ranges:

```
SELECT FirstName, LastName, Salary,
    CASE
        WHEN Salary < 55000 THEN 'Low'
        WHEN Salary >= 55000 AND Salary < 65000 THEN 'Medium'
        ELSE 'High'
    END AS SalaryRange
FROM Employees;
```

In this example, the CASE WHEN statement is used to determine the salary range for each employee. If the salary is less than 55000, it is categorized as 'Low'. If the salary is between 55000 and 65000 (inclusive), it is categorized as 'Medium'. Otherwise, it is categorized as 'High'.
The result of the query would be:

FirstName	LastName	Salary	SalaryRange
John	Smith	50000	Low
Jane	Doe	60000	Medium
David	Johnson	70000	High

18. What is a group by clause?

The GROUP BY clause in SQL is used to group rows based on specified columns and apply aggregate functions to each group. It is often used in conjunction with aggregate functions like COUNT, SUM, AVG, MAX, or MIN to perform calculations on grouped data.

Here's an example to illustrate the usage of the GROUP BY clause: Let's say we have a table named "Orders" that contains information about customer orders:

OrderID	CustomerID	OrderDate	TotalAmount
1	101	2022-01-01	100
2	102	2022-01-02	150
3	101	2022-01-03	200
4	103	2022-01-03	50
5	102	2022-01-04	300

We can use the GROUP BY clause to calculate the total order amount for each customer:

```
SELECT CustomerID, SUM(TotalAmount) AS TotalOrderAmount
FROM Orders
GROUP BY CustomerID;
```

In this example, we are selecting the CustomerID column and using the SUM function to calculate the total order amount for each customer. The result of the query would be:

CustomerID	TotalOrderAmount
101	300
102	450
103	50

The GROUP BY clause groups the rows based on the CustomerID column, and the SUM function calculates the total order amount for each group.

19. What is a having clause?

The HAVING clause in SQL is used to filter the results of a GROUP BY query based on conditions applied to the grouped data. It allows you to specify a condition for the groups created by the GROUP BY clause, similar to how the WHERE clause filters individual rows. Here's an example to illustrate the usage of the HAVING clause: Let's consider the "Orders" table from the previous example:

OrderID	CustomerID	OrderDate	TotalAmount
1	101	2022-01-01	100
2	102	2022-01-02	150
3	101	2022-01-03	200
4	103	2022-01-03	50
5	102	2022-01-04	300

We can use the HAVING clause to filter the groups based on a condition, such as displaying only those customers who have a total order amount greater than 200:

```
SELECT CustomerID, SUM(TotalAmount) AS TotalOrderAmount
FROM Orders
GROUP BY CustomerID
HAVING SUM(TotalAmount) > 200;
```

In this example, we are selecting the CustomerID column and using the SUM function to calculate the total order amount for each customer. The HAVING clause is then used to filter the groups, specifying that only groups with a total order amount greater than 200 should be included. The result of the query would be:

CustomerID	TotalOrderAmount
101	300
102	450

20. What is a rank function?

The RANK() function in SQL is used to assign a rank or position to each row within the result set based on a specified column's values. It is often used to analyze and compare the relative positions of rows based on certain criteria.Here's an example to illustrate the usage of the RANK() function: Consider a table named "Students" with the following data:

StudentID	StudentName	Score
1	John	80
2	Jane	90
3	Alice	75
4	Bob	90
5	Mark	85

We can use the RANK() function to assign a rank to each student based on their scores:

```
SELECT StudentName, Score, RANK() OVER (ORDER BY Score DESC) AS Rank
FROM Students;
```

In this example, we are selecting the StudentName, Score, and using the RANK() function to assign a rank to each student based on their scores. The result of the query would be:

StudentName	Score	Rank
Jane	90	1
Bob	90	1
Mark	85	3
John	80	4
Alice	75	5

The RANK() function assigns a rank to each row based on the ORDER BY clause. In this case, the students with the highest scores (90) are assigned the rank 1, while the student with the lowest score (75) is assigned the rank 5.

21. Explain lag and lead window functions with examples.

lag and lead window functions are used to access data from previous or subsequent rows within a specified window. These functions allow you to retrieve data from rows that are located before or after the current row, based on a defined ordering. Lag Window Function: The lag window function retrieves the value from a previous row within a given window. It allows you to access data from the preceding row relative to the current row. Let's consider an example to understand the lag function better. Assume we have a table named "Sales" with the following columns: "Year", "Quarter", and "Revenue." We want to calculate the revenue difference between each quarter and the previous quarter within each year. Here's an example query:

```
SELECT Year, Quarter, Revenue,
Revenue - LAG(Revenue, 1, 0) OVER (PARTITION BY Year ORDER BY Quarter) AS Revenue_Difference
FROM Sales;
```

This query uses the lag function to calculate the revenue difference between the current quarter and the previous quarter within each year. Lead Window Function: The lead window function is similar to the lag function but retrieves values from subsequent rows instead. It allows you to access data from the following row(s) relative to the current row.
Let's continue with the "Sales" table example. This time, we want to calculate the revenue difference between each quarter and the subsequent quarter within each year. Here's an example query using the lead function:

```
SELECT Year, Quarter, Revenue,
LEAD(Revenue, 1, 0) OVER (PARTITION BY Year ORDER BY Quarter) - Revenue AS Revenue_Difference
FROM Sales;
```

In this query, the lead function is used to calculate the revenue difference between the current quarter and the subsequent quarter within each year.

22. What is a temporary table?

A temporary table, also known as a temp table, is a database object that is created and used to store temporary data during the execution of a specific session or task. Temporary tables are typically used to hold intermediate results, perform complex calculations, or store temporary data that is only required for a short period of time. Here are some key points about temporary tables:

a. Scope: Temporary tables are specific to the session or connection that creates them. They are automatically dropped and destroyed when the session ends or when they are explicitly dropped by the user.
b. Structure: Temporary tables have a structure similar to regular database tables, including column definitions, data types, and constraints. They can be created and modified using standard SQL syntax.
c. Usage: Temporary tables can be used to store and manipulate data just like regular tables. They can be queried, updated, joined with other tables, and used in various database operations.
d. Performance: Temporary tables are typically stored in memory or temporary disk space, which can provide faster access and improved performance compared to regular tables. However, the actual storage and performance characteristics may vary depending on the database system and configuration.

Here's an example of creating and using a temporary table:

```sql
-- Create a temporary table
CREATE TEMPORARY TABLE TempOrders (
    OrderID INT,
    CustomerID INT,
    OrderDate DATE
);

-- Insert data into the temporary table
INSERT INTO TempOrders (OrderID, CustomerID, OrderDate)
VALUES (1, 101, '2022-01-01'),
       (2, 102, '2022-01-02'),
       (3, 101, '2022-01-03');

-- Query the temporary table
SELECT * FROM TempOrders WHERE CustomerID = 101;
```

In this example, a temporary table named "TempOrders" is created with columns for OrderID, CustomerID, and OrderDate. Data is then inserted into the temporary table using the INSERT statement. Finally, a SELECT statement is used to query the temporary table and retrieve rows where the CustomerID is 101.

23. What is a common table expression?

A Common Table Expression (CTE) is a temporary named result set that you can reference within a SQL statement. It allows you to define a query that can be referenced multiple times within the same query, providing a more readable and modular approach to complex queries. CTEs are commonly used to break down complex queries into smaller, more manageable parts. Here's an example to illustrate the usage of a Common Table Expression:
Consider a table named "Employees" with the following data:

EmployeeID	FirstName	LastName	Department
1	John	Smith	Sales
2	Jane	Doe	Marketing
3	David	Johnson	Sales
4	Mary	Brown	HR
5	Robert	Wilson	Marketing

We can use a CTE to retrieve employees from the Sales department:

```sql
WITH SalesEmployees AS (
    SELECT EmployeeID, FirstName, LastName
    FROM Employees
    WHERE Department = 'Sales'
)
SELECT EmployeeID, FirstName, LastName
FROM SalesEmployees;
```

In this example, we define a CTE named "SalesEmployees" that retrieves the EmployeeID, FirstName, and LastName columns from the Employees table for employees in the Sales department. The CTE is then referenced in the main query, which selects and displays the EmployeeID, FirstName, and LastName columns from the SalesEmployees CTE. The result of the query would be:

EmployeeID	FirstName	LastName
1	John	Smith
3	David	Johnson

24. What is a transaction log?

In SQL, a transaction log is a file that records all modifications and changes made to a database. It is a crucial component of database management systems and plays a vital role in ensuring data integrity, durability, and recoverability. Here are some key points about the transaction log in SQL:

a. Purpose: The primary purpose of the transaction log is to maintain a record of all transactions performed on a database. It captures all modifications, such as insertions, updates, and deletions, made to the database's data and schema.
b. Recovery: The transaction log is essential for database recovery. In the event of a system failure or an error, the transaction log allows the database management system to restore the database to a previous state by applying the logged transactions or rolling them back if necessary.
c. ACID Properties: The transaction log ensures the ACID (Atomicity, Consistency, Isolation, Durability) properties of database transactions. It guarantees that transactions are atomic (all or nothing), consistent (maintains data integrity), isolated (concurrent transactions don't interfere), and durable (committed changes are permanent).
d. Point-in-Time Recovery: The transaction log enables point-in-time recovery, allowing the database to be restored to a specific moment in time. By replaying or rolling back

transactions recorded in the log, the database can be recovered to a consistent state at any desired point in the past.

e. Log Backup: Transaction logs are typically backed up regularly to ensure data protection and facilitate recovery. Log backups capture the changes made since the last backup, allowing for efficient restoration and reducing the risk of data loss.

25. What is data warehousing?

Data warehousing in SQL refers to the process of designing, building, and managing a centralized repository of data that is optimized for reporting, analytics, and business intelligence purposes. It involves extracting data from various sources, transforming it into a consistent format, and loading it into a dedicated database called a data warehouse. Here are some key aspects of data warehousing in SQL:

a. Data Integration: Data warehousing involves integrating data from multiple sources, such as operational databases, spreadsheets, external systems, and more. The data is extracted using ETL (Extract, Transform, Load) processes, where it is transformed, cleaned, and standardized to ensure consistency and quality.
b. Schema Design: The data warehouse employs a specific schema design known as a star schema or snowflake schema. These schemas consist of a central fact table surrounded by dimension tables, enabling efficient querying and analysis of data.
c. Aggregated and Historical Data: Data warehousing focuses on storing large volumes of historical and aggregated data. This allows for analysis of trends, patterns, and performance metrics over time, facilitating decision-making and strategic planning.
d. Query Performance: Data warehouses are optimized for complex analytical queries. Techniques such as indexing, partitioning, and materialized views are employed to enhance query performance and provide rapid access to large datasets.

26. What is a data mart?

A data mart is a subset of a data warehouse that focuses on a specific subject area or business function. It is a smaller, more specialized version of a data warehouse that contains a subset of data relevant to a particular department, team, or user group within an organization. Here are some key aspects of data marts:

a. Subject-Specific: A data mart is designed to address the needs of a specific subject area, such as sales, marketing, finance, or human resources. It focuses on providing data that is relevant and tailored to the requirements of a particular business function.
b. Data Subset: Data marts contain a subset of data from the larger data warehouse. They are populated with data that is specifically curated and optimized for the subject area they serve. This selective approach helps to improve performance and simplify data retrieval for the targeted users.
c. Simplified Structure: Data marts typically employ a simplified structure, such as a star schema or snowflake schema, which consists of a central fact table surrounded by dimension tables. This structure enables easier querying, reporting, and analysis within the specific subject area.

d. Departmental Scope: Data marts are often created to serve the needs of a particular department or user group within an organization. They provide a focused and self-contained data repository that allows users to access and analyze data relevant to their specific area of responsibility.

27. What is OLAP?

OLAP, which stands for Online Analytical Processing, is a technology and approach used for analyzing and querying multidimensional data from different perspectives. It enables users to perform complex, interactive analysis of data to gain insights and make informed decisions. Here are some key points about OLAP:

a. Multidimensional Analysis: OLAP allows users to analyze data along multiple dimensions, such as time, geography, product, and customer. It provides a multidimensional view of data, enabling users to drill down, roll up, slice, and dice data to explore relationships and patterns.
b. Cubes: In OLAP, data is organized into multidimensional structures called cubes. A cube represents a collection of measures (numeric values to be analyzed) and dimensions (categories or attributes). Cubes provide a structured and efficient way to store and access data for analysis.
c. Aggregation: OLAP cubes support pre-calculated aggregations, which improve query performance for complex analytical queries. Aggregations summarize data at various levels of granularity, such as total sales by year, quarter, month, and day. This allows users to quickly analyze data at different levels of detail.

28. What is ETL?

ETL stands for Extract, Transform, Load, which refers to a process commonly used in data integration and data warehousing. ETL involves extracting data from various sources, transforming it to fit the target data model or requirements, and loading it into a destination system, such as a data warehouse or a database. Here's a breakdown of each step in the ETL process:

a. Extract: In the extraction phase, data is extracted from multiple sources, which can include databases, files, APIs, web services, or other systems. The goal is to gather the necessary data for analysis or integration. Extraction methods can vary depending on the source system, such as querying databases, reading files, or using data integration tools.
b. Transform: Once the data is extracted, it often requires transformations to ensure compatibility, consistency, and quality. Transformations involve cleaning, filtering, aggregating, merging, or applying business rules to the data. Common transformation tasks include data cleansing, data validation, data standardization, and data enrichment.
c. Load: After the data is transformed, it is loaded into the target system, which could be a data warehouse, a data mart, or a database. The load phase involves mapping the transformed data to the target data model and structure. It may include tasks such as

schema creation, data mapping, data validation, and data loading into the destination system.

29. What is a database schema?

In the context of databases, a database schema refers to the logical structure or blueprint of a database. It defines the organization, relationships, and attributes of the data stored in the database. The schema provides a framework for how data is stored, how tables are related to each other, and the constraints and rules applied to the data. Here are key points about a database schema:

a. Structure: A database schema defines the structure of the database by specifying the tables, columns, data types, and constraints that make up the database. It defines the entities (tables) and attributes (columns) that represent the data and the relationships between them.
b. Table Definitions: The schema describes the tables in the database and their attributes. It specifies the name of each table, the columns within the table, and the data types of those columns. The schema may also include constraints, such as primary keys, foreign keys, and unique constraints, to enforce data integrity rules.
c. Relationships: The schema defines the relationships between tables, indicating how they are related to each other. These relationships are established through foreign key constraints, which link the primary key of one table to the corresponding column in another table. Relationships define the associations and dependencies between data entities.

30. Can you explain the difference between INSERT, UPDATE, and DELETE statements in SQL?
- INSERT is used to add new rows to a table
- UPDATE is used to modify existing rows in a table
- DELETE is used to remove rows from a table

31. How do you prevent SQL injection attacks when using CRUD operations?

To prevent SQL injection attacks when using CRUD operations (Create, Read, Update, Delete) in SQL, you should follow security best practices and employ parameterized queries or prepared statements. Here are some measures you can take:

a. Parameterized Queries/Prepared Statements: Instead of dynamically concatenating user input directly into SQL statements, use parameterized queries or prepared statements. These mechanisms separate the SQL code from the user input by treating user-supplied values as parameters, ensuring they are properly escaped and preventing malicious SQL injection.

b. Input Validation: Validate and sanitize user input on the server-side. Implement strict validation rules to ensure that user input adheres to the expected format, length, and type. Reject or sanitize any input that doesn't meet the validation criteria.
c. Least Privilege Principle: Grant database access and privileges to users and applications with the principle of least privilege. Restrict the permissions granted to database accounts to only what is necessary for the CRUD operations they perform. This minimizes the potential damage that can be caused by an SQL injection attack.
d. Stored Procedures: Utilize stored procedures to encapsulate and execute database operations. Stored procedures can help mitigate SQL injection attacks by enforcing strict parameterized input and preventing direct execution of arbitrary SQL statements. Input
e. Sanitization and Escaping: For situations where you need to incorporate user input directly into dynamic SQL statements, ensure that input is properly sanitized and escaped. Use appropriate escaping or encoding techniques to neutralize special characters and prevent them from being interpreted as part of the SQL code.
f. Database Firewall and Intrusion Detection Systems: Implement a database firewall or intrusion detection system that can identify and block suspicious SQL statements or patterns commonly used in SQL injection attacks. These systems can provide an additional layer of protection against such attacks.
g. Regular Updates and Patching: Keep your database system up to date with the latest security patches and updates. This helps to address any known vulnerabilities or weaknesses that could be exploited by attackers.

32. What is the difference between a natural join and an inner join?

A natural join is a type of join that matches rows from two tables where the values of all columns with the same name are equal. An inner join, on the other hand, matches rows from two tables where the join condition is true. In SQL, both natural join and inner join are used to combine data from two or more tables based on a specified condition. However, there is a difference between the two:

a. Natural Join: A natural join is a type of join where the columns with the same name and data type in the participating tables are automatically matched and used as the join condition. It eliminates the need to explicitly specify the join condition. The resulting join includes only the rows where the values in the matched columns are equal. Example:

```
SELECT *
FROM table1
NATURAL JOIN table2;
```

b. Inner Join: An inner join is a join that explicitly specifies the join condition using the JOIN keyword and an ON clause. It selects records where the specified condition is true, combining rows from the participating tables based on the matching values in the specified columns.

```sql
SELECT *
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

33. What is a self-join in SQL, and when would you use it?

In SQL, a self-join is a type of join where a table is joined with itself. It allows you to combine rows within the same table based on a specified condition. In a self-join, the table is typically given two different aliases to distinguish between the different occurrences of the table. Here's an example of a self-join:

```sql
SELECT e1.employee_name, e2.employee_name
FROM employees e1
JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

In this example, the "employees" table is self-joined based on the "manager_id" and "employee_id" columns. The join condition matches each employee with their respective manager, allowing you to retrieve the names of both the employee and their manager.

You would typically use a self-join in the following scenarios:

a. Hierarchical Relationships: When you have a table that represents a hierarchical structure, such as an organizational chart or a parent-child relationship, a self-join can be used to traverse and retrieve information at different levels of the hierarchy. For example, you can retrieve all employees along with their respective managers or obtain a hierarchical view of the organization.
b. Comparing Related Data: A self-join can be useful when you need to compare or analyze related data within the same table. For instance, you might want to compare the sales performance of different regions, compare product sales within a specific category, or identify employees who have the same manager.
c. Multi-Level Relationships: If your table contains multiple levels of relationships or dependencies, a self-join can help you navigate and query those relationships. This is particularly relevant when dealing with complex data models or graphs.

34. Can you explain the difference between a left join and a right join?

Left Join (or Left Outer Join): A left join returns all records from the left (or "left-hand") table and the matching records from the right (or "right-hand") table. If there is no match in the right table, NULL values are returned for the columns of the right table. The left table is the one specified before the JOIN keyword. Right Join (or Right Outer Join): A right join returns all records from the right table and the matching records from the left table. If there is no match in the left table, NULL values are returned for the columns of the left table. The right table is the one specified after the JOIN keyword.

35. What is a full outer join, and when would you use it?

A full outer join, also known as a full join, combines the results of both a left join and a right join. It returns all records from both the left and right tables, and matches the records where possible. If there is no match, NULL values are included for the columns of the non-matching table.
Syntax:

```
SELECT *
FROM left_table
FULL OUTER JOIN right_table
ON left_table.column = right_table.column;
```

```
left_table          right_table          Result (Full Outer Join)
+----+----------+   +----+----------+      +----+----------+
| ID | Name     |   | ID | Name     |      | ID | Name     |
+----+----------+   +----+----------+      +----+----------+
| 1  | John     |   | 1  | Apple    |      | 1  | Apple    |
| 2  | Sarah    |   | 3  | Orange   |      | 2  | NULL     |
| 3  | Michael  |   | 4  | Banana   |      | 3  | Orange   |
+----+----------+   +----+----------+      | 4  | Banana   |
                                           +----+----------+
```

In the example above, the full outer join returns all records from both the left_table and right_table. Matching records are joined based on the specified join condition (ID = 1, 3, 4), while non-matching records (ID = 2 in left_table) and (ID = NULL in right_table) are included in the result with NULL values for the columns of the non-matching table. You would use a full outer join when you want to retrieve all records from both tables, including matching and non-matching records. It can be useful in scenarios where you need to combine data from two tables, ensuring that you capture all available information from both sources, even if they don't have a direct match. Full outer joins are particularly handy when performing data reconciliation, data analysis, or merging datasets from multiple sources.

36. What is a correlated subquery, and when would you use it?

A correlated subquery is a type of subquery that references a column from the outer query in its WHERE or HAVING clause. The subquery is executed for each row of the outer query, using the value from the outer query to filter the results. The result of the correlated subquery is then used in the outer query's condition or to retrieve specific data. Here's an example of a correlated subquery:

```sql
SELECT column1
FROM table1 t1
WHERE column2 = (
    SELECT MAX(column2)
    FROM table2 t2
    WHERE t1.column3 = t2.column3
);
```

In this example, the correlated subquery is SELECT MAX(column2) FROM table2 t2 WHERE t1.column3 = t2.column3. The subquery is executed for each row of table1, using the value of column3 from table1 to filter the results in table2. The result of the subquery, the maximum value of column2 for the matching column3 values, is then used in the outer query's condition (WHERE column2 = ...).
You would use a correlated subquery when you need to perform a query that depends on values from the outer query. Some common scenarios where correlated subqueries are useful include:

a. Filtering Based on Aggregated Data: You can use a correlated subquery to filter rows based on aggregated values from another table. For example, finding the employees whose salary is higher than the average salary in their department.
b. Existence or Non-Existence Check: You can use a correlated subquery to check for the existence or non-existence of related records in another table. For instance, finding customers who have never placed an order by checking the absence of their records in the orders table.
c. Ranking or Top-N Queries: Correlated subqueries can be used to perform ranking or retrieve the top or bottom N records based on specific criteria. For example, retrieving the top three products with the highest sales within each category.
d. Dynamic Filtering: When you need to dynamically filter the results based on different conditions for each row, a correlated subquery can be used to achieve this. It allows you to use values from the outer query to determine the filtering criteria.

37. How do you use the COUNT function in SQL, and what does it do?

The COUNT function in SQL is used to count the number of rows that match a specific condition or meet certain criteria. It can be applied to a single column or the entire table. Here's an example: Consider a table called "students" with the following structure and sample data:

```
students table:
+----+----------+-----+
| ID | Name     | Age |
+----+----------+-----+
| 1  | John     | 20  |
| 2  | Sarah    | 22  |
| 3  | Michael  | 21  |
| 4  | Emily    | 20  |
| 5  | Jacob    | 22  |
+----+----------+-----+
```

Example 1: Counting all rows in a table

sql

```sql
SELECT COUNT(*) AS TotalRows
FROM students;
```

Output:

```
TotalRows
5
```

Example 2: Counting rows that meet a specific condition

```sql
SELECT COUNT(*) AS AdultsCount
FROM students
WHERE Age >= 21;
```

Output:

```
AdultsCount
3
```

Example 3: Counting distinct values in a column

```sql
SELECT COUNT(DISTINCT Age) AS UniqueAgesCount
FROM students;
```

Output:

```
UniqueAgesCount
3
```

In Example 1, the COUNT function is used without any condition, resulting in the count of all rows in the "students" table. In Example 2, the COUNT function is combined with a WHERE clause to count the number of rows where the "Age" column is greater than or equal to 21,

indicating the count of adults. In Example 3, the COUNT function with the DISTINCT keyword is used to count the number of unique age values in the "Age" column. The COUNT function is versatile and can be combined with other SQL statements and clauses to perform various calculations and data analysis tasks.

38. How do you use the SUM function in SQL, and what does it do?

The SUM function in SQL is used to calculate the sum of values in a specified column. It is typically used with numeric data types such as integers or decimals. Here's an example: Consider a table called "sales" with the following structure and sample data:

```
sales table:
+------+---------+-------+
| ID   | Product | Price |
+------+---------+-------+
| 1    | Apple   | 10.5  |
| 2    | Orange  | 7.2   |
| 3    | Banana  | 5.9   |
| 4    | Apple   | 12.8  |
| 5    | Orange  | 9.6   |
+------+---------+-------+
```

Example: Calculating the total sales amount

```
SELECT SUM(Price) AS TotalSales
FROM sales;
```

Output:

```
TotalSales
45.0
```

In this example, the SUM function is used to calculate the sum of the "Price" column in the "sales" table. The result, 45.0, represents the total sales amount. The SUM function can also be combined with other SQL statements and clauses for more complex calculations and data analysis tasks. Additionally, you can use the SUM function with conditions or filters to calculate the sum of values that meet specific criteria.

39. How do you use the AVG function in SQL, and what does it do?

The AVG function is used to calculate the average of a numeric column in a table. It can be used with or without the GROUP BY clause to return the overall average or the average for each group. The AVERAGE function in SQL is used to calculate the average value of a column or expression. It is commonly used with numeric data types.

Here's an example: Consider a table called "grades" with the following structure and sample data:

```
grades table:
+------+-------+
| ID   | Score |
+------+-------+
| 1    | 85    |
| 2    | 92    |
| 3    | 78    |
| 4    | 90    |
| 5    | 88    |
+------+-------+
```

Example: Calculating the average score

```
SELECT AVG(Score) AS AverageScore
FROM grades;
```

Output:

```
AverageScore
86.6
```

In this example, the AVG function is used to calculate the average value of the "Score" column in the "grades" table. The result, 86.6, represents the average score. The AVG function can also be combined with other SQL statements and clauses for more complex calculations and data analysis tasks.

40. How do you use the MAX function in SQL, and what does it do?
The MAX function is used to find the highest value in a column in a table. It can be used with or without the GROUP BY clause to return the overall maximum or the maximum for each group. Here's an example: Consider a table called "products" with the following structure and sample data:

```
products table:
+------+------------+
| ID   | Price      |
+------+------------+
| 1    | 10.5       |
| 2    | 7.2        |
| 3    | 5.9        |
| 4    | 12.8       |
| 5    | 9.6        |
+------+------------+
```

Example: Finding the maximum price

```
SELECT MAX(Price) AS MaxPrice
FROM products;
```

Output:

```
MaxPrice
12.8
```

In this example, the MAX function is used to retrieve the maximum value from the "Price" column in the "products" table. The result, 12.8, represents the highest price among all the products. The MAX function can also be used with other data types such as date/time columns to retrieve the maximum date or time value.

41. How do you use the MIN function in SQL, and what does it do?
The MIN function is used to find the lowest value in a column in a table. It can be used with or without the GROUP BY clause to return the overall minimum or the minimum for each group. Here's an example: Consider a table called "products" with the following structure and sample data:

```
products table:
+------+------------+
| ID   | Price      |
+------+------------+
| 1    | 10.5       |
| 2    | 7.2        |
| 3    | 5.9        |
| 4    | 12.8       |
| 5    | 9.6        |
+------+------------+
```

Example: Finding the minimum price

```
SELECT MIN(Price) AS MinPrice
FROM products;
```

Output:

```
MinPrice
5.9
```

In this example, the MIN function is used to retrieve the minimum value from the "Price" column in the "products" table. The result, 5.9, represents the lowest price among all the products. The MIN function can also be used with other data types such as date/time columns to retrieve the minimum date or time value.

42. How do you use the GROUP_CONCAT function in SQL, and what does it do?
The GROUP_CONCAT function is used to concatenate strings from multiple rows into a single string. It can be used with or without the GROUP BY clause to concatenate the strings for each group or all rows. The GROUP_CONCAT function in SQL is used to concatenate multiple rows of data into a single string, grouping them based on a specific column. It is available in some database management systems, such as MySQL and SQLite. Here's an example: Consider a table called "students" with the following structure and sample data:

```
students table:
+------+----------+---------+
| ID   | Name     | Subject |
+------+----------+---------+
| 1    | John     | Math    |
| 2    | Sarah    | Science |
| 3    | Michael  | Math    |
| 4    | Emily    | History |
| 5    | Jacob    | Math    |
+------+----------+---------+
```

Example: Concatenating names of students by subject

```
SELECT Subject, GROUP_CONCAT(Name) AS Students
FROM students
GROUP BY Subject;
```

Output:

```
+---------+-----------------+
| Subject | Students        |
+---------+-----------------+
| Math    | John,Michael,Jacob|
| Science | Sarah           |
| History | Emily           |
+---------+-----------------+
```

In this example, the GROUP_CONCAT function is used to concatenate the names of students based on the subject they are studying. The result is grouped by the "Subject" column. Each group of names is separated by a delimiter (a comma in this case).

43. How do you use the DENSE_RANK function in SQL, and what does it do?
The DENSE_RANK function is similar to the RANK function, but it assigns consecutive ranks to rows with the same values in the ORDER BY column. This means that there are no gaps in the ranking sequence. The DENSE_RANK function in SQL is used to assign a rank to each row within a result set based on a specified criteria. It is similar to the RANK function, but it handles ties differently. If two or more rows have the same values and should receive the same rank, the next rank is skipped, resulting in dense ranks without gaps. Here's an example: Consider a table called "scores" with the following structure and sample data:

```
scores table:
+------+-------+
| Name | Score |
+------+-------+
| John | 85    |
| Sarah| 92    |
| Emma | 78    |
| John | 90    |
| Emma | 88    |
+------+-------+
```

Example: Calculating the dense rank of scores

```
SELECT Name, Score, DENSE_RANK() OVER (ORDER BY Score DESC) AS DenseRank
FROM scores;
```

Output:

```
+------+-------+-----------+
| Name | Score | DenseRank |
+------+-------+-----------+
| Sarah| 92    | 1         |
| John | 90    | 2         |
| Emma | 88    | 3         |
| John | 85    | 4         |
| Emma | 78    | 5         |
+------+-------+-----------+
```

In this example, the DENSE_RANK function is used to assign a dense rank to each row based on the "Score" column in descending order. The result is a new column called "DenseRank" that indicates the rank of each score. Notice that when there are ties in the scores (e.g., John's scores of 85 and 90), the next rank is skipped, resulting in dense ranks without gaps. The DENSE_RANK function is commonly used for ranking and analyzing data where ties need to be handled by skipping ranks. It is particularly useful when you want to assign unique rankings to each row but maintain a dense ranking sequence.

44. How do you use the ON keyword in a JOIN statement in SQL?

The ON keyword is used to specify the condition for the JOIN operation, i.e., the common column(s) that will be used to combine the data from the tables. The ON keyword is used in SQL to specify the join condition between two tables in a JOIN operation. It allows you to define the relationship between the tables based on the specified columns or conditions. The ON keyword is followed by the join condition, which determines how the tables are linked. Here's an example: Consider two tables, "employees" and "departments," with the following structures and sample data:

```
employees table:
+------+------------+------------+
| ID   | Name       | Department |
+------+------------+------------+
| 1    | John       | 1          |
| 2    | Sarah      | 2          |
| 3    | Michael    | 1          |
| 4    | Emily      | 3          |
| 5    | Jacob      | 2          |
+------+------------+------------+

departments table:
+------+--------------+
| ID   | Department   |
+------+--------------+
| 1    | Sales        |
| 2    | Marketing    |
| 3    | HR           |
+------+--------------+
```

Example: Joining the employees and departments tables based on the department ID

```
SELECT employees.Name, departments.Department
FROM employees
JOIN departments ON employees.Department = departments.ID;
```

Output:

```
+------------+--------------+
| Name       | Department   |
+------------+--------------+
| John       | Sales        |
| Sarah      | Marketing    |
| Michael    | Sales        |
| Emily      | HR           |
| Jacob      | Marketing    |
+------------+--------------+
```

In this example, the JOIN operation is performed between the "employees" and "departments" tables using the ON keyword. The join condition is specified as employees.Department =

departments.ID, which means that the "Department" column in the "employees" table should match the "ID" column in the "departments" table. The result is a combined result set that includes the names of employees and their respective departments. The ON keyword allows you to define more complex join conditions using logical operators, comparisons, or other expressions, depending on your specific requirements. It provides flexibility in determining how the tables should be joined and linked together.

45. How do you use the UNION ALL operator in SQL, and what does it do?
The UNION ALL operator is similar to the UNION operator, but it does not remove duplicate rows from the result set. It simply combines the results of two or more SELECT statements into a single result set. The UNION ALL operator in SQL is used to combine the result sets of two or more SELECT statements into a single result set. It concatenates the rows from each SELECT statement, including duplicates, into a unified result set. Here's an example: Consider two tables, "employees" and "customers," with the following structures and sample data:

```
employees table:
+------+------------+
| ID   | Name       |
+------+------------+
| 1    | John       |
| 2    | Sarah      |
| 3    | Michael    |
+------+------------+

customers table:
+------+------------+
| ID   | Name       |
+------+------------+
| 1    | Emily      |
| 2    | Jacob      |
| 3    | Emma       |
+------+------------+
```

Example: Combining the employees and customers tables using UNION ALL

```
SELECT ID, Name
FROM employees
UNION ALL
SELECT ID, Name
FROM customers;
```

Output:

```
+------+------------+
| ID   | Name       |
+------+------------+
| 1    | John       |
| 2    | Sarah      |
| 3    | Michael    |
| 1    | Emily      |
| 2    | Jacob      |
| 3    | Emma       |
+------+------------+
```

In this example, the UNION ALL operator is used to combine the result sets of two SELECT statements. The first SELECT statement retrieves the ID and Name columns from the "employees" table, while the second SELECT statement retrieves the same columns from the "customers" table. The result is a single unified result set that includes all the rows from both tables, including duplicates.

46. What is a cross join in SQL, and how is it used?

A cross join, also known as a Cartesian join, in SQL is a type of join operation that combines each row from the first table with every row from the second table, resulting in a Cartesian product of the two tables. In other words, it generates all possible combinations of rows between the two tables. Here's an example to illustrate: Consider two tables, "colors" and "sizes," with the following structures and sample data:

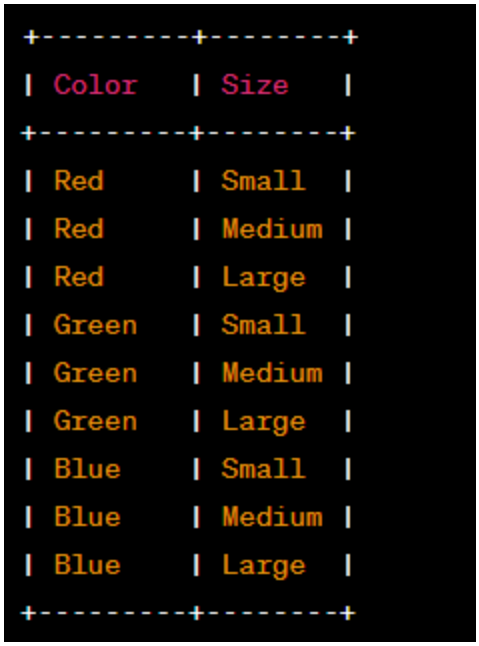
```
colors table:
+---------+
| Color   |
+---------+
| Red     |
| Green   |
| Blue    |
+---------+

sizes table:
+--------+
| Size   |
+--------+
| Small  |
| Medium |
| Large  |
+--------+
```

Example: Performing a cross join between colors and sizes

```
SELECT *
FROM colors
CROSS JOIN sizes;
```

Output:



Color	Size
Red	Small
Red	Medium
Red	Large
Green	Small
Green	Medium
Green	Large
Blue	Small
Blue	Medium
Blue	Large

In this example, the CROSS JOIN operator is used to perform a cross join between the "colors" and "sizes" tables. It generates all possible combinations of colors and sizes, resulting in a Cartesian product. Each row from the "colors" table is paired with every row from the "sizes" table. Cross joins are typically used when you want to generate all possible combinations between two tables, especially when there is no explicit relationship or join condition specified.

47. How do you use the NATURAL JOIN keyword in SQL, and what does it do?
The NATURAL JOIN keyword is used to combine data from two or more tables based on columns that have the same name and data type. It automatically detects the common columns and performs the JOIN operation based on those columns. The NATURAL JOIN in SQL is a join operation that combines two tables based on columns with the same name and automatically matches them. It does not require explicitly specifying the join condition. The join is performed by matching columns with identical names in both tables. Here's an example: Consider two tables, "employees" and "departments," with the following structures and sample data:

```
employees table:
+------+----------+------------+
| ID   | Name     | Department |
+------+----------+------------+
| 1    | John     | Sales      |
| 2    | Sarah    | Marketing  |
| 3    | Michael  | Sales      |
| 4    | Emily    | HR         |
| 5    | Jacob    | Marketing  |
+------+----------+------------+

departments table:
+------+--------------+
| ID   | Department   |
+------+--------------+
| 1    | Sales        |
| 2    | Marketing    |
| 3    | HR           |
+------+--------------+
```

Example: Performing a natural join between employees and departments

```
SELECT *
FROM employees
NATURAL JOIN departments;
```

Output:

```
+------+----------+------------+
| ID   | Name     | Department |
+------+----------+------------+
| 1    | John     | Sales      |
| 3    | Michael  | Sales      |
| 2    | Sarah    | Marketing  |
| 5    | Jacob    | Marketing  |
| 4    | Emily    | HR         |
+------+----------+------------+
```

In this example, the NATURAL JOIN operator is used to combine the "employees" and "departments" tables. The join is performed by matching the columns with the same name ("Department" in this case) in both tables. The result is a new table that includes the matched rows from both tables, only retaining the common column once in the output. It's important to note that the NATURAL JOIN relies on column names to match and perform the join. Therefore, if the tables have additional columns with the same name that should not be used for the join, it can lead to unintended results or ambiguity.

48. What is a check constraint in SQL, and how is it used?

A CHECK constraint in SQL is used to specify a condition that must be true for each row in a table. It ensures that the data stored in the table meets certain criteria. The CHECK constraint is applied to a column or a combination of columns and restricts the values that can be inserted or updated. Here's an example: Consider a table called "employees" with the following structure:

```
employees table:
+------+----------+--------+
| ID   | Name     | Salary |
+------+----------+--------+
| 1    | John     | 5000   |
| 2    | Sarah    | 6000   |
| 3    | Michael  | 4500   |
| 4    | Emily    | 7000   |
+------+----------+--------+
```

Example: Adding a CHECK constraint to restrict the salary range

```
ALTER TABLE employees
ADD CONSTRAINT chk_salary_range CHECK (Salary >= 4000 AND Salary <= 8000);
```

In this example, a CHECK constraint is added to the "employees" table to ensure that the "Salary" column values fall within the range of 4000 to 8000. The constraint is specified using the CHECK keyword, followed by the condition enclosed in parentheses. The condition (Salary >= 4000 AND Salary <= 8000) ensures that the salary values are between 4000 and 8000 (inclusive). Once the CHECK constraint is defined, any attempt to insert or update a row that violates the specified condition will result in an error.

49. What is a unique constraint in SQL, and how is it used?

A UNIQUE constraint in SQL is used to ensure that the values in a column or a combination of columns are unique across all rows in a table. It guarantees that each value in the specified column(s) is unique and does not allow duplicate entries. Here's an example: Consider a table called "students" with the following structure:

```
students table:
+------+----------+--------+
| ID   | Name     | Email  |
+------+----------+--------+
| 1    | John     | john@example.com     |
| 2    | Sarah    | sarah@example.com    |
| 3    | Michael  | michael@example.com  |
+------+----------+--------+
```

Example: Adding a UNIQUE constraint to the email column

```
ALTER TABLE students
ADD CONSTRAINT uc_email UNIQUE (Email);
```

In this example, a UNIQUE constraint is added to the "Email" column of the "students" table. The CONSTRAINT keyword is used to define the constraint, followed by a name for the constraint (in this case, "uc_email") and the UNIQUE keyword. The column name "Email" is specified within parentheses to indicate that the uniqueness constraint applies to this column. Once the UNIQUE constraint is defined, it ensures that each email value in the "Email" column is unique. Any attempt to insert or update a row with a duplicate email value will result in an error.

50. What is a default constraint in SQL, and how is it used?

A DEFAULT constraint in SQL is used to specify a default value for a column when no explicit value is provided during an INSERT operation. It defines a default value that is automatically assigned to the column if no other value is specified. Here's an example: Consider a table called "employees" with the following structure:

```
employees table:
+------+----------+---------+
| ID   | Name     | Gender  |
+------+----------+---------+
| 1    | John     | M       |
| 2    | Sarah    | F       |
| 3    | Michael  | M       |
+------+----------+---------+
```

Example: Adding a DEFAULT constraint to the gender column

```
ALTER TABLE employees
ALTER COLUMN Gender SET DEFAULT 'Unknown';
```

Once the DEFAULT constraint is defined, if a row is inserted into the table without providing a value for the "Gender" column, the default value 'Unknown' will be automatically assigned to that column for that row. For example, if you insert a new row into the "employees" table without specifying the "Gender" value:

```
INSERT INTO employees (ID, Name) VALUES (4, 'Emily');
```

The resulting table will be:

```
+------+---------+---------+
| ID   | Name    | Gender  |
+------+---------+---------+
| 1    | John    | M       |
| 2    | Sarah   | F       |
| 3    | Michael | M       |
| 4    | Emily   | Unknown |
+------+---------+---------+
```

The "Gender" column for the newly inserted row is assigned the default value 'Unknown' since no explicit value was provided. The DEFAULT constraint is useful when you want to ensure that a specific default value is assigned to a column if no other value is specified. It simplifies data entry and ensures consistency in the absence of explicitly provided values.

51. What is a null constraint in SQL, and how is it used?
A null constraint in SQL is a table constraint that specifies whether a column can contain NULL values or not. It is used to ensure data integrity by preventing NULL values from being inserted into columns where they are not allowed. In SQL, a NULL constraint is used to enforce that a column does not contain any NULL values. It ensures that a specific column must always have a non-NULL value. Here's an example: Consider a table called "students" with the following structure:

```
students table:
+------+---------+--------+
| ID   | Name    | Grade  |
+------+---------+--------+
| 1    | John    | A      |
| 2    | Sarah   | NULL   |
| 3    | Michael | B      |
+------+---------+--------+
```

Example: Adding a NULL constraint to the grade column

```
ALTER TABLE students
ALTER COLUMN Grade SET NOT NULL;
```

Once the NULL constraint is defined, any attempt to insert or update a row with a NULL value in the "Grade" column will result in an error. For example, if you try to insert a new row with a NULL value for the "Grade" column:

```
INSERT INTO students (ID, Name, Grade) VALUES (4, 'Emily', NULL);
```

An error will be thrown because the NULL constraint does not allow the "Grade" column to have a NULL value. By setting a NULL constraint, you ensure that the column always contains a valid value, improving data integrity and consistency. It helps prevent unexpected or missing data in columns where NULL values are not allowed.

52. How do you add a table constraint to an existing table in SQL?

To add a table constraint to an existing table in SQL, you can use the ALTER TABLE statement along with the ADD CONSTRAINT clause. Here's the general syntax:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name constraint_definition;
```

Let's break down the syntax:
ALTER TABLE is used to modify an existing table. table_name is the name of the table to which you want to add the constraint.
ADD CONSTRAINT is the clause that indicates you want to add a new constraint.
constraint_name is a user-defined name for the constraint.
Choose a descriptive name that reflects the purpose of the constraint.
constraint_definition specifies the constraint itself, including its type and condition.

53. How do you modify or remove a table constraint in SQL?

To modify or remove a table constraint in SQL, you can use the ALTER TABLE statement with the MODIFY CONSTRAINT or DROP CONSTRAINT clause, respectively. Here's how you can do it: Modifying a table constraint:

```
ALTER TABLE table_name
ALTER CONSTRAINT constraint_name NEW_CONSTRAINT_DEFINITION;
```

In this syntax: table_name is the name of the table containing the constraint. constraint_name is the name of the constraint you want to modify. NEW_CONSTRAINT_DEFINITION is the new definition or condition for the constraint.

Removing a table constraint:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

In this syntax: table_name is the name of the table containing the constraint. constraint_name is the name of the constraint you want to remove. By using the DROP CONSTRAINT clause, you can remove the specified constraint from the table.

54. Explain the row_number window function with an example.

The row_number window function assigns a unique sequential number to each row within a specified window. It is commonly used to create a ranking or to add a unique identifier to each row in the result set. The row_number function operates independently of the data values and is based solely on the order of the rows within the window. Let's consider an example to better understand the row_number function. Assume we have a table named "Students" with the following columns: "StudentID," "Name," "Grade," and "Score." We want to assign a unique rank to each student based on their score within each grade. Here's an example query:

```
SELECT StudentID, Name, Grade, Score,
       ROW_NUMBER() OVER (PARTITION BY Grade ORDER BY Score DESC) AS Rank
FROM Students
```

In this query, the row_number function is used to generate a rank for each student within their respective grade. The "PARTITION BY Grade" clause ensures that the ranking restarts for each grade, while the "ORDER BY Score DESC" clause orders the students within each grade in descending order of their score. The result of this query might look like:

```
StudentID | Name     | Grade | Score | Rank
-----------------------------------------
1         | John     | A     | 95    | 1
2         | Emily    | A     | 92    | 2
3         | Michael  | A     | 89    | 3
4         | Sophia   | B     | 88    | 1
5         | William  | B     | 85    | 2
6         | Olivia   | B     | 82    | 3
7         | Ethan    | B     | 79    | 4
```

As shown in the result, each student is assigned a unique rank based on their score within their respective grade. The student with the highest score in each grade receives a rank of 1, followed by subsequent ranks for lower-scoring students within the same grade. The row_number function is valuable when you need to establish an ordered sequence of rows or assign a unique identifier to each row within a specific grouping or partitioning criteria. It allows for various ranking and numbering scenarios in data analysis and reporting.

55. Can you update or delete data in a view in SQL?

In SQL, a view is a virtual table that is based on the result of a query. It does not store any data itself but provides a way to present data from underlying tables in a customized manner. Views are read-only by default, meaning you cannot directly update or delete data through a view. However, you can update or delete data indirectly by modifying the underlying tables that the view is based on. To update data through a view, you need to update the underlying table(s) directly. Any changes made to the tables will be reflected in the view. Here's an example:

```sql
-- Create a view
CREATE VIEW my_view AS
SELECT column1, column2
FROM my_table
WHERE condition;

-- Update data in the underlying table
UPDATE my_table
SET column1 = 'new_value'
WHERE condition;
```

When you update the data in my_table, the changes will be visible when you query my_view. To delete data through a view, you can use the DELETE statement with appropriate conditions to target the rows you want to delete. Again, the actual deletion happens in the underlying table(s). Here's an example:

```sql
-- Create a view
CREATE VIEW my_view AS
SELECT column1, column2
FROM my_table
WHERE condition;

-- Delete data from the underlying table
DELETE FROM my_table
WHERE condition;
```

The rows that match the specified condition will be deleted from my_table, and the view will reflect the updated data.

56. What is the difference between a view and a table in SQL?

In SQL, a table and a view are both database objects used to store and retrieve data, but they have some key differences:

a. Data Storage: A table is a physical structure that stores data in the database, occupying storage space on disk. It consists of rows and columns, where each row represents a record, and each column represents a data attribute. In contrast, a view is a virtual table that does not store any data itself. It is a saved query that retrieves data from one or more underlying tables.
b. Data Modification: Tables allow both read and write operations. You can insert, update, and delete data directly into a table. On the other hand, views are typically used for read operations only. While you can update or delete data indirectly through a view by modifying the underlying tables, views themselves are usually read-only. They provide a convenient way to present data from multiple tables or apply complex queries without altering the underlying data.
c. Structure and Schema: Tables have a defined structure with named columns and data types. Each column has a specific name, data type, and constraints. Views, on the other hand, do not have their own physical structure. They derive their structure from the underlying tables or expressions used in the view definition. The columns and data types in a view are determined by the query used to create the view.

57. What are the properties of a transaction in SQL?

In SQL, a transaction is a logical unit of work that consists of one or more database operations. Transactions ensure data integrity and provide the following properties, commonly known as ACID properties:

a. Atomicity: Atomicity ensures that a transaction is treated as a single indivisible unit of work. It means that either all the operations within a transaction are successfully completed, or none of them are applied. If any operation within a transaction fails or encounters an error, the entire transaction is rolled back, and the database is left unchanged.
b. Consistency: Consistency ensures that a transaction brings the database from one consistent state to another. It means that a transaction must satisfy all the integrity constraints defined on the database. If a transaction violates any constraints, the changes made by the transaction are rolled back, and the database remains unchanged.
c. Isolation: Isolation ensures that concurrent transactions do not interfere with each other. Each transaction must be executed as if it is the only transaction being executed, even if multiple transactions are executing concurrently.
d. Durability: Durability guarantees that once a transaction is committed, its changes are permanently stored in the database and survive any subsequent failures, such as power outages or system crashes.

58. What is a savepoint in SQL, and when would you use it?

In SQL, a savepoint is a way to mark a specific point within a transaction. It allows you to create a named checkpoint or intermediate state within a transaction, which can be used to roll back to that particular point later if needed. Savepoints are particularly useful when you want to undo only a portion of the changes made within a transaction while keeping the rest intact.

59. How do you use savepoints in SQL?
To use savepoints in SQL, you can follow these steps: Begin a Transaction: Start a transaction using the BEGIN TRANSACTION or BEGIN statement. This marks the beginning of the transaction and allows you to create savepoints within it.

```
BEGIN TRANSACTION;
-- or
BEGIN;
```

Create a Savepoint: To create a savepoint, use the SAVEPOINT statement followed by the name you want to assign to the savepoint. The savepoint name must be unique within the transaction.

```
SAVEPOINT savepoint_name;
```

Execute SQL Statements: Continue executing SQL statements within the transaction. These statements can include data manipulation (e.g., INSERT, UPDATE, DELETE) or other database operations.

```
-- SQL statements within the transaction
INSERT INTO table_name (column1, column2) VALUES (value1, value2);
UPDATE table_name SET column1 = value WHERE condition;
-- Additional statements...
```

Rollback to a Savepoint: If needed, you can roll back to a specific savepoint within the transaction using the ROLLBACK TO SAVEPOINT statement. This undoes all changes made after the savepoint, restoring the transaction to the savepoint's state.

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

It's important to note that savepoints are only applicable within a single transaction. If you attempt to reference a savepoint outside the transaction or in a different transaction, it will result in an error.

60. What are window functions in SQL, and how do they differ from regular aggregate functions?

In SQL, window functions are a powerful feature that allows you to perform calculations and aggregations over a specific subset or "window" of rows within a result set. Window functions

operate on a set of rows defined by a window specification, which can be based on a range of rows or a logical partition of the data. Here are some key differences between window functions and regular aggregate functions:

a. Scope of Operation: Regular aggregate functions, such as SUM, COUNT, and AVG, operate on an entire result set or a specific group of rows defined by a GROUP BY clause. In contrast, window functions perform calculations on individual rows within the result set, without collapsing them. The window function's result is associated with each row in the output, rather than producing a single result for the entire group.
b. Inclusion of Additional Columns: When using regular aggregate functions, if you want to include additional columns in the output, you typically need to group by those columns. This can result in a more limited and aggregated view of the data. With window functions, you can include additional columns without modifying the window specification.
c. Window Specification: Window functions require a window specification that defines the set of rows the function operates on. The window specification can include clauses such as PARTITION BY, ORDER BY, and ROWS/RANGE.
d. Flexibility in Calculations: Window functions offer more flexibility in calculations compared to regular aggregate functions. You can perform various calculations within the window, such as computing cumulative sums, calculating row numbers, finding the first or last value in a window, and calculating moving averages.

61. What are some common uses for window functions in SQL?

Window functions in SQL offer a wide range of capabilities for data analysis and reporting. Here are some common uses for window functions:

a. Calculating Aggregates: Window functions allow you to compute various aggregates over a specific window of rows. This includes functions like SUM, AVG, MIN, MAX, and COUNT.
b. Ranking and Percentiles: Window functions enable ranking and percentile calculations, providing insights into the relative position of rows within a result set. Functions like ROW_NUMBER, RANK, DENSE_RANK, and NTILE help identify top performers, find the highest and lowest values, or divide data into equal-sized groups based on a specified criterion.
c. Moving Averages and Rolling Aggregates: Window functions allow you to calculate moving averages and rolling aggregates over a defined window of rows. This is useful for generating trend analysis or smoothing out fluctuations in data. Functions like AVG or SUM with the ROWS or RANGE clause enable sliding window calculations.
d. Windowing Data: Window functions provide control over the window specification, allowing you to define partitions and order rows within those partitions. This flexibility enables calculations on different subsets of data based on specific criteria.
e. Lead and Lag Analysis: Window functions like LEAD and LAG allow you to access values from the next or previous rows within a specified window. This is valuable for

analyzing time series data, identifying changes over time, or comparing values between adjacent rows.

f. Running Totals and Cumulative Aggregates: Window functions enable the calculation of running totals and cumulative aggregates. You can calculate cumulative sums, products, or other aggregates by including all preceding rows up to the current row. This is useful for financial analysis, inventory management, or tracking progress over time.

g. Data Imputation and Fill Values: Window functions provide the ability to impute missing or null values based on the surrounding data within a window. You can use functions like COALESCE or LAST_VALUE to fill in missing values with the last known value or derive imputed values based on the available data.

62. How do you use a window function in SQL?

To use a window function in SQL, you can use the OVER clause in the SELECT statement, followed by the partitioning and ordering clauses that define the set of rows to include in the calculation. The window function is then applied to this set of rows to produce the result.
Here's an example to illustrate the usage of a window function. Let's say we have a table called sales with columns region, year, and sales_amount. We want to calculate the average sales amount per region for each year, along with the individual sales amount and the average across all regions for that year.

```
SELECT
    region,
    year,
    sales_amount,
    AVG(sales_amount) OVER (PARTITION BY year) AS avg_sales_per_year,
    AVG(sales_amount) OVER () AS overall_avg_sales
FROM
    sales;
```

In the above example, we use the AVG window function to calculate the average sales amount per year (AVG(sales_amount) OVER (PARTITION BY year)) and the overall average sales amount across all regions (AVG(sales_amount) OVER ()). The window function is applied to each row individually, and the results are included in the output for each row.

63. What is a Common Table Expression (CTE) in SQL, and how does it differ from a subquery?

A Common Table Expression (CTE) in SQL is a temporary named result set that can be referenced within a query. It provides a way to define a query block that can be treated as a virtual table within the scope of a single query. CTEs improve code readability and reusability by allowing complex subqueries or derived tables to be defined and referenced as a separate entity. Here are some key differences between a CTE and a subquery:

a. Readability and Maintainability: CTEs enhance the readability and maintainability of SQL queries by allowing you to break down complex queries into smaller, more manageable parts. This promotes code reuse and reduces the need to repeat complex subqueries.
b. Self-Referencing and Recursive Queries: CTEs are particularly useful for self-referencing or recursive queries, where a query refers to itself during execution. With a CTE, you can define the base case and recursive part of the query separately and combine them using UNION ALL.
c. Query Structure: In a subquery, the subquery is embedded within the main query, typically in the FROM or WHERE clause.CTEs, on the other hand, are defined separately using the WITH clause before the main query and are referenced by name within the main query.
d. Reusability: CTEs provide a way to reuse the defined query block multiple times within a single query. You can reference the CTE multiple times in the same query, simplifying complex logic and avoiding duplication. Subqueries, on the other hand, are typically used inline within a specific query and cannot be referenced multiple times.
e. Optimization: In some cases, the use of CTEs may allow for better query optimization by the database engine. Subqueries, on the other hand, may be treated as independent queries and optimized individually.

64. What is a recursive CTE in SQL, and when would you use it?

A recursive Common Table Expression (CTE) in SQL is a CTE that references itself in its own definition. It allows you to perform iterative operations or traverse hierarchical data structures. Recursive CTEs provide a powerful way to express recursive or iterative logic in SQL queries. The recursive CTE consists of two parts:

a. Base case: This is the initial query that forms the starting point of the recursion. It selects the initial set of rows that will be used in the recursive part of the CTE.
b. Recursive part: This is the query that references the CTE itself within its definition. It uses the results of the previous iteration to generate new rows that will be combined with the previous iteration's results. The recursion continues until a termination condition is met, such as reaching a specific depth in a hierarchical structure or satisfying a certain condition.

Recursive CTEs are commonly used in scenarios such as:

a. Hierarchical data traversal: Recursive CTEs are particularly useful for traversing and querying hierarchical data structures like organization charts, file systems, product categories, or bill-of-materials structures. By defining the recursive relationship and base case, you can recursively navigate through the hierarchy and perform operations on each level.
b. Graph traversal: If you have graph-like data, such as social networks or network topologies, recursive CTEs can help you traverse the graph and analyze the relationships between nodes. You can define the recursive relationship based on graph edges and find paths, shortest distances, or connected components.

c. Iterative calculations: Recursive CTEs can be used to perform iterative calculations, such as calculating running totals, cumulative aggregations, or iterative mathematical computations. Each iteration builds upon the results of the previous iteration, allowing you to perform complex calculations or simulations.

65. What is string transformation in SQL, and how is it used?

String transformation in SQL refers to the manipulation and modification of string values using built-in functions and operators. It allows you to change the format, case, length, or content of strings to meet specific requirements or perform certain operations. Here are some commonly used string transformation functions and techniques in SQL:

a. Concatenation: The concatenation operator (|| in most database systems) allows you to combine multiple strings together.
b. Substring Extraction: Functions like SUBSTRING or SUBSTR retrieve a portion of a string based on a specified starting position and length.
c. Case Conversion: Functions like UPPER and LOWER change the case of a string to uppercase or lowercase, respectively. This can be handy for standardizing the case of string data or for case-insensitive comparisons.
d. String Manipulation: SQL provides various functions for manipulating strings. Examples include REPLACE (to replace occurrences of a substring with another substring), TRIM (to remove leading and trailing spaces), LEFT (to extract a specified number of characters from the left side of a string), and RIGHT (to extract a specified number of characters from the right side of a string).
e. Padding and Trimming: Functions like LPAD and RPAD enable you to pad a string with a specific character to a specified length.
f. String Length: The LENGTH or LEN function calculates the length of a string, allowing you to determine the number of characters in a string.

String transformation functions are used in a wide range of scenarios, including data cleaning, data manipulation, generating reports, formatting output, and performing text-based searches or comparisons. By applying string transformation functions, you can modify and manipulate string values to fit your specific requirements and derive meaningful insights from your data.

66. What is a regular expression (regex) in SQL, and how is it used?

A regular expression (regex) in SQL is a pattern-matching language used to search, match, and manipulate text data based on specific patterns. It provides a powerful and flexible way to perform complex string matching and transformation operations. SQL databases often support regular expressions through specific functions or operators. These functions enable you to

search for patterns within strings, extract matching substrings, replace matched patterns with other strings, and more. Here are some common use cases and functions for working with regular expressions in SQL:

a. Pattern Matching: Regular expressions allow you to search for specific patterns within a string. For example, you can use the REGEXP_LIKE function to check if a string matches a particular pattern or use the REGEXP_SUBSTR function to extract a substring that matches a specific pattern.
b. String Replacement: SQL provides functions like REGEXP_REPLACE that allow you to find and replace substrings within a string using regular expressions. This is useful for performing complex search-and-replace operations based on patterns.
c. String Extraction: Regular expressions enable you to extract specific parts of a string that match a given pattern. Functions like REGEXP_SUBSTR or REGEXP_EXTRACT can be used to retrieve portions of a string based on a specified regular expression pattern.
d. String Splitting: Regular expressions can help split a string into multiple substrings based on a delimiter or pattern. Functions like REGEXP_SPLIT_TO_ARRAY or REGEXP_SPLIT can be used to split a string into an array or generate a table of values based on a regular expression pattern.
e. Validation: Regular expressions are often used for validating input data against specific patterns. For example, you can use a regular expression to ensure that a user-provided email address follows a valid email format.
f. Data Cleaning: Regular expressions are useful for cleaning and normalizing data. They can be employed to remove unwanted characters, replace invalid values, or standardize the format of data.

67. What are some common regex patterns used in SQL?

Regular expressions (regex) can be used in SQL queries to perform pattern matching and data manipulation tasks. Here are some common regex patterns used in SQL:

a. Match any character: "." Example: SELECT * FROM table WHERE column LIKE 'a%'
b. Match a specific character: "[ ]" Example: SELECT * FROM table WHERE column LIKE '[a-c]%'
c. Match any character in a range: "[-]" Example: SELECT * FROM table WHERE column LIKE '[0-9]%'
d. Match any character not in a range: "[^]" Example: SELECT * FROM table WHERE column LIKE '[^aeiou]%'
e. Match a specific character occurring zero or one time: "?" Example: SELECT * FROM table WHERE column LIKE 'colou?r%'
f. Match a specific character occurring zero or more times: "*" Example: SELECT * FROM table WHERE column LIKE 'colou*r%'
g. Match a specific character occurring one or more times: "+" Example: SELECT * FROM table WHERE column LIKE 'colou+r%'
h. Match the beginning of a string: "^" Example: SELECT * FROM table WHERE column LIKE '^abc%'

i. Match the end of a string: "$" Example: SELECT * FROM table WHERE column LIKE '%xyz$'
j. Match a word boundary: "\b" Example: SELECT * FROM table WHERE column REGEXP '\\bword\\b'

68. How can you use regex to validate data in SQL?

Regex can be used in SQL to validate data by checking if a particular string conforms to a specific pattern. Here's an example of how you can use regex to validate data in SQL: Let's say you have a table called "users" with a column called "email" that stores email addresses. You want to ensure that the email addresses entered by users follow a valid format. You can use regex to validate the email addresses before inserting or updating the data in the table. Here's an example of how you can validate email addresses using regex in SQL:

```
-- Example: Validate email addresses using regex
INSERT INTO users (email)
VALUES ('example@example.com')
WHERE email REGEXP '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$';
```

In this example, the REGEXP function is used to match the email address against a regular expression pattern. The pattern ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ checks for the following criteria:

- Starts with one or more alphanumeric characters, dots, underscores, percent signs, plus signs, or hyphens.
- Followed by the "@" symbol.
- Followed by one or more alphanumeric characters, dots, or hyphens.
- Followed by a dot.
- Ends with two or more alphabetic characters.

If the email address matches the specified pattern, the data will be inserted into the "users" table. Otherwise, the insertion will be rejected, indicating that the email address is invalid.

69. What are some potential drawbacks of using regex in SQL?

While regex can be a powerful tool for pattern matching and data validation in SQL, there are some potential drawbacks to consider:

a. Complexity: Regular expressions can be complex and difficult to understand, especially for users who are not familiar with regex syntax. Constructing and maintaining complex regex patterns can be time-consuming and error-prone.
b. Performance Impact: Regex matching can be resource-intensive, especially for large datasets. Complex regex patterns or patterns that require backtracking can lead to slow query execution times and increased resource usage.
c. Limited Portability: Regex syntax and functionality can vary between different SQL database systems. A regex pattern that works in one database may not work in another,

or the syntax may differ slightly. This lack of portability can make it challenging to write SQL queries that use regex across different database platforms.

d. Limited Functionality: SQL's built-in regex support may be limited compared to dedicated regex libraries or programming languages. Some advanced regex features or operations may not be available in SQL, making it harder to accomplish certain tasks.
e. Maintenance Challenges: Over time, regex patterns can become difficult to maintain and modify. As requirements change or new patterns need to be implemented, understanding and modifying existing regex patterns can be challenging, leading to potential errors or unintended behavior.

70. What is data modeling in SQL, and why is it important?

Data modeling in SQL refers to the process of designing the structure and relationships of a database system. It involves creating a conceptual representation of the data, defining tables, columns, constraints, and establishing the connections between them. The resulting data model serves as a blueprint for organizing, storing, and manipulating data within a database. Data modeling is important for several reasons:

a. Structure and Organization: A well-designed data model provides a structured and organized approach to store and retrieve data. It defines the entities, attributes, and relationships, ensuring consistency and integrity in data storage.
b. Data Integrity and Quality: By enforcing constraints and relationships, data modeling helps maintain data integrity and quality. It ensures that data is accurate, consistent, and adheres to defined business rules, minimizing data anomalies or inconsistencies.
c. Efficient Data Operations: A well-designed data model allows for efficient data retrieval, manipulation, and querying. By optimizing the schema and defining appropriate indexes, data modeling helps improve the performance of SQL operations, such as SELECT, UPDATE, and DELETE statements.
d. Scalability and Flexibility: Data modeling considers future scalability and flexibility needs. By anticipating potential data growth, changes in requirements, and business rules, the data model can be designed to accommodate future expansions or modifications without significant disruptions.
e. Communication and Collaboration: Data modeling provides a common language and visual representation of the database structure. It facilitates communication and collaboration between stakeholders, including database administrators, developers, and business users, ensuring a shared understanding of the data model and its implications.
f. Application Development: A well-designed data model simplifies application development. Developers can use the data model as a guide to map business requirements to database entities, define data access methods, and ensure data consistency across different application modules.

71. What is an entity-relationship (ER) diagram?

An Entity-Relationship (ER) diagram is a visual representation of the entities (objects or concepts) within a system, their attributes, and the relationships between them. It is a popular technique used in data modeling to design and communicate the structure and relationships of a

database system. In an ER diagram, entities are represented as rectangles, attributes as ovals or ellipses, and relationships as lines connecting entities. The diagram provides a high-level overview of the database schema and helps stakeholders understand the data model and its components. Here are the key components of an ER diagram:

a. Entities: Entities represent the objects or concepts in the system being modeled. They typically correspond to tables in a database. Each entity is identified by a unique identifier known as a primary key. Examples of entities could be "Customer," "Product," or "Order."
b. Attributes: Attributes define the characteristics or properties of an entity. They represent the data elements stored within an entity. Attributes are depicted as ovals or ellipses and are connected to the corresponding entity. For example, an attribute of a "Customer" entity could be "Name" or "Email."
c. Relationships: Relationships illustrate the associations and dependencies between entities. They describe how entities are related to each other. Relationships can be one-to-one, one-to-many, or many-to-many. They are represented as lines connecting entities, and their cardinality (the number of instances) is indicated using symbols such as "1" or "N" on each end of the line.
d. Cardinality: Cardinality represents the number of instances or occurrences of an entity that can be associated with another entity through a relationship. It specifies the relationship's multiplicity. For example, a one-to-many relationship between "Customer" and "Order" means that one customer can have multiple orders, while each order is associated with only one customer.

72. What are some common types of relationships between entities in SQL data modeling?

In SQL data modeling, there are several common types of relationships that can exist between entities. These relationships define how entities are related to each other and help establish the structure and integrity of the database. Here are some commonly used types of relationships:

a. One-to-One (1:1) Relationship: In a one-to-one relationship, each instance of one entity is associated with exactly one instance of another entity. Example: A "Person" entity and a "Passport" entity, where each person has only one passport, and each passport belongs to only one person.
b. One-to-Many (1:N) Relationship: In a one-to-many relationship, each instance of one entity is associated with zero or more instances of another entity. Example: A "Department" entity and an "Employee" entity, where each department can have multiple employees, but each employee belongs to only one department.
c. Many-to-One (N:1) Relationship: In a many-to-one relationship, multiple instances of one entity are associated with exactly one instance of another entity. Example: An "Order" entity and a "Customer" entity, where multiple orders can be placed by the same customer, but each order is associated with only one customer.
d. Many-to-Many (N:N) Relationship: In a many-to-many relationship, multiple instances of one entity are associated with multiple instances of another entity. Example: A "Student"

entity and a "Course" entity, where multiple students can enroll in multiple courses, and each course can have multiple students.

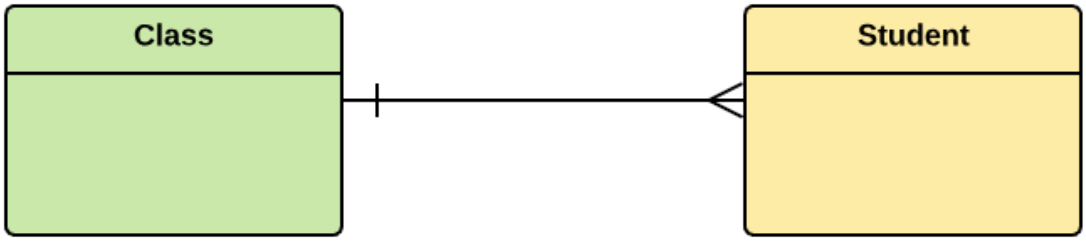
To represent these relationships in an entity-relationship (ER) diagram, lines are drawn between the entities, indicating the type of relationship and its cardinality. Cardinality is often indicated using symbols like "1" or "N" on each end of the line, representing the number of instances involved.

73. How do you represent a one-to-many relationship in an ER diagram?

In an Entity-Relationship (ER) diagram, a one-to-many relationship is represented using a specific notation. Here's how you can represent a one-to-many relationship:

a. Identify the entities involved: Let's say we have two entities, Entity A and Entity B.
b. Determine the cardinality: In a one-to-many relationship, an instance of Entity A can be associated with multiple instances of Entity B, but an instance of Entity B can only be associated with one instance of Entity A.
c. Represent the relationship using a crow's foot notation: In the ER diagram, you can represent the one-to-many relationship by drawing a straight line between Entity A and Entity B. At the end of the line connected to Entity B, draw a small crow's foot symbol (⋂) to indicate the "many" side of the relationship.

For example, one class consists of multiple students.



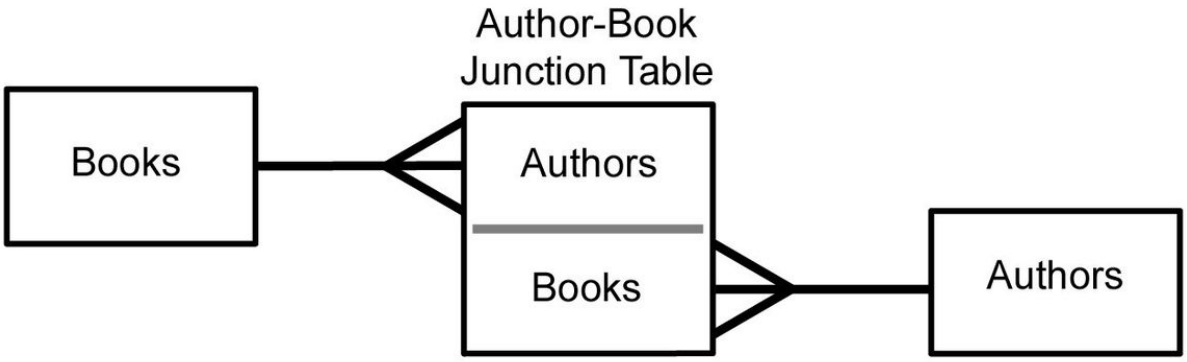
74. How do you represent a many-to-many relationship in an ER diagram?

In an Entity-Relationship (ER) diagram, a many-to-many relationship is represented using a specific notation. Here's how you can represent a many-to-many relationship:

a. Identify the entities involved: Let's say we have two entities, Entity A and Entity B.
b. Determine the cardinality: In a many-to-many relationship, an instance of Entity A can be associated with multiple instances of Entity B, and vice versa. Both entities can have multiple associations with each other.
c. Create an intermediate entity (junction table): To represent a many-to-many relationship, you need to introduce an intermediate entity or junction table. This entity acts as a bridge between Entity A and Entity B, capturing the associations between them.

d. Represent the entities and the junction table: In the ER diagram, you represent Entity A, Entity B, and the junction table as separate entities. Connect Entity A and Entity B to the junction table using one-to-many relationships.

Example:



75. What is a data dictionary in SQL data modeling, and how is it used?

In SQL data modeling, a data dictionary is a centralized repository that provides detailed information about the data structures, attributes, relationships, and constraints within a database system. It serves as a documentation tool that describes the metadata of the database, including tables, columns, data types, primary keys, foreign keys, indexes, and other relevant information. The data dictionary is typically maintained and accessed by database administrators, data architects, and developers. It can be created manually or automatically generated by database management systems (DBMS) based on the schema definition. The data dictionary provides several benefits in SQL data modeling:

a. Data Understanding: It helps users understand the structure and meaning of the data stored in the database. It provides a comprehensive view of the database schema, allowing stakeholders to gain insights into the available data elements and their relationships.
b. Data Consistency: The data dictionary ensures consistency by defining and enforcing data standards and rules across the database. It acts as a reference for data naming conventions, data types, and constraints, ensuring uniformity and integrity in data modeling and development processes.
c. Data Governance: It supports data governance initiatives by documenting data lineage, data ownership, and data usage. The data dictionary helps track the origin and transformation of data elements, facilitating compliance, auditing, and data quality efforts.
d. Application Development: Developers can leverage the data dictionary during application development. It provides a reference for accessing and manipulating data, including column names, data types, and relationships between tables. Developers can use the data dictionary to understand database constraints and optimize SQL queries.

e. Maintenance and Impact Analysis: The data dictionary assists in database maintenance activities, such as schema modifications, index creation, or table updates. It helps identify dependencies between database objects, enabling impact analysis before making changes.

76. Given a large dataset with millions of rows, how would you approach identifying and removing duplicates?

When dealing with a large dataset with millions of rows, identifying and removing duplicates can be a complex task. Here's an approach you can follow to tackle this challenge efficiently:

a. Identify the key columns: Determine the columns that define uniqueness in your dataset. These columns will be used to identify duplicates. For example, in a dataset of customer records, the key columns could be "Customer ID" or a combination of columns like "First Name," "Last Name," and "Email."
b. Sort the dataset: Sort the dataset based on the key columns identified in step 1. Sorting the data will help group the duplicates together, making it easier to identify and remove them.
c. Iterate through the dataset: Iterate through the sorted dataset and compare consecutive rows to identify duplicates based on the key columns. Depending on the programming language or tool you are using, you can implement this comparison using loops or built-in functions.
d. Remove duplicates: Once duplicates are identified, you can choose how to handle them based on your requirements. You have a few options: a. Delete duplicates in-place: If your dataset is stored in a database, you can directly delete the duplicate rows from the table using SQL statements. b. Create a new dataset without duplicates: If you want to preserve the original dataset, you can create a new dataset or table without the duplicate rows. You can store the non-duplicate rows in a separate table or export them to a new file. c. Update duplicates with merged data: In some cases, you may want to merge the data from duplicate rows and keep a single, consolidated record. You can update the duplicates with merged data and remove the remaining duplicate rows.
e. Validate the results: After removing duplicates, it is essential to validate the results to ensure accuracy. Perform checks and queries to verify that no duplicates remain in the dataset.
f. Optimize performance: When working with large datasets, efficiency is crucial. Consider using indexing, parallel processing, or leveraging specialized tools and frameworks that can handle large-scale duplicate identification and removal more efficiently.

77. How would you perform a time series analysis in SQL?

Performing time series analysis in SQL involves utilizing various SQL functions and techniques to analyze and extract insights from temporal data. Here's an overview of steps you can follow to perform time series analysis in SQL:

a. Prepare the data: Ensure your dataset includes a column with temporal information, such as a timestamp or date column. Make sure the data is properly formatted and consistent.
b. Explore and summarize the data: Begin by gaining an understanding of the data distribution and patterns. Use SQL aggregate functions like COUNT, MIN, MAX, AVG, and SUM to calculate summary statistics, identify trends, and detect outliers.
c. Grouping and aggregation: Depending on your analysis goals, you may need to group the data by time intervals, such as day, month, or year. Use SQL's GROUP BY clause to aggregate the data and calculate metrics within each time period. For example, you can calculate the average sales per month or the total revenue per day.
d. Window functions: SQL window functions provide powerful capabilities for time series analysis. They allow you to perform calculations over a sliding window of data. Functions like ROW_NUMBER, LAG, LEAD, and AVG with the OVER clause enable computations such as calculating moving averages, detecting trends, or comparing values with previous or future time periods.
e. Time-based filtering: Apply SQL's WHERE clause to filter the data based on specific time ranges or conditions. For example, you can extract data for a particular month, year, or a range of dates to focus on specific time periods of interest.
f. Seasonality and trends: SQL can help identify seasonality patterns and trends in time series data. Use techniques like Fourier analysis, autoregressive integrated moving average (ARIMA), or exponential smoothing models to detect and analyze seasonality, trends, and patterns in the data. These analyses may require more complex SQL queries or the use of specialized SQL extensions.
g. Visualization: While SQL itself is not a visualization tool, you can use SQL to extract and aggregate data, and then export it to other tools or libraries for visualization. SQL's output can be consumed by tools like Python's Matplotlib or libraries like Tableau for creating charts, graphs, and interactive dashboards to visualize time series patterns.

78. Given a dataset with missing values, how would you approach imputing those missing values?

When dealing with a dataset that contains missing values, there are several approaches you can take to impute those missing values. The choice of method depends on the nature of the data and the underlying assumptions. Here are a few commonly used approaches for imputing missing values:

a. Mean/Median/Mode imputation: Replace missing values with the mean, median, or mode of the respective feature. This method assumes that the missing values are missing at random and that the imputed values will not significantly distort the distribution of the data.
b. Forward fill/Backward fill: For time series or sequential data, you can use forward fill (or last observation carried forward) or backward fill (or next observation carried backward) to propagate the last observed value or the next observed value respectively to fill in the missing values.

c. Linear interpolation: For ordered data with a linear relationship, you can use linear interpolation to estimate missing values based on the values before and after the missing entry. This method assumes a linear relationship between the observed values.
d. Multiple imputation: Multiple imputation involves creating multiple imputed datasets by using techniques such as regression imputation, k-nearest neighbors imputation, or predictive modeling. Each imputed dataset is analyzed separately, and the results are combined to obtain a final estimate. This approach accounts for the uncertainty introduced by imputation.
e. Model-based imputation: Fit a model (such as regression, decision trees, or random forests) using the observed data and use that model to predict the missing values. The model can take into account other variables to make more accurate predictions.
f. Domain-specific imputation: Depending on the domain and the specific characteristics of the data, domain-specific knowledge can be used to impute missing values. For example, imputing missing geographic data based on known geographical relationships or using expert domain knowledge to estimate missing values.
g. Dropping rows/columns: In some cases, if the missing values are substantial or if imputation methods are not appropriate, you may choose to drop the rows or columns with missing values. However, this should be done carefully, considering the impact on the overall analysis and the potential loss of information.

79. Consider the tables given below and solve the queries.

Table – EmployeeDetails

EmpId	FullName	ManagerId	DateOfJoining	City
121	John Snow	321	01/31/2019	Toronto
321	Walter White	986	01/30/2020	California
421	Kuldeep Rana	876	27/11/2021	New Delhi

Table – EmployeeSalary

EmpId	Project	Salary	Variable
121	P1	8000	500
321	P2	10000	1000
421	P1	12000	0

Fetch all the employees who are not working on any project.

```
SELECT EmpId
FROM EmployeeSalary
WHERE Project IS NULL;
```

80. Write an SQL query to fetch employee names having a salary greater than or equal to 5000 and less than or equal to 10000.

```
SELECT FullName
FROM EmployeeDetails
WHERE EmpId IN
(SELECT EmpId FROM EmployeeSalary
WHERE Salary BETWEEN 5000 AND 10000);
```

a. The SELECT statement specifies that only the "FullName" column should be returned in the result set.
b. The FROM clause indicates that the data is retrieved from the "EmployeeDetails" table.
c. The WHERE clause includes a condition: "EmpId IN (SELECT EmpId FROM EmployeeSalary WHERE salary BETWEEN 5000 AND 10000)".
d. The subquery "(SELECT EmpId FROM EmployeeSalary WHERE salary BETWEEN 5000 AND 10000)" is used to retrieve the "EmpId" values from the "EmployeeSalary" table where the salary is between 5000 and 10000. The BETWEEN operator is used to specify a range, inclusive of both the lower and upper bounds. The subquery selects only the "EmpId" values that meet the specified condition.
e. The main query selects the "FullName" column from the "EmployeeDetails" table, but only for the rows where the "EmpId" exists in the result set of the subquery.

f. The result set includes the "FullName" values of employees whose "EmpId" matches a record in the "EmployeeSalary" table and whose salary falls between 5000 and 10000.

81. Write an SQL query to fetch all the Employee details from the EmployeeDetails table who joined in the Year 2020.

```
SELECT * FROM EmployeeDetails
WHERE YEAR(DateOfJoining) = '2020';
```

This SQL query retrieves all columns from the "EmployeeDetails" table where the "DateOfJoining" falls in the year 2020. Let's break it down step by step:

a. The SELECT statement specifies that all columns should be returned in the result set. The asterisk (*) is used as a wildcard to represent all columns.
b. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "EmployeeDetails" table.
c. The WHERE clause includes the condition "YEAR(DateOfJoining) = '2020'".
d. The function YEAR(DateOfJoining) is used to extract the year from the "DateOfJoining" column. It converts the date to a four-digit year format.
e. The condition compares the extracted year with the string '2020'. It checks if the year of the "DateOfJoining" column is equal to '2020'.
f. The result set includes all columns from the "EmployeeDetails" table for the rows where the "DateOfJoining" falls in the year 2020.

82. Write an SQL query to fetch all employee records from the EmployeeDetails table who have a salary record in the EmployeeSalary table.

```
SELECT * FROM EmployeeDetails E
WHERE EXISTS
(SELECT * FROM EmployeeSalary S
WHERE  E.EmpId = S.EmpId);
```

This SQL query retrieves all columns from the "EmployeeDetails" table where there is a matching record in the "EmployeeSalary" table based on the "EmpId" column. Let's break it down step by step:

a. The SELECT statement specifies that all columns should be returned in the result set. The asterisk (*) is used as a wildcard to represent all columns.
b. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "EmployeeDetails" table, and it is aliased as "E".
c. The WHERE clause includes the condition "EXISTS (SELECT * FROM EmployeeSalary S WHERE E.EmpId = S.EmpId)".
d. The subquery "(SELECT * FROM EmployeeSalary S WHERE E.EmpId = S.EmpId)" is used to check if there exists any record in the "EmployeeSalary" table that has the same

"EmpId" as the current row in the "EmployeeDetails" table. The subquery is executed for each row in the outer query. It selects all columns from the "EmployeeSalary" table (aliased as "S") where the "EmpId" matches the "EmpId" of the current row in the "EmployeeDetails" table (referred to as "E.EmpId").

e. The EXISTS keyword is used to check if the subquery returns any records. If there is at least one matching record, the condition is considered true.
f. The result set includes all columns from the "EmployeeDetails" table for the rows where there is a matching record in the "EmployeeSalary" table based on the "EmpId" column.

83. Write an SQL query to fetch the project-wise count of employees sorted by project's count in descending order.

```
SELECT Project, count(EmpId) EmpProjectCount
FROM EmployeeSalary
GROUP BY Project
ORDER BY EmpProjectCount DESC;
```

This SQL query retrieves the count of employees working on each project from a table called "EmployeeSalary" and sorts the result set in descending order based on the count. Let's break it down step by step:

a. The SELECT statement specifies the columns to be returned in the result set. In this case, it selects two columns: "project" and "count(EmpId)" aliased as "EmpProjectCount".
b. The "project" column represents the project name, and "count(EmpId)" calculates the number of employees (count of distinct "EmpId" values) working on each project.
c. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "EmployeeSalary" table.
d. The GROUP BY clause groups the rows based on the "project" column. This groups the rows with the same project name together.
e. The COUNT(EmpId) function is used in the SELECT statement to calculate the count of distinct "EmpId" values for each group. This gives the number of employees working on each project.
f. The ORDER BY clause sorts the result set based on the "EmpProjectCount" column in descending order. This arranges the projects in the result set starting from the project with the highest number of employees.

84. Write an SQL query to fetch duplicate records from EmployeeDetails (without considering the primary key – EmpId).

```
SELECT FullName, ManagerId, DateOfJoining, City, COUNT(*)
FROM EmployeeDetails
GROUP BY FullName, ManagerId, DateOfJoining, City
HAVING COUNT(*) > 1;
```

This SQL query retrieves employee details from a table called "EmployeeDetails" where there are duplicates based on specific columns. Let's break it down step by step:

a. The SELECT statement specifies the columns to be returned in the result set. In this case, it selects "FullName", "ManagerId", "DateOfJoining", "City", and "count(*)".
b. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "EmployeeDetails" table.
c. The GROUP BY clause groups the rows based on multiple columns: "FullName", "ManagerId", "DateOfJoining", and "City". This groups the rows with the same combination of values in these columns.
d. The HAVING clause filters the grouped results based on a condition. In this case, the condition is "count(*) > 1". It selects only those groups where the count of rows within each group is greater than 1.
e. The result set includes the selected columns and an additional column representing the count of rows within each group.

85. Write an SQL query to remove duplicates from a table without using a temporary table.

```
DELETE E1 FROM EmployeeDetails E1
INNER JOIN EmployeeDetails E2
WHERE E1.EmpId > E2.EmpId
AND E1.FullName = E2.FullName
AND E1.ManagerId = E2.ManagerId
AND E1.DateOfJoining = E2.DateOfJoining
AND E1.City = E2.City;
```

This SQL query performs a delete operation on a table called "EmployeeDetails" based on specific conditions involving a self-join. Let's break it down step by step:

a. The DELETE statement indicates that rows will be deleted from the table.
b. The "e1" and "e2" are table aliases assigned to the "EmployeeDetails" table to distinguish between two instances of the same table during the join operation.
c. The FROM clause specifies the table from which the data is retrieved for deletion. In this case, it is the "EmployeeDetails" table.

d. The INNER JOIN clause joins the "EmployeeDetails" table to itself based on matching conditions specified in the subsequent WHERE clause.
e. The WHERE clause includes multiple conditions that must be satisfied for a row to be deleted: "e1.EmpId > e2.EmpId": This condition checks if the "EmpId" of the first instance (e1) is greater than the "EmpId" of the second instance (e2). It ensures that only rows with higher employee IDs are considered for deletion. "e1.FullName = e2.FullName": This condition checks if the "FullName" of the first instance matches the "FullName" of the second instance. It ensures that only rows with the same full name are considered for deletion. "e1.ManagerId = e2.ManagerId": This condition checks if the "ManagerId" of the first instance matches the "ManagerId" of the second instance. It ensures that only rows with the same manager ID are considered for deletion. "e1.DateOfJoining = e2.DateOfJoining": This condition checks if the "DateOfJoining" of the first instance matches the "DateOfJoining" of the second instance. It ensures that only rows with the same date of joining are considered for deletion. "e1.City = e2.City": This condition checks if the "City" of the first instance matches the "City" of the second instance. It ensures that only rows with the same city are considered for deletion.
f. The delete operation will remove the rows from the "EmployeeDetails" table that satisfy all the conditions mentioned above.

86. Write SQL query to find the 3rd highest salary from a table without using the TOP/limit keyword.

Solution:

```
SELECT Salary
FROM EmployeeSalary Emp1
WHERE 2 = (
            SELECT COUNT( DISTINCT ( Emp2.Salary ) )
            FROM EmployeeSalary Emp2
            WHERE Emp2.Salary > Emp1.Salary
        )
```

This SQL query retrieves the salary of an employee from a table called "EmployeeSalary" based on a specific condition. Let's break it down step by step:

a. The SELECT statement specifies the column to be returned in the result set. In this case, it selects the "salary" column.

b. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "EmployeeSalary" table, and it is aliased as "emp1".
c. The WHERE clause applies a condition to filter the rows. The condition is "2 = (SELECT COUNT(DISTINCT(emp2.salary)) FROM EmployeeSalary emp2 WHERE emp2.salary > emp1.salary)".
d. The subquery "(SELECT COUNT(DISTINCT(emp2.salary)) FROM EmployeeSalary emp2 WHERE emp2.salary > emp1.salary)" is used to calculate the count of distinct salaries that are greater than the salary of the current row. The subquery is executed for each row in the outer query. It counts the number of distinct salaries from the "EmployeeSalary" table (aliased as "emp2") where the salary is greater than the salary of the current row (referred to as "emp1.salary").
e. The outer query compares the result of the subquery with the value 2. If the count of distinct salaries greater than the current salary is exactly 2, then the condition is satisfied.
f. The result set includes the salary column for the rows that meet the condition.

87. Consider the table:

id	movie	description	rating
1	War	thriller	8.9
2	Dhakkad	action	2.1
3	Gippi	boring	1.2
4	Dangal	wrestling	8.6
5	P.K.	Sci-Fi	9.1

Write an SQL query to report the movies with an odd-numbered ID and a description that is not "boring". Return the result table ordered by rating in descending order.

Solution:

```
select *
from cinema
where mod(id, 2) = 1 and description != 'boring'
order by rating DESC;
```

This SQL query retrieves data from a table called "cinema" based on specific conditions and sorts the results in descending order based on the "rating" column. Let's break it down step by step:

a. The SELECT statement specifies that all columns should be returned in the result set. The asterisk (*) is used as a wildcard to represent all columns.
b. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "cinema" table.
c. The WHERE clause filters the rows based on two conditions: "mod(id, 2) = 1": This condition checks if the remainder of the division of the "id" column by 2 is equal to 1. In other words, it selects only those rows where the "id" is an odd number. "description != 'boring'": This condition checks if the value in the "description" column is not equal to the string 'boring'. It excludes rows where the description is specifically marked as 'boring'.
d. The ORDER BY clause specifies the sorting order of the result set. In this case, it sorts the rows in descending order based on the "rating" column.

88. Consider the tables users and transactions:
Users table:

Account_number	name
12300001	Ram
12300002	Tim
12300003	Shyam

Transactions table:

trans_id	account_number	amount	transacted_on
1	12300001	8000	2022-03-01
2	12300001	8000	2022-03-01
3	12300001	-3000	2022-03-02
4	12300002	4000	2022-03-12
5	12300003	7000	2022-02-07
6	12300003	7000	2022-03-07
7	12300003	-4000	2022-03-11

Construct a SQL query to display the names and balances of people who have a balance greater than $10,000. The balance of an account is equal to the sum of the amounts of all transactions involving that account. You can return the result table in any order.
Solution:

```
SELECT u.name, SUM(t.amount) AS balance
FROM Users natural join Transactions t
GROUP BY t.account_number
HAVING balance> 10000;
```

This SQL query retrieves the names of users and the total balance of their transactions from a table called "users" and a table called "transactions". Let's break it down step by step:

a. The SELECT statement specifies the columns to be returned in the result set. In this case, it selects two columns: "u.name" and "sum(t.amount)" aliased as "balance".
b. The "u.name" represents the user's name, and "sum(t.amount)" calculates the total sum of the transaction amounts for each user.
c. The FROM clause includes the "users" table, which is used to retrieve user information, and the "transactions" table, which contains transaction records.
d. The NATURAL JOIN clause combines the "users" and "transactions" tables based on matching column names. It implicitly joins the tables using the common column(s) between them.

e. The GROUP BY clause groups the rows based on the "t.account_number" column, which represents the account number associated with each transaction. This allows the aggregation function, "sum(t.amount)", to calculate the total sum of transaction amounts for each account.
f. The HAVING clause filters the grouped results based on a condition. In this case, the condition is "balance > 10000". Since "balance" is an alias for the calculated sum of transaction amounts, this condition checks if the total balance for each account is greater than 10000.
g. The result set includes the user's name and the calculated balance for each account that satisfies the condition.

89. Consider the employee table:

Id	Name	Department	ManagerId
201	Ram	A	null
202	Naresh	A	201
203	Krishna	A	201
204	Vaibhav	A	201
205	Jainender	A	201
206	Sid	B	201

Write a SQL query that detects managers with at least 5 direct reports from the Employee table.

Solution:

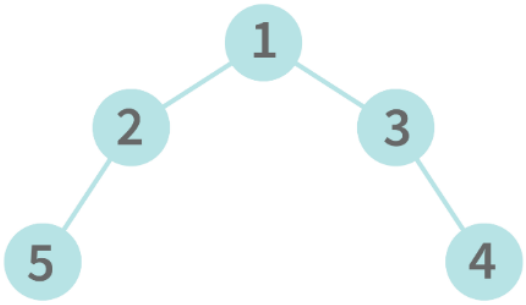
```sql
SELECT Name
FROM Employee
WHERE id IN
  (SELECT ManagerId
    FROM Employee
    GROUP BY ManagerId
    HAVING COUNT(DISTINCT Id) >= 5);
```

This SQL query retrieves the names of employees from a table called "employee" who have a specific number of subordinates. Let's break it down step by step:

a. The SELECT statement specifies the column to be returned in the result set. In this case, it selects the "name" column.
b. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "employee" table.
c. The WHERE clause filters the rows based on a condition. The condition is "id IN (SELECT managerId FROM employee GROUP BY managerId HAVING COUNT(DISTINCT id) >= 5)".
d. The subquery "(SELECT managerId FROM employee GROUP BY managerId HAVING COUNT(DISTINCT id) >= 5)" is used to identify the "managerId" values that meet a specific condition. The "GROUP BY managerId" groups the rows in the "employee" table based on the "managerId" column. The "HAVING COUNT(DISTINCT id) >= 5" condition specifies that only groups with a count of distinct "id" values greater than or equal to 5 should be considered.
e. The main query uses the "id IN" condition to check if the "id" value of an employee is present in the subquery result set. If it is, that means the employee is a manager with at least five subordinates.
f. The result set includes the "name" column, which represents the names of the employees who satisfy the condition.

90. Consider the tree table

id	parent_id
1	null
2	1
3	1
4	3
5	2



Write a SQL query to find and return the type of each of the nodes in the given tree according to the following output. You can return the result in any order.
Sample output:

id	type
1	Root
2	Inner
3	Inner
4	Leaf
5	Leaf

Solution:

```sql
SELECT
    id AS `Id`,
    CASE
        WHEN tree.id = (SELECT aliastree.id FROM tree aliastree WHERE aliastree.parent_id IS NULL)
          THEN 'Root'
        WHEN tree.id IN (SELECT aliastree.parent_id FROM tree aliastree)
          THEN 'Inner'
        ELSE 'Leaf'
    END AS Type
FROM
    tree
ORDER BY `Id`;
```

This SQL query retrieves data from a table called "tree" and assigns a type to each row based on certain conditions. It also renames the column "id" as "Id" in the result set. Let's break it down step by step:

a. The SELECT statement specifies the columns to be returned in the result set. In this case, it selects two columns: "id" (renamed as "Id") and the result of the CASE statement, which is aliased as "Type".
b. The CASE statement is used to perform conditional logic and determine the type of each row based on the values in the "tree" table.
c. The first condition in the CASE statement checks if the "id" value of the current row is equal to the "id" value obtained from a subquery. This subquery retrieves the "id" value from the "tree" table where the "parent_id" column is NULL. If the condition is true, it means that the current row represents a root node, and the value 'Root' is assigned to the "Type" column.
d. The second condition in the CASE statement checks if the "id" value of the current row is present in the subquery result set. This subquery retrieves the "parent_id" values from the "tree" table. If the condition is true, it means that the current row represents an inner node (i.e., it has a parent node), and the value 'Inner' is assigned to the "Type" column.
e. If neither of the above conditions is true, the ELSE part of the CASE statement is executed, assigning the value 'Leaf' to the "Type" column. This indicates that the current row represents a leaf node (i.e., it has no child nodes).
f. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "tree" table.
g. The ORDER BY clause is used to sort the result set based on the "Id" column in ascending order.

91. Consider the student table:

id	student
1	Ram
2	Shyam
3	Vaibhav
4	Govind
5	Krishna

Sample output:

id	student
1	Shyam
2	Ram
3	Govind
4	Vaibhav
5	Krishna

You need to write a query that swaps alternate students' seat id and returns the result. If the number of students is odd, you can leave the seat id for the last student as it is.

Solution:

```
SELECT
    CASE WHEN MOD(id, 2) != 0 AND counts != id THEN id + 1 -- for odd ids
         WHEN MOD(id, 2) != 0 AND counts = id THEN id -- special case for last seat
         ELSE id - 1 -- For even ids
         END as id,
    student
FROM
seat, (SELECT COUNT(*) as counts
       FROM seat) AS seat_count
ORDER by id;
```

This SQL query retrieves data from a table called "seat" and performs some calculations based on the values in the table. Let's break it down step by step:

a. The query starts with the SELECT statement, which specifies the columns to be returned in the result set. In this case, the query selects two columns: "id" and "student".
b. The CASE statement is used to evaluate conditions and return different values based on those conditions. It is used here to calculate the modified "id" values for each row in the result set.
c. The first condition in the CASE statement is "MOD(id, 2) != 0 AND counts != id". This condition checks if the "id" is an odd number and if it's not equal to the "counts" value

(which represents the total count of seats in the "seat" table). If the condition is true, the "id + 1" is returned as the modified "id" value.

d. The second condition in the CASE statement is "MOD(id, 2) != 0 AND counts = id". This condition checks if the "id" is an odd number and if it's equal to the "counts" value. If the condition is true, the original "id" is returned as the modified "id" value. This is a special case for the last seat, where the modified "id" remains the same.
e. The ELSE clause of the CASE statement is triggered when none of the previous conditions are met. In this case, if the "id" is an even number, the "id - 1" is returned as the modified "id" value.
f. The "FROM" clause specifies the tables involved in the query. In this case, it uses the "seat" table and a subquery that calculates the total count of seats in the "seat" table. The subquery aliased as "seat_count" returns a single row with a single column named "counts" containing the count of seats.
g. The ORDER BY clause is used to sort the result set based on the "id" column in ascending order.

92. Consider the stadium table:

id	date_visited	count_people
1	2022-03-01	6
2	2022-03-02	102
3	2022-03-03	135
4	2022-03-04	90
5	2022-03-05	123
6	2022-03-06	115
7	2022-03-07	101
8	2022-03-09	235

Sample output:

id	date_visited	count_people
5	2022-03-05	123
6	2022-03-06	115
7	2022-03-07	101
8	2022-03-09	235

Construct a SQL query to display records that have three or more rows of consecutive ids and a total number of people higher than or equal to 100. Return the result table in ascending order by visit date.

Solution:

```
select distinct t1.*
from stadium t1, stadium t2, stadium t3
where t1.count_people >= 100 and t2.count_people >= 100 and t3.count_people >= 100
and
(
    (t1.id - t2.id = 1 and t1.id - t3.id = 2 and t2.id - t3.id =1)
    or
    (t2.id - t1.id = 1 and t2.id - t3.id = 2 and t1.id - t3.id =1)
    or
    (t3.id - t2.id = 1 and t2.id - t1.id =1 and t3.id - t1.id = 2)
)
order by t1.id;
```

This SQL query selects distinct rows from a table called "stadium" based on certain conditions. Let's break it down step by step:

a. The SELECT statement specifies that all columns (denoted by "t1.*") from the table "stadium" should be returned in the result set.
b. The FROM clause lists the table "stadium" multiple times, with aliases "t1", "t2", and "t3". This is done to create three separate instances of the "stadium" table to compare the data between different rows.
c. The WHERE clause contains conditions that filter the rows based on the "count_people" column. It ensures that the "count_people" value is greater than or equal to 100 for all three instances of the "stadium" table (t1, t2, t3).

d. The main part of the query is the set of conditions enclosed within the parentheses after the "and" keyword. These conditions compare the "id" values of the different instances of the "stadium" table. a. The first condition checks if the "id" of t1 is one less than t2, the "id" of t1 is two less than t3, and the "id" of t2 is one less than t3. b. The second condition checks if the "id" of t2 is one less than t1, the "id" of t2 is two less than t3, and the "id" of t1 is one less than t3. c. The third condition checks if the "id" of t3 is one less than t2, the "id" of t2 is one less than t1, and the "id" of t3 is two less than t1. These conditions ensure that the "id" values of the selected rows form a sequence where the difference between consecutive "id" values is either 1 or 2.
e. The ORDER BY clause is used to sort the result set based on the "id" column of the t1 instance of the "stadium" table in ascending order.

93. What will be the output of the below query, given an Employee table having 10 records?

```
BEGIN TRAN
TRUNCATE TABLE Employees
ROLLBACK
SELECT * FROM Employees
```

Solution:
This query will return 10 records as TRUNCATE was executed in the transaction. TRUNCATE does not itself keep a log but BEGIN TRANSACTION keeps track of the TRUNCATE command.

94. Imagine a single column in a table that is populated with either a single digit (0-9) or a single character (a-z, A-Z). Write a SQL query to print ‘Fizz’ for a numeric value or ‘Buzz’ for alphabetical value for all values in that column.

Example:

```
['d', 'x', 'T', 8, 'a', 9, 6, 2, 'V']
```

...should output:

```
['Buzz', 'Buzz', 'Buzz', 'Fizz', 'Buzz','Fizz', 'Fizz', 'Fizz',
'Buzz']
```

Solution:

```
SELECT col, case when upper(col) = lower(col) then 'Fizz' else 'Buzz'
end as FizzBuzz from table;
```

This SQL query selects data from a table named "table" and performs a transformation on a column called "col" using a CASE statement. Let's break it down step by step:

a. The SELECT statement specifies the columns to be returned in the result set. In this case, it selects two columns: "col" and the result of the CASE statement, which is aliased as "FizzBuzz".
b. The CASE statement is used to perform conditional logic and return different values based on the conditions. In this query, it checks if the uppercase version of the "col" value is equal to the lowercase version of the same value. This condition is determined by the comparison "upper(col) = lower(col)".
c. If the condition is true, meaning the "col" value contains only characters that are not affected by case (e.g., numbers or symbols), then the CASE statement returns 'Fizz' as the value for the "FizzBuzz" column.
d. If the condition is false, meaning the "col" value contains at least one character that has different uppercase and lowercase representations, then the CASE statement returns 'Buzz' as the value for the "FizzBuzz" column.
e. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "table" table.

95. Consider the table:

ID	C1	C2	C3
1	Red	Yellow	Blue
2	NULL	Red	Green
3	Yellow	NULL	Violet

Print the rows which have ‘Yellow’ in one of the columns C1, C2, or C3, but without using OR

Solution:

```
SELECT * FROM table
WHERE 'Yellow' IN (C1, C2, C3)
```

This SQL query selects all rows from a table named "table" where the value 'YELLOW' is present in any of the three columns C1, C2, or C3. Let's break it down step by step:

a. The SELECT statement specifies that all columns (*) should be returned in the result set. This means that all columns of the "table" table will be included in the query result.
b. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "table" table.
c. The WHERE clause is used to filter the rows based on a condition. The condition in this query is the expression 'YELLOW' IN (C1, C2, C3).
d. The IN operator is used to check if a value is present in a list of values. In this case, it checks if the value 'YELLOW' is present in the list of values formed by the columns C1, C2, and C3.
e. If 'YELLOW' is found in any of the three columns (C1, C2, or C3) for a particular row, that row will be included in the query result.

96. Consider the mass table:

weight
5.67
34.567
365.253
34

Write a query that produces the output:

weight	kg	gms
5.67	5	67
34.567	34	567
365.253	365	253
34	34	0

Solution:

```
select weight, trunc(weight) as kg,
nvl(substr(weight - trunc(weight), 2), 0) as gms
from mass_table;
```

This SQL query retrieves data from a table named "mass_table" and performs calculations on a column called "weight". Let's break it down step by step:

a. The SELECT statement specifies the columns to be returned in the result set. In this case, it selects three columns: "weight", "trunc(weight) as kg", and "nvl(substr(weight - trunc(weight), 2), 0) as gms".
b. The "weight" column represents the original weight values from the "mass_table" table.
c. The "trunc(weight) as kg" expression calculates the integer part of the "weight" column, effectively truncating any decimal places and returning the result as "kg".
d. The "nvl(substr(weight - trunc(weight), 2), 0) as gms" expression calculates the fractional part of the "weight" column and returns it as "gms". The "weight - trunc(weight)" expression subtracts the truncated part of the weight from the original weight, leaving only the decimal part. The "substr(..., 2)" function extracts the substring starting from the second character, effectively removing the leading decimal point. The "nvl(..., 0)" function is used to handle cases where the decimal part is null. If the decimal part is null, it substitutes it with 0.
e. The FROM clause specifies the table from which the data is retrieved. In this case, it is the "mass_table" table.

97. You are given the following table containing historical employee salaries for company XYZ:
Table: EmployeeSalaries

employee_ID	salary	year
1	80000	2020
1	70000	2019
1	60000	2018
2	65000	2020
2	65000	2019
2	60000	2018
3	65000	2019
3	60000	2018

Given the above table, can you write a SQL query to return the employees who have received at least 3 year over year raises based on the table's data?

Solution:

```sql
SELECT
 a.employee_ID as employee_ID
FROM
 (SELECT
     employee_ID,
     salary,
     LEAD(salary) OVER (PARTITION BY employee_ID ORDER BY year DESC) as previous_year_sal
  FROM Employee ) a
WHERE a.salary > a.previous_year_sal
GROUP BY employee_ID
HAVING count(*) = 2;
```

This SQL query retrieves the "employee_ID" values from a table called "Employee" based on certain conditions and grouping criteria. Let's break it down step by step:

a. The SELECT statement specifies the column to be returned in the result set. In this case, it selects the column "employee_ID" and aliases it as "employee_ID".
b. The FROM clause starts with a subquery enclosed in parentheses. This subquery retrieves data from the "Employee" table and assigns it an alias "a".
c. Inside the subquery, another SELECT statement is used to retrieve columns: "employee_ID", "salary", and the use of the LEAD() function.
d. The LEAD(salary) OVER (PARTITION BY employee_ID ORDER BY year DESC) expression calculates the salary of the employee in the previous year based on the "salary" column. The LEAD() function retrieves the value of "salary" from the next row within the same "employee_ID" partition, ordered by the "year" column in descending order.
e. The subquery result set, aliased as "a", includes the columns "employee_ID", "salary", and "previous_year_sal". The WHERE clause filters the rows based on a condition: "a.salary > a.previous_year_sal". It selects only those rows where the "salary" value is greater than the "previous_year_sal" value.
f. The GROUP BY clause groups the rows by the "employee_ID" column.
g. The HAVING clause specifies a condition that is applied after the grouping. In this case, "count(*) = 2" means that only the groups having exactly two rows will be included in the result set. This ensures that the employee has records for two consecutive years where the salary has increased.

98. Consider the employee table given below, write a query to identify the employee who has the third-highest salary from the given employee table.

Name	Salary
Tarun	70,000
Sabid	60,000
Adarsh	30,000
Vaibhav	80,000

Solution:

```
WITH CTE AS
(
    SELECT Name, Salary, RN = ROW_NUMBER() OVER (ORDER BY Salary DESC) FROM EMPLOYEE
)
SELECT Name, Salary FROM CTE WHERE RN =3
```

This SQL query retrieves the name and salary of the employee who has the third-highest salary from a table called "employee". It uses a common table expression (CTE) to calculate a row number for each employee based on their salary in descending order. Let's break it down step by step:

a. The query starts with a "WITH" clause, which defines a CTE named "cte" (short for Common Table Expression). The CTE is essentially a temporary result set that can be used within the query. In this case, the CTE selects the "name", "salary", and assigns a row number (RN) using the ROW_NUMBER() function. The ROW_NUMBER() function generates a unique number for each row based on the specified order, which is "ORDER BY salary desc" in this query. The highest salary will have a row number of 1, the second highest will have a row number of 2, and so on.
b. After defining the CTE, the main query is executed. It selects the "name" and "salary" columns from the CTE.
c. The WHERE clause filters the results based on the condition "RN = 3", which means it retrieves only the rows where the row number is equal to 3. This ensures that only the employee with the third-highest salary is returned.