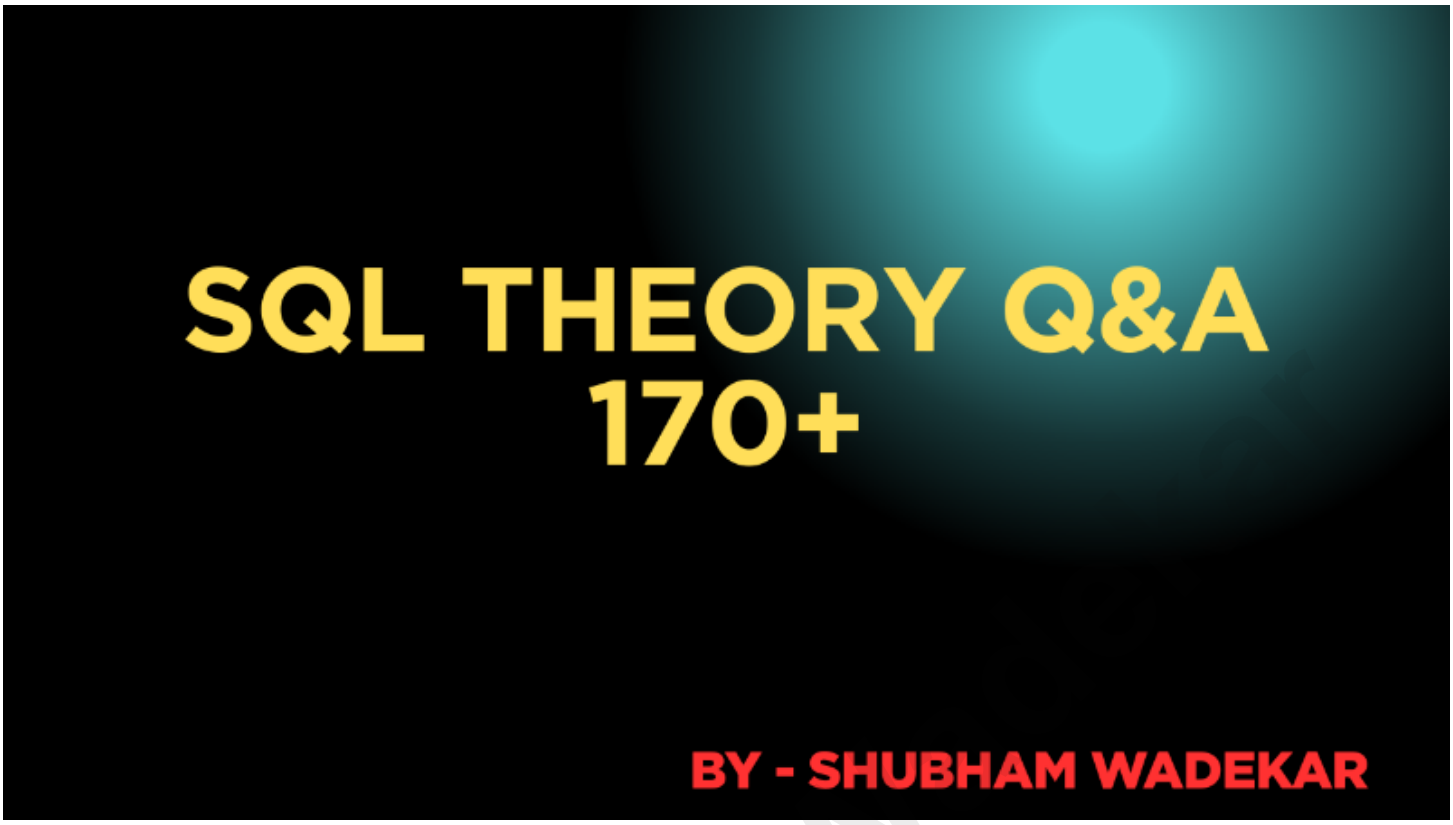

Table of Contents

No.	Questions
28	Why do we need triggers?
29	What are the different types of triggers?
30	What is a view in the database?
31	Why do I need views in a database?
32	What is the difference between Primary key and unique key?
33	How can you increase SQL performance?
34	What is the use of OLAP?
35	What is a measure in OLAP?
36	What are dimensions in OLAP?
37	What are levels in dimensions?
38	What are fact tables and dimension tables in OLAP?
39	What is DTS?
40	What is fill factor?
41	What is RAID and how does it work?
42	Difference between @@IDENTITY, SCOPE_IDENTITY() and IDENT_CURRENT
43	Difference between char, varchar and nvarchar in SQL Server
44	What is the difference between DELETE TABLE and TRUNCATE TABLE commands?
45	If locking is not implemented, what issues can occur?
46	What are different transaction levels in SQL Server?
47	What are the different locks in SQL Server?
48	Can we suggest locking hints to SQL Server?
49	What is LOCK escalation?
50	What are the different ways of moving data between databases in SQL Server?
51	What is the difference between HAVING CLAUSE and WHERE CLAUSE?
52	What is the difference between UNION and UNION ALL SQL syntax?
53	What are the different types of triggers in SQL Server?
54	If we have multiple AFTER Triggers on table how can we define the sequence of the triggers?
55	What is SQL injection?
56	What is the difference between Stored Procedure (SP) and User Defined Function (UDF)?
57	How can you raise custom errors from stored procedure?
58	What is DBCC?
59	What is the purpose of Replication?
60	What are the different types of replication supported by SQL Server?
61	What is BCP utility in SQL Server?

No.	Questions
62	What is a Cursor?
63	What are local and global variables and their differences?
64	What is an index?
65	Why do I need an index in a database?
66	What is a query in a database?
67	What are query types in a database?
68	What is a join in SQL Server?
69	What are different types of joins in SQL Server?
70	What is Self-Join?
71	What is Cross-Join?
72	What is user defined functions?
73	What are all types of user defined functions?
74	What is collation?
75	What are all different types of collation sensitivity?
76	Advantages and Disadvantages of Stored Procedure?
77	What is Online Transaction Processing (OLTP)?
78	What is CLAUSE?
79	What is recursive stored procedure?
80	What is Union, minus and Intersect commands?
81	What is an ALIAS command?
82	What is the difference between TRUNCATE and DROP statements?
83	What are aggregate and scalar functions?
84	What is an inner join in SQL?
85	What is an outer join in SQL?
86	What is full join in SQL?
87	What is left join in SQL Server?
88	What is a right join in SQL Server?
89	What is database engine in SQL Server?
90	What are the Analysis Services in SQL Server?
91	What are the integration services in SQL Server?
92	What are the data quality services in SQL Server?
93	What are the reporting services in SQL Server?
94	What are the master data services in SQL Server?
95	What is replication in SQL Server?

No.	Questions
96	How do I select data from an SQL Server table?
97	What is a check in SQL?
98	What is a default in SQL?
99	How to create a database using SQL?
100	What is a constraint in SQL?
No.	Questions
101	What is data Integrity?
102	What is Auto Increment?
103	What is the difference between Cluster and Non-Cluster Index?
104	What is Datawarehouse?
105	How do I define constraints in SQL?
106	What is the meaning of Not Null in SQL?
107	How to alter a table schema in SQL Server?
108	How to create index in SQL Server?
109	How to get unique records in SQL?
110	How to create a date column in SQL Server?
111	What is ACID fundamental? What are transactions in SQL SERVER?
112	What is a candidate key?
113	How do GROUP and ORDER BY Differ?
114	Compare SQL & PL/SQL?
115	What is BCP? When is it used?
116	When is the UPDATE_STATISTICS command used?
117	Explain the steps needed to Create the scheduled job?
118	When are we going to use truncate and delete?
119	Explain correlated query work?
120	When is the Explicit Cursor Used?
121	Find What is Wrong in this Query?
122	Write the Syntax for STUFF function in an SQL server?
123	Name some commands that can be used to manipulate text in T-SQL code.
124	What are the three ways that Dynamic SQL can be executed?
125	In what version of SQL Server were synonyms released? How do synonyms work and explain its use cases?
126	If you are a SQL Developer, how can you delete duplicate records in a table with no primary key?
127	Is it possible to import data directly from T-SQL commands without using SQL Server Integration Services? If so, what are the commands?

No.	Questions
128	What is the native system stored procedure to execute a command against all databases?
129	How can a SQL Developer prevent T-SQL code from running on a production SQL Server?
130	How do you maintain database integrity where deletions from one table will automatically cause deletions in another table?
131	What port does SQL server run on?
132	What is the SQL CASE statement used for? Explain with an example?
133	What are the risks of storing a hibernate-managed object in cache? How do you overcome the problems?
134	When is the use of UPDATE_STATISTICS command?
135	What is SQL Profiler?
136	What command using Query Analyzer will give you the version of SQL server and operating system?
137	What does it mean to have QUOTED_IDENTIFIER ON? What are the implications of having it OFF?
138	What is the STUFF function and how does it differ from the REPLACE function in SQL?
139	How to get @@ERROR and @@ROWCOUNT at the same time?
140	What is de-normalization in SQL database administration? Give examples?
141	Can you explain about buffer cache and log Cache in SQL Server?
142	Describe how to use Linked Server?
143	Explain how to send email from SQL database?
144	How to make remote connection in database?
145	What is the purpose of OPENXML clause SQL server stored procedure?
146	How to store pdf file in SQL Server?
147	Explain the use of keyword WITH ENCRYPTION. Create a Store Procedure with Encryption?
148	What is lock escalation?
149	What is Failover clustering overview?
150	What is Builtin/Administrator?
151	What XML support does the SQL server extend?
152	Difference between Primary Key and Foreign Key?
153	SQL Query to find second highest salary of Employee?
154	SQL Query to find Max Salary from each department?
155	Write SQL Query to display the current date?
156	Write an SQL Query to check whether date passed to Query is the date of given format or not?
157	Write an SQL Query to print the name of the distinct employee whose DOB is between 01/01/1960 to 31/12/1975?
158	Write an SQL Query find number of employees according to gender whose DOB is between 01/01/1960 to 31/12/1975?
159	Write an SQL Query to find an employee whose Salary is equal or greater than 10000?
160	Write an SQL Query to find name of employee whose name Start with 'M'?
161	Find all Employee records containing the word "Joe", regardless of whether it was stored as JOE, Joe, or joe?
162	Write an SQL Query to find the year from date?

No.	Questions
163	How can you create an empty table from an existing table?
164	How to fetch common records from two tables?
165	How to fetch alternate records from a table?
166	How to select unique records from a table?
167	What is the command used to fetch first 5 characters of the string?
168	Which operator is used in query for pattern matching?
169	Write SQL Query to find duplicate rows in a database? and then write SQL query to delete them?
170	There is a table which contains two column Student and Marks, you need to find all the students, whose marks are greater than average marks i.e. list of above average students.
171	How do you find all employees which are also manager?
172	You have a composite index of three columns, and you only provide the value of two columns in WHERE clause of a select query? Will Index be used for this operation?
173	What is the default join in SQL? Give an example query?
174	Describe all the joins with examples in SQL?
175	What is Union and Union All ? Explain the differences?

SQL Interview Questions & Answers

What is a database?

A **database** is described as an organized collection of **data**. It is a system that allows you to store, manage, and retrieve data efficiently. A database consists of various objects such as schemas, tables, queries, reports, views, and more.

Syntax:

```
CREATE DATABASE DatabaseName;
```

Example:

```
CREATE DATABASE Student;
```

Alternatively, you can create a database using the **Design/Wizard** form by right-clicking on the **DATABASE** option and selecting **New Database**.

What is DBMS?

A **Database Management System (DBMS)** is a software program that controls the creation, maintenance, and use of a database. It serves as a **File Manager**, handling data in a structured database format rather than storing it in traditional file systems. The DBMS ensures efficient data storage, retrieval, and manipulation while also managing data security, integrity, and consistency.

What is SQL?

Structured Query Language (SQL) is a programming language specifically designed for managing and manipulating data in **Relational Database Management Systems (RDBMSs)**. SQL is an **International Organization for Standardization (ISO)** standard, providing a standardized way to interact with databases. In an RDBMS, data is organized into **tables**, with each table consisting of **rows** and **columns**.

Example of SQL Server 2014 SQL Format:

```
CREATE DATABASE ExampleDB;
```

Example of Oracle SQL Format:

```
CREATE DATABASE ExampleDB;
```

Output: The database is created successfully, and you can now interact with it for further operations like inserting, updating, and querying data.

What is PL/SQL?

PL/SQL (Procedural Language/SQL) is Oracle's procedural extension to SQL. It allows you to write code that combines SQL queries with procedural constructs like loops and conditional statements to create complex business logic.

Control Statements in PL/SQL:

Control statements in PL/SQL are used to manage the flow of execution based on certain conditions. These are crucial for implementing decision-making processes in your code.

Here are the main types of control statements in PL/SQL:

- **If Statement**
- **If-THEN-ELSE**
- **Nested If**
- **Branching with Logical Connectivity (AND, OR)**
- **While Loop**
- **For Loop**

These control statements allow you to build logic that reacts to different conditions or iterates over data, offering more flexibility than standard SQL queries.

What is the difference between SQL and PL/SQL?

SQL (Structured Query Language):

- SQL is a language used for querying and manipulating data in a relational database.
- It supports simple control structures like **IF/ELSE**.
- SQL can be used to interact with databases via **ADO.NET**.
- It executes one line of code at a time.
- SQL is platform-specific (primarily works on Windows).

PL/SQL (Procedural Language/SQL):

- PL/SQL is Oracle's procedural extension to SQL, allowing more complex logic and control structures.
- It executes **blocks of code** rather than individual lines.
- PL/SQL supports **deep control statements** (loops, conditional branching, etc.).
- It can run on both **UNIX** and **Windows**.
- PL/SQL includes object-oriented programming concepts like **encapsulation**, **function overloading**, and **information hiding** (but not inheritance).

What is RDBMS?

RDBMS (Relational Database Management System):
An RDBMS is a database management system that uses a relational model to store and manage data. It possesses the following characteristics:

- **Write-intensive operations:** RDBMS systems are often used in transaction-oriented applications, where data is frequently written to.
- **Data in flux or historical data:** RDBMS can handle frequently changing data or store vast amounts of historical data for analysis or mining.
- **Application-specific schema:** Each application in an RDBMS has a unique schema, tailored to the specific needs of that application.
- **Complex data models:** The relational model is suited for handling sophisticated data structures involving many tables, foreign key relationships, and complex join operations.
- **Data integrity:** RDBMS systems ensure data integrity through features like rollback operations, referential integrity, and transaction handling.

What is a database table?

Database Table:
A database table stores data in the form of rows and columns. Each table represents an entity and consists of records (rows) and attributes (columns). A table is a permanent structure in a database that remains until it is deleted.

Syntax:

1. **Create Table:**

   ```
   CREATE TABLE TableName (ID INT, NAME VARCHAR(30));
   ```

2. **Drop Table:**

   ```
   DROP TABLE TableName;
   ```

3. **Select Data:**

```
SELECT * FROM TableName;
```

What is a query?

Query:
A database query is a code written to retrieve information from a database. It is designed to match the expectations of the result set. Essentially, a query is a question posed to the database to fetch specific data based on certain conditions.

What is a subquery?

Subquery:
A subquery is a query nested within another query. The outer query is known as the **main query**, and the inner query is called the **subquery**. The subquery is executed first, and its result is passed on to the main query for further processing or filtering.

What are the types of subquery?

There are two types of subqueries:

1. **Correlated Subquery:**
   A correlated subquery cannot be considered as an independent query because it refers to columns in a table that is part of the main query's `FROM` clause. It depends on the outer query for its execution.
2. **Non-Correlated Subquery:**
   A non-correlated subquery is independent and can be executed on its own. The result of the subquery is then used in the main query to perform operations like filtering or comparison.

How to create a table in SQL?

SQL provides an organized way to create tables with the following syntax:

```
CREATE TABLE TableName (
    columnName1 datatype,
    columnName2 datatype
);
```

Example of creating a simple table:

```
CREATE TABLE Info (
    Name VARCHAR(20),
    BirthDate DATE,
    Phone NVARCHAR(12),
    City VARCHAR(20)
);
```

What are tables and fields?

A table is a set of data organized into columns and rows. The columns are vertical and represent the fields, while the rows are horizontal and represent records. A table has a specified number of columns (fields) but can have any number of rows (records).

How to delete a table in SQL Server?

To delete data records from a database table or to delete an existing table, you can use the following methods:

Syntax to delete all records from a table:

```
DELETE FROM TableName;
```

For example, to delete all records from the "info" table:

```
DELETE FROM info;
```

Syntax to drop (delete) the table itself:

```
DROP TABLE TableName;
```

For example, to delete the "info" table entirely:

```
DROP TABLE info;
```

How to update a database table using SQL?

To update an existing table, we use the SQL command `UPDATE`. This command updates the records based on the user's specified conditions.

Syntax:

```
UPDATE TableName
SET ColumnName = NewData
WHERE Condition;
```

For example, to update the "City" column to 'Baroda' where the `id` is 2:

```
UPDATE info
SET City = 'Baroda'
WHERE id = 2;
```

What is a database relationship?

A database relationship is created by linking a column in one table with a column in another table. There are four different types of relationships that can be established between tables:

1. One to One Relationship

In a one-to-one relationship, one row in a table is associated with one row in another table.

2. Many to One Relationship

In a many-to-one relationship, many rows in one table are associated with a single row in another table.

3. Many to Many Relationship

In a many-to-many relationship, rows from two tables can be associated with multiple rows in the other table. A third table, often called a "junction" or "bridge" table, is used to store the relationships between the two tables.

4. One to Many Relationship

In a one-to-many relationship, a single row in one table is related to many rows in another table. This is the reverse of the "many to one" relationship.

One to Many & Many to One Relationship:
In a one-to-many relationship, a single value in one table can have one or more corresponding dependent values in another table.

Many to Many Relationship:
A bridge table is used to establish a many-to-many relationship, storing the common information between the two related tables.

What is a primary key of a database?

A **primary key** is a column (or a set of columns) in a database table that uniquely identifies each record in that table. The primary key ensures that the values in the key column(s) are **unique** and that **null values** are not allowed.

- **Uniqueness**: Each value in the primary key column(s) must be unique across all rows in the table.
- **No Null Values**: A primary key column cannot have NULL values.

A table can only have **one primary key**, but it can consist of **multiple columns** (called a composite key).

Example:

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    StudentName VARCHAR(50),
    Age INT
);
```

In the above example, `StudentID` is the primary key of the `Students` table, ensuring that each student has a unique identifier.

What is a unique key?

A **unique key** constraint ensures that all values in a column or a combination of columns are distinct from each other, meaning no two rows can have the same value for the unique key column(s). Unlike the **primary key**, a unique key allows **NULL values** (though only one NULL value is allowed per column in some database systems). While a primary key automatically enforces uniqueness and does not allow NULLs, a unique key constraint provides uniqueness without imposing the same restrictions.

- A **primary key** can only be defined once per table.
- Multiple **unique keys** can be defined on a single table.

Example:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Email VARCHAR(100) UNIQUE,
    Name VARCHAR(50)
);
```

In this example, `EmployeeID` is the primary key, ensuring each employee has a unique ID, while `Email` is defined with a unique constraint, ensuring that each email address is also distinct.

What is a foreign key of a database?

A **foreign key** is a column or a set of columns used to create a relationship between two tables. It is a reference in a child table that points to the primary key in a parent table. This ensures referential integrity by ensuring that a foreign key value in the child table must exist in the parent table.

- A table can have multiple foreign keys.
- Each foreign key in a child table refers to a primary key in another table.

Example:

Consider the following two tables:

- **CUSTOMER**: Contains customer details.

  ```
  CUSTOMER {Cust_ID, Cust_Name, Age, ContactNo, Gender, Address}
  ```

- **VENDOR**: Contains vendor details, and `Cust_ID` is a foreign key that references `Cust_ID` in the **CUSTOMER** table.

  ```
  VENDOR {Vend_ID, Vend_Name, Cust_ID}
  ```

In this case, the `Cust_ID` in the **VENDOR** table is a foreign key that refers to the `Cust_ID` in the **CUSTOMER** table.

Syntax:

To define a foreign key constraint when creating a table:

```
CREATE TABLE table_name (
    Col1 datatype NOT NULL,
    Col2 datatype NOT NULL,
    Col3 datatype NOT NULL,
    CONSTRAINT FK_Column
    FOREIGN KEY (Col1, Col2, Col3)
    REFERENCES parent_table (Col1, Col2, Col3)
);
```

This defines a foreign key constraint where `Col1`, `Col2`, and `Col3` in the child table must reference `Col1`, `Col2`, and `Col3` in the parent table.

Single-column level:

If the foreign key is defined on a single column:

```
CREATE TABLE Vendor (
    Vend_ID INT PRIMARY KEY,
    Vend_Name VARCHAR(100),
    Cust_ID INT,
    FOREIGN KEY (Cust_ID) REFERENCES Customer(Cust_ID)
);
```

Here, the `Cust_ID` column in the **VENDOR** table is a foreign key referencing the `Cust_ID` in the **CUSTOMER** table.

What is database normalization?

Database normalization is the process of organizing the fields and tables of a relational database to reduce redundancy and dependency. The goal is to divide large tables into smaller, more manageable ones, minimizing redundancy and improving the consistency of the data. Normalization is considered a bottom-up technique for database design.

The normalization process involves several stages, each improving the organization of the data. The most common normal forms are:

1. First Normal Form (1NF)

- Ensures that the table has only atomic (indivisible) values.
- Each column contains unique values, and each column's value must be of a single type.
- Eliminates duplicate rows.

2. Second Normal Form (2NF)

- Achieved by meeting the requirements of 1NF.
- Ensures that all non-key attributes are fully dependent on the primary key (removes partial dependency).
- If a table has a composite key, all non-key attributes must depend on the entire composite key.

3. Third Normal Form (3NF)

- Achieved by meeting the requirements of 2NF.
- Ensures that no transitive dependencies exist, i.e., non-key attributes should not depend on other non-key attributes.

4. Boyce-Codd Normal Form (BCNF)

- A more strict version of 3NF.
- Every determinant must be a candidate key. This eliminates anomalies that might still exist in 3NF.

5. Fourth Normal Form (4NF)

- Achieved by meeting the requirements of BCNF.
- Deals with multi-valued dependencies by ensuring that a record cannot have multiple sets of values that are independent of each other.

6. Fifth Normal Form (5NF)

- Achieved by meeting the requirements of 4NF.
- Ensures that no facts are lost in the decomposition of tables, aiming to eliminate redundancy by using join dependencies.

7. Sixth Normal Form (6NF)

- Achieved by meeting the requirements of 5NF.
- Deals with temporal data and ensures that the database design can handle data that evolves over time without causing redundancy.

Each normal form builds upon the previous one to reduce redundancy and improve the integrity of the database structure.

What are database normalization forms?

Normalization is the process of organizing data into related tables, eliminating redundancy, and improving the integrity and performance of queries. It involves dividing a database into multiple tables and establishing relationships between them.

The different **normalization forms** are as follows:

1. First Normal Form (1NF):

- **Goal**: Remove duplicate columns.
- **Action**: Ensure that each column contains only atomic (indivisible) values, and eliminate duplicate rows. Each column must have unique names, and each column's value should be of a single type.

- **Result**: Data is organized into a table without repeating groups.

2. Second Normal Form (2NF):

- **Goal**: Remove partial dependencies.
- **Action**: Meet all requirements of 1NF and place subsets of data into separate tables. Establish relationships between these tables using primary keys. Ensure that each non-key attribute is fully functionally dependent on the primary key.
- **Result**: No non-key column depends only on part of a composite key.

3. Third Normal Form (3NF):

- **Goal**: Remove transitive dependencies.
- **Action**: Meet all requirements of 2NF. Ensure that no non-key column depends on other non-key columns. All non-key attributes must depend only on the primary key.
- **Result**: There are no transitive dependencies, meaning that non-key columns cannot be indirectly dependent on the primary key.

4. Boyce-Codd Normal Form (BCNF):

- **Goal**: Address situations where 3NF is not sufficient.
- **Action**: Meet all requirements of 3NF. Ensure that every determinant (a column that determines other columns) is a candidate key.
- **Result**: Even if the table satisfies 3NF, it may still have some anomalies due to dependencies on non-candidate keys. BCNF eliminates such anomalies.

5. Fourth Normal Form (4NF):

- **Goal**: Remove multi-valued dependencies.
- **Action**: Meet all requirements of BCNF. Ensure that there are no multi-valued dependencies, where one record contains multiple sets of independent values for different attributes.
- **Result**: Data is split into smaller, more focused tables, removing unnecessary dependencies between columns.

6. Fifth Normal Form (5NF):

- **Goal**: Remove join dependencies.
- **Action**: Meet all requirements of 4NF. Ensure that no facts are lost during the decomposition of tables. A table should be split only if it is necessary to remove redundancy and avoid anomalies during joins.
- **Result**: Ensures that every join dependency is a consequence of the candidate keys.

Each of these normalization forms improves the structure of the database, ensuring data integrity, minimizing redundancy, and making the database more efficient to query and maintain.

What is Denormalization?

Denormalization is the process of intentionally introducing redundancy into a database by incorporating data from related tables. It involves moving from higher normal forms (like 3NF or BCNF) to lower normal forms. This is typically done to improve query performance by reducing the number of joins needed when accessing data.

While normalization reduces data redundancy and improves data integrity, denormalization can increase redundancy for the sake of performance, especially in read-heavy databases. By reducing the complexity of joins, denormalization can speed up queries but may lead to issues such as data anomalies and inconsistencies. Therefore, denormalization is usually applied selectively, depending on specific performance requirements.

What is a stored procedure?

A Stored Procedure is a collection or a group of T-SQL statements that are precompiled and stored together in the database. These procedures are used to perform operations such as retrieving, inserting, updating, or deleting data, and can also include control-of-flow statements.

Stored Procedures offer benefits such as reduced network load because they are precompiled and stored in the database. They allow for code reuse, improve security (by controlling access to underlying data), and can help with performance optimization.

To create a Stored Procedure, the `CREATE PROCEDURE` (or `CREATE PROC`) statement is used, followed by the procedure name and its associated logic.

Example:

```
CREATE PROCEDURE GetEmployeeDetails
AS
BEGIN
   SELECT * FROM Employees
END
```

Why we use Stored Procedure?

There are several reasons to use a Stored Procedure. They are a network load reducer and decrease execution time because they are precompiled. The most important use of a Stored Procedure is for security purposes. They can restrict SQL Injection. We can avoid SQL injection by use of a Stored Procedure.

How to create a Stored Procedure

To create a stored procedure in SQL Server, use the `CREATE PROCEDURE` statement, followed by the procedure name and the logic enclosed within a `BEGIN...END` block. A stored procedure can be used to retrieve, insert, or modify data, and it can include complex SQL queries.

Example:

```
CREATE PROCEDURE spEmployee
AS
BEGIN
    SELECT EmployeeId, Name, Gender, DepartmentName
    FROM tblEmployees
    INNER JOIN tblDepartments ON tblEmployees.EmployeeDepartmentId = tblDepartments.DepartmentId
END
```

Advantages of using a Stored Procedure in SQL Server:

- **Easy Maintenance & Reusability**: Stored procedures are easy to maintain and can be reused across different applications.
- **Execution Plan Retention**: Stored procedures retain the state of execution plans, improving performance when executed multiple times.
- **Security**: Stored procedures can be encrypted, preventing direct access to the underlying SQL code and helping to protect against SQL injection attacks.

What is a function in SQL Server?

A function is a sequence of statements that accepts input, processes them to perform a specific task and provides the output. Functions must have a name but the function name can never start with a special character such as @, $, #, and so on.

Types of function

- Pre-Defined Function
- User-Defined Function

User-defined Function:

In a user-defined function we write our logic according to our needs. The main advantage of a user-defined function is that we are not just limited to pre-defined functions. We can write our own functions for our specific needs or to simplify complex SQL code. The return type of a SQL function is either a scalar value or a table.

Creation of a function

```
Create function ss(@id int)
```

returns table as return select * from item where itemId = @id

Execution of the Function

```
select * from ss(1)
```

What are the different types of functions in SQL Server?

In SQL Server, functions are used to return a result or value. They can be utilized within **SELECT** statements, and unlike stored procedures, they are always expected to return a value.

- Functions only work with **SELECT** statements.
- Functions can be used anywhere in SQL, such as with **AVG**, **COUNT**, **SUM**, **MIN**, **DATE**, and more.
- Functions are **compiled** each time they are executed.
- A function **must return a result** or value.
- Functions can only work with **input parameters**.
- **TRY...CATCH** statements are **not supported** in functions.

Function Types in SQL Server:

SQL Server includes several types of functions:

1. **Scalar Functions**:
    - A scalar function returns a single value based on the input parameters.
    - Examples: `LEN()`, `GETDATE()`, `UPPER()`, etc.
2. **Table-Valued Functions (TVFs)**:
    - A table-valued function returns a table as the result. These are often used to simplify complex queries by encapsulating logic in a reusable manner.

- Examples: **Inline Table-Valued Function (iTVF)**, **Multistatement Table-Valued Function (mTVF)**.

3. **Aggregate Functions**:
   - These are pre-defined functions in SQL Server that operate on a set of values and return a single value.
   - Examples: `AVG()`, `COUNT()`, `SUM()`, `MIN()`, `MAX()`.
4. **Predefined System Functions**:
   - These are built-in functions in SQL Server that perform a variety of operations, such as string manipulation, mathematical operations, date operations, and more.
   - Examples: `GETDATE()`, `DATEPART()`, `ISNULL()`, `CONVERT()`.
5. **User-Defined Functions (UDFs)**:
   - These are custom functions created by the user. They can either return a scalar value or a table.
   - Example: Custom functions to calculate business-specific values or simplify complex logic.

These functions allow SQL Server users to abstract repetitive logic, increase reusability, and make queries more readable and efficient.

What is a trigger in SQL Server?

"A Trigger is a Database object just like a stored procedure or we can say it is a special kind of Stored Procedure which fires when an event occurs in a database.".

It is a database object that is bound to a table and is executed automatically. We cannot explicitly call any trigger. Triggers provide data integrity and used to access and check data before and after modification using DDL or DML query.

There are two types of Triggers: 1. DDL Trigger 2. DML trigger

DDL Triggers: They fire in response to DDL (Data Definition Language) command events that start with Create, Alter and Drop like Create_table, Create_view, drop_table, Drop_view and Alter_table.

Code of DDL Triggers:

```
create trigger saftey on database for
create_table, alter_table, drop_table
```

as print 'you can not create ,drop and alter table in this database' rollback;

DML Triggers: They fire in response to DML (Data Manipulation Language) command events that start with Insert, Update and Delete like insert_table, Update_view and Delete_table.

Code of DML Trigger:

```
create trigger deep on emp  for insert, update, delete  as print 'you can notinsert,update and delete this table I'  rollback;
```

Why do we need triggers?

Triggers are needed in databases for several reasons:

1. **Automatic Actions**: Triggers automatically execute predefined actions in response to specific events such as `INSERT`, `UPDATE`, or `DELETE` on a table. This reduces the need for manual intervention.
2. **Enforcing Business Rules**: Triggers help enforce complex business rules and constraints that cannot be easily handled by standard constraints. For example, ensuring data integrity by automatically updating related records when one record is changed.
3. **Audit Trails**: Triggers can maintain an audit trail by logging changes to sensitive data (e.g., tracking who made changes, what was changed, and when the change occurred).
4. **Consistency**: Triggers help maintain consistency across related data by ensuring that updates or deletions in one table automatically reflect in others, such as updating or deleting child records when parent records are changed.
5. **Data Validation**: Triggers can be used to validate data before it's inserted or modified, ensuring that only valid data is stored in the database.

In short, triggers automate database management tasks, enhance data integrity, and support complex operations that may not be easily handled by application code alone.

Why and When to Use a Trigger?

A **trigger** is a special type of stored procedure that automatically executes or fires when specific events occur in a database. Triggers are typically used to automate processes, enforce business rules, and ensure data integrity. They help maintain consistency and can perform various tasks such as updating records, logging changes, or preventing unauthorized operations.

When to Use Triggers:

1. **Data Integrity**: Ensuring that data remains consistent across multiple tables after changes occur. For example, if a related table needs to be updated when a record is modified in another table.

2. **Audit Logging**: Automatically tracking and recording changes made to data. This helps maintain an audit trail for compliance and monitoring purposes.
3. **Enforcing Business Rules**: Automatically checking conditions or applying rules when insert, update, or delete operations occur. This ensures that the data conforms to the predefined business logic.

Example:

If you want to log the addition of new employees automatically, a **trigger** can be used to log the action into an audit table whenever a new record is inserted into the **Employees** table.

```
CREATE TRIGGER Employee_Audit_Trigger
ON Employees
AFTER INSERT
AS
BEGIN
    INSERT INTO AuditLog (Action, ActionDate, EmployeeID)
    SELECT 'Inserted', GETDATE(), EmployeeID
    FROM inserted;
END;
```

In this example, every time a new employee is inserted into the **Employees** table, the trigger logs the insertion into the **AuditLog** table.

Types of Triggers in SQL Server:

1. **Data Definition Language (DDL) Triggers**: These are used to respond to database schema changes like creating, altering, or dropping tables.
2. **Data Manipulation Language (DML) Triggers**: These are used to automatically respond to **INSERT**, **UPDATE**, or **DELETE** operations on a table.
3. **Logon Triggers**: These are fired when a user logs into the database.

Example of DML Trigger:

```
CREATE TRIGGER trgAfterInsert ON [dbo].[Employee_Test]
FOR INSERT
AS
BEGIN
    DECLARE @empid INT, @empname VARCHAR(100), @empsal DECIMAL(10, 2);
    DECLARE @audit_action VARCHAR(100);

    SELECT @empid = i.Emp_ID FROM inserted i;
    SELECT @empname = i.Emp_Name FROM inserted i;
    SELECT @empsal = i.Emp_Sal FROM inserted i;

    SET @audit_action = 'Inserted Record -- After Insert Trigger.';

    INSERT INTO Employee_Test_Audit (Emp_ID, Emp_Name, Emp_Sal, Audit_Action, Audit_Timestamp)
    VALUES (@empid, @empname, @empsal, @audit_action, GETDATE());

    PRINT 'AFTER INSERT trigger fired.';
END;
GO
```

Example of DDL Trigger:

```
CREATE TRIGGER tr_TableAudit ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
    PRINT 'You must disable the TableAudit trigger to modify any table in this database.'
    ROLLBACK;
GO
```

Triggers are used when you need to automate specific actions or ensure data consistency in your database without requiring manual intervention.

What is a View in the Database?

A **View** is nothing but a **select query** with a name given to it, or we can simply say a view is a **Named Query**. Ok! Why do we need a view? There can be many answers for this. Some of the important reasons are:

- A view can combine data from multiple tables using adequate joins, and while bringing it, it may require complex filters and calculated data to form the

required result set. From a user's point of view, all these complexities are hidden, and data is queried from a single table.

- Sometimes for **security purposes**, access to the table, table structures, and table relationships are not given to the database user. All they have is access to a view, not knowing what tables actually exist in the database.
- Using the view, you can restrict the user to update only portions of the records.

The following are the key points to be noted about views:

1. Multiple views can be created on one table.
2. Views can be defined as **read-only** or **updatable**.
3. Views can be indexed for better performance.
4. **Insert**, **update**, and **delete** can be done on an updatable view.

Why Do I Need Views in a Database?

There are a number of scenarios where we have to look for a **view** as a solution:

- To **hide the complexity** of the underlying database schema, or customize the data and schema for a set of users.
- To **control access** to rows and columns of data.
- To **aggregate data** for performance.

Views are used for **security purposes** because they provide encapsulation of the name of the table. Data is in the virtual table, not stored permanently. Views display only selected data.

Syntax of a View:

```
CREATE VIEW view_name AS
SELECT column_name(s) FROM table_name WHERE condition;
```

There are two types of views:

- Simple View
- Complex View

What is the Difference Between Primary Key and Unique Key?

- **Primary Key** does not allow **NULL** values, while **Unique Key** allows **one NULL** value.
- **Primary Key** will create a **clustered index** on the column, whereas **Unique Key** will create a **non-clustered index** by default.

How Can You Increase SQL Performance?

Following are tips that can help increase your SQL performance:

- **Limit the Number of Indexes**: Every index increases the time taken to perform **INSERT**, **UPDATE**, and **DELETE** operations. Try to keep the number of indexes between **4-5** per table. If the table is read-only, you may increase the number of indexes.
- **Keep Indexes Narrow**: Narrow indexes reduce the size of the index and the number of reads required. Always try to make indexes as small as possible.
- **Use Integer Columns for Indexes**: It's preferable to create indexes on columns that have **integer** values rather than **character** values for better performance.
- **Order of Columns in Composite Indexes**: In composite (multi-column) indexes, the order of columns matters. Place the most selective columns on the leftmost side of the index key to optimize queries.
- **Use Surrogate Keys for Joins**: When joining multiple tables, consider creating surrogate integer keys and creating indexes on them. Surrogate integer primary keys (such as identity) are ideal for tables with fewer insert operations.
- **Prefer Clustered Indexes**: Clustered indexes are more beneficial when selecting by a range of values or using **GROUP BY** or **ORDER BY**. If you perform the same query frequently on the table, consider creating a **covering index**.
- **SQL Server Profiler**: Use the **SQL Server Profiler Trace Wizard** with the **"Identify Scans of Large Tables"** trace to determine which tables are being scanned by queries, rather than using an index.

By following these practices, you can ensure your SQL performance remains optimal, especially for large datasets.

What is the Use of OLAP?

OLAP (Online Analytical Processing) is useful because it allows fast and interactive access to aggregated data, enabling users to quickly analyze and explore large datasets. OLAP provides the ability to drill down into detailed data, giving users deeper insights into trends, patterns, and anomalies. It is particularly effective for complex queries and multidimensional analysis, supporting tasks such as data reporting, forecasting, and decision-making.

OLAP systems typically allow users to slice and dice data, perform dynamic calculations, and generate reports in real-time, making them invaluable for business intelligence applications.

What is a Measure in OLAP?

A **measure** in OLAP refers to the key performance indicator (KPI) that you want to evaluate in your data. Measures are typically numerical values that are aggregated, such as totals, averages, or sums. To identify which numbers in the data are measures, a useful rule of thumb is: **If a number makes sense when it is aggregated**, then it is likely a measure.

For example, in a sales database, the number of sales, revenue, or profit would be considered measures because they are values that can be summed, averaged, or analyzed for patterns.

What are Dimensions in OLAP?

Dimensions in OLAP refer to the categories or attributes used for analyzing data. They provide context to the measures, allowing users to view data from different perspectives. For instance, in a revenue report that analyzes data by month and sales region, the two dimensions are **time** and **sales region**.

Typical dimensions include:

- **Product**: Categories of products being sold.
- **Time**: Time periods like days, months, or years.
- **Region**: Geographical regions like countries, states, or cities.

These dimensions allow you to segment and organize data for more meaningful analysis and reporting.

What are Levels in Dimensions?

Levels in dimensions refer to the hierarchical structure within a dimension, organizing data from the most general to the most specific. Each level represents a unique position in the hierarchy, and within each level, there are **members** that provide actual data values.

For example, in a **Time dimension**, the hierarchy might consist of four levels:

- **Year**
- **Quarter**
- **Month**
- **Day**

Alternatively, a **Time dimension** could have three levels, such as:

- **Year**
- **Week**
- **Day**

The values within these levels are called **members**. For example, the years **2002** and **2003** are members of the **Year** level in the Time dimension.

What are Fact Tables and Dimension Tables in OLAP?

In **OLAP** systems, data is typically organized into **fact tables** and **dimension tables**.

- **Fact Tables**: These contain quantitative data, typically **measures** that you want to analyze (e.g., sales, profit, etc.). They also contain **foreign keys** linking to the related **dimension tables**. A fact table might have multiple columns for different measures, such as sales amounts or quantities sold.
- **Dimension Tables**: These contain the descriptive (categorical) attributes related to the facts. These are often the **dimensions** along which the data will be analyzed (e.g., time, region, product).

Star Schema in OLAP:

A **star schema** is a type of database schema used in **OLAP** to represent data in a way that is optimized for querying and analysis.

- In a **star schema**, the **fact table** is at the center, and the **dimension tables** are connected to it like the points of a star.
- The **fact table** contains **foreign keys** linking to each dimension and **measure columns** representing the data you want to analyze.
- The **dimension tables** contain descriptive attributes that allow you to filter, group, and aggregate the data.

Example:

Consider a sales analysis scenario:

- **Fact Table**: `Sales`
  - Columns: `SalesAmount`, `QuantitySold`, `ProductID`, `DateID`, `StoreID`
- **Dimension Tables**:
  - `Products`: `ProductID`, `ProductName`, `Category`
  - `Dates`: `DateID`, `Year`, `Month`, `Day`
  - `Stores`: `StoreID`, `StoreName`, `Region`

The **fact table** `Sales` will store the measures (`SalesAmount`, `QuantitySold`) and will have foreign keys referencing the **dimension tables** `Products`, `Dates`, and `Stores`.

This design allows fast querying and aggregation across various dimensions.

What is DTS?

DTS (Data Transformation Services) is a set of tools in SQL Server that allows users to import, transform, and export data between different sources and destinations. DTS is particularly useful for tasks such as:

- **Importing** data from various external sources like flat files, spreadsheets, or other databases.
- **Transforming** data as it is imported, such as changing formats, modifying values, or applying business rules.
- **Exporting** data to different destinations such as other databases or files.

The name "Data Transformation Services" highlights the transformation capabilities, as it can modify the data during the transfer process to meet specific requirements.

Example:

1. You can use DTS to import data from a CSV file into a SQL Server database.
2. During the import, you can apply transformations like:
    - Changing date formats
    - Combining data from multiple columns
    - Calculating new values based on existing columns

Although DTS is now replaced by **SQL Server Integration Services (SSIS)** in newer versions of SQL Server, it still serves as an important concept in data integration tasks.

What is Fill Factor?

Fill Factor in SQL Server specifies how full SQL Server will make each index page when creating or rebuilding an index. It defines the percentage of space to be filled with data on each index page, leaving the rest as free space to accommodate future insertions. The fill factor is a value between 1 and 100, where:

- **Fill Factor = 100**: The index page is completely filled, leaving no free space. This results in fewer pages but may cause page splits when new rows are inserted.
- **Fill Factor < 100**: The index pages are only partially filled, leaving some free space for future insertions, which reduces page splits.

When the index page becomes full and there is no free space to insert a new row, SQL Server performs a **page split**. This involves creating a new index page and transferring some rows from the full page to the new one. Page splits can cause performance issues as they involve extra IO and maintenance.

Example:

- **Fill Factor = 100**: Use when the data is mostly static or rarely changes (e.g., read-only tables).
- **Fill Factor = 70**: Use for tables with frequent updates or inserts, allowing space for growth without excessive page splits.

Page Split Example:

When an index page is filled beyond its capacity and SQL Server cannot fit new rows, a **page split** occurs. The data in the page is divided between the original and the new page, which can lead to fragmentation if not managed properly.

By adjusting the fill factor, you can optimize the index storage based on the workload and reduce the number of page splits, improving overall performance.

What is RAID and How Does it Work?

RAID (Redundant Array of Independent Disks) is a technology used to improve data availability, reliability, and performance by combining multiple disk drives into an array. The data is distributed across these drives, using different techniques such as **striping** (splitting data across multiple drives), **mirroring** (duplicating data across drives), and **parity** (error-checking data). RAID can increase the speed of data access and also provide redundancy in case of drive failure. The most common RAID configurations are **RAID 1**, **RAID 5**, and **RAID 10**.

RAID Levels:

- **RAID 1 (Mirroring)**: Data is duplicated across two drives. If one drive fails, the data remains intact on the other drive. This provides high availability but requires double the storage capacity. It's optimized for fast writes.
- **RAID 5 (Striping with Parity)**: Data is split across at least three drives, with one drive storing parity data. If a drive fails, the lost data can be reconstructed using the parity information. RAID 5 is optimized for read-heavy operations but provides redundancy and good storage efficiency.
- **RAID 10 (Combination of RAID 1 and RAID 0)**: This is a hybrid of RAID 1 and RAID 0. It offers both redundancy and performance by mirroring data and also striping it across multiple drives. RAID 10 requires at least four drives and is suitable for databases that need high availability and performance.

Example of RAID 5:

RAID 5 works by writing parts of data across all drives in the array, with one drive storing parity. If a drive fails, the data can be rebuilt using the remaining data and parity. For instance:

Data: 3 + 7 = 10 (where 10 is the parity) If a drive fails, the data can be restored using the other drives and the parity bit.

SQL Server Difference Between @@IDENTITY, SCOPE_IDENTITY(), and IDENT_CURRENT

- **@IDENTITY**: Returns the last inserted identity value from any table in the current session, regardless of the scope. It can return the identity from a different table if triggered by a trigger.

  Syntax:

  ```
  SELECT @@IDENTITY
  ```

- **SCOPE_IDENTITY()**: Returns the last inserted identity value within the current session and scope. It is more reliable when used within stored procedures or queries that may trigger other inserts.

  Syntax:

  ```
  SELECT SCOPE_IDENTITY()
  ```

- **IDENT_CURRENT(table_name)**: Returns the last identity value inserted into a specific table, regardless of the session or scope. This is useful if you need the identity from a specific table.

  Syntax:

  ```
  SELECT IDENT_CURRENT('table_name')
  ```

Example:

```
CREATE TABLE SAMPLE1 (Id INT IDENTITY)
CREATE TABLE SAMPLE2 (Id INT IDENTITY(100,1))

-- Trigger to execute while inserting data into SAMPLE1 table
GO
CREATE TRIGGER TRGINSERT ON SAMPLE1 FOR INSERT
AS
BEGIN
    INSERT SAMPLE2 DEFAULT VALUES
END
GO

-- Inserting data into SAMPLE1
INSERT SAMPLE1 DEFAULT VALUES

-- Checking identity values
SELECT @@IDENTITY          -- Returns 100 (inserted by trigger into SAMPLE2)
SELECT SCOPE_IDENTITY()    -- Returns 1 (inserted by the query into SAMPLE1)
SELECT IDENT_CURRENT('SAMPLE2')  -- Returns the last inserted value in SAMPLE2
```

In this scenario, `@@IDENTITY` may return an unexpected result from a different table (SAMPLE2) due to triggers. `SCOPE_IDENTITY()` is the most reliable for fetching the identity from the current operation and table.

Difference between char, varchar, and nvarchar in SQL Server

In SQL Server, **char**, **varchar**, and **nvarchar** are used to store text, but they differ in terms of length, storage, and use cases:

Char DataType

- **Description**: Used to store a fixed length of characters.
- **Memory Allocation**: Allocates a fixed amount of memory, regardless of the actual data inserted.
- **Example**: If you declare `char(50)` and insert only 10 characters, it still uses 50 characters of memory, wasting the remaining space.
- **Use Case**: Best suited for storing fixed-length data (e.g., country codes or telephone numbers).

Varchar DataType

- **Description**: Used to store variable-length characters, suitable for non-Unicode data.
- **Memory Allocation**: Allocates memory dynamically based on the number of characters inserted.
- **Example**: If you declare `varchar(50)` and insert only 10 characters, it will use memory for just those 10 characters.
- **Use Case**: Ideal for storing data that varies in length (e.g., names or email addresses), where Unicode support is not needed.

Nvarchar DataType

- **Description**: Similar to **varchar**, but used for storing Unicode characters, enabling support for multiple languages.
- **Memory Allocation**: Takes twice the memory of **varchar** to store characters, as it stores Unicode data.

- **Example**: If you declare `nvarchar(50)` and insert 10 characters, it allocates space for 10 Unicode characters (which can take more space than standard characters).
- **Use Case**: Best for applications that require storing data in multiple languages or support internationalization.

Difference between bit, tinyint, smallint, int, and bigint data types in SQL Server

SQL Server offers various integer types, each with different ranges and storage sizes:

Bit DataType

- **Description**: Used to store a single bit of data, either 0 or 1.
- **Storage Size**: 1 bit.
- **Use Case**: Ideal for Boolean values or flags.

Tinyint DataType

- **Description**: Represents an unsigned 8-bit integer.
- **Range**: 0 to 255.
- **Storage Size**: 1 byte.
- **Use Case**: Suitable for small range values (e.g., age or counts).

Smallint DataType

- **Description**: Represents a signed 16-bit integer.
- **Range**: -32,768 to 32,767.
- **Storage Size**: 2 bytes.
- **Use Case**: Best for storing small integer values that fit within this range.

Int DataType

- **Description**: Represents a signed 32-bit integer.
- **Range**: -2,147,483,648 to 2,147,483,647.
- **Storage Size**: 4 bytes.
- **Use Case**: Most commonly used for general integer storage in SQL Server.

Bigint DataType

- **Description**: Represents a signed 64-bit integer.
- **Range**: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- **Storage Size**: 8 bytes.
- **Use Case**: Suitable for very large integer values (e.g., large counts, financial data).

These data types help optimize storage and performance based on the size and range of the data being stored.

What is the Difference Between DELETE TABLE and TRUNCATE TABLE Commands?

The **DELETE** and **TRUNCATE** commands are both used to remove data from a table, but they work in different ways and have distinct characteristics:

1. **Logging**:
    - **DELETE**: This command is a fully logged operation, meaning every row that is deleted is logged. This can slow down performance, especially for large datasets.
    - **TRUNCATE**: This command is minimally logged. It only logs the deallocation of data pages rather than individual row deletions, which makes it much faster than DELETE.
2. **Criteria**:
    - **DELETE**: You can specify conditions (using `WHERE`) to delete specific rows based on a criteria. Example: `DELETE FROM Employees WHERE Age > 50;`
    - **TRUNCATE**: You cannot specify a condition. It removes all rows from the table without the option to filter data.
3. **Triggers**:
    - **DELETE**: This command fires any `DELETE` triggers defined on the table.
    - **TRUNCATE**: Does not fire `DELETE` triggers because it is a more "direct" operation.
4. **Rollback**:
    - **DELETE**: Can be rolled back if part of a transaction, as each deleted row is logged.
    - **TRUNCATE**: Can also be rolled back if part of a transaction, but it is not logged in the same way as DELETE. However, it removes all rows without logging each deletion.
5. **Performance**:
    - **DELETE**: Slower because it logs every deleted row and can invoke triggers.
    - **TRUNCATE**: Faster because it is minimally logged and does not involve row-by-row deletion.
6. **Impact on Identity Column**:

- **DELETE**: Does not reset the identity column (if present). The identity value continues from where it left off.
- **TRUNCATE**: Resets the identity column to its seed value.

Example:

```
-- DELETE example with criteria
DELETE FROM Employees WHERE Age > 50;

-- TRUNCATE example (removes all rows without criteria)
TRUNCATE TABLE Employees;
```

What Issues Can Occur If Locking Is Not Implemented?

When proper locking is not implemented in SQL Server, several issues can arise that affect the consistency and reliability of data:

1. **Lost Updates**:
   - Lost updates occur when two transactions attempt to modify the same data simultaneously, and the update from the first transaction is overwritten by the second one. This happens when no locks are applied, especially under the `READ UNCOMMITTED` isolation level. For example, a customer pays $1000 and then buys a product for $500. If two counters process these transactions at the same time, the first transaction might be lost, leading to incorrect calculations for the customer's pending amount.
2. **Non-Repeatable Reads**:
   - Non-repeatable reads happen when a transaction reads the same row multiple times and gets different results each time. This issue arises when other transactions modify the data while the original transaction is still in progress. For instance, if a travel agent checks the availability of a flight seat, but another agent books it in the meantime, the first agent might get a different status on the seat upon attempting to book it.
3. **Dirty Reads**:
   - Dirty reads occur when a transaction reads data that has been modified by another transaction but not yet committed. This leads to inconsistencies, as the data might be rolled back later. For example, a customer invoice report is generated while a transaction is processing payments. If a customer makes a payment while the report is being run, the report may show incorrect data, such as an outstanding balance that has already been paid.
4. **Phantom Reads**:
   - Phantom reads occur when a transaction reads a set of rows, but another transaction modifies the dataset by adding or deleting rows during the transaction's execution. For example, if you start editing a row but another transaction deletes it, you'll encounter an error when trying to update a non-existing record. This issue can even occur with the default isolation level (`READ COMMITTED`), but it is prevented by the `SERIALIZABLE` isolation level, which fully isolates transactions from one another.

In summary, locking is crucial to prevent data anomalies such as lost updates, non-repeatable reads, dirty reads, and phantom reads. Without it, concurrent transactions can lead to inconsistent and unpredictable results.

Different Transaction Levels in SQL Server

SQL Server provides different transaction isolation levels that determine how transactions are isolated from one another. These levels control the behavior of locks and how data is accessed during a transaction, impacting concurrency and consistency.

There are four main transaction isolation levels in SQL Server:

1. **READ COMMITTED**:
   - **Locking Behavior**: A shared lock is held for the duration of the transaction. This ensures that other transactions cannot modify the data being read, but other transactions can insert or update data as long as it is not locked by the first transaction.
   - **Effect on Data**: Other transactions can insert or modify data, but they cannot modify data that is currently being read by the transaction holding the lock.
2. **READ UNCOMMITTED**:
   - **Locking Behavior**: No shared or exclusive locks are honored. This allows for the best concurrency but at the cost of data integrity.
   - **Effect on Data**: Transactions can read data that is in the process of being modified by other transactions (dirty reads), potentially resulting in inconsistent data being read.
3. **REPEATABLE READ**:
   - **Locking Behavior**: Holds locks on all data that has been read during the transaction, preventing other transactions from modifying or deleting it.
   - **Effect on Data**: Prevents dirty reads and non-repeatable reads (where a transaction reads the same data and gets different values), but new rows can still be inserted into the table by other transactions.
4. **SERIALIZABLE**:
   - **Locking Behavior**: The most restrictive isolation level, holding shared locks on a range of data. This ensures that no other transactions can insert, update, or delete data in the range that is locked.
   - **Effect on Data**: Prevents phantom reads (where rows appear or disappear between transaction reads). No other transaction can access the locked data range, ensuring complete isolation of the transaction.

Types of Locks in SQL Server

SQL Server uses various types of locks to maintain transaction consistency and integrity:

1. **Shared Lock (S)**:
    - Allows multiple transactions to read data concurrently but prevents any transaction from modifying the data until the shared lock is released.
2. **Exclusive Lock (X)**:
    - Prevents other transactions from accessing the locked data for both reading and writing. This lock is acquired when a transaction modifies data.
3. **Update Lock (U)**:
    - Prevents other transactions from acquiring exclusive locks on the same resource. It is often used to prevent deadlocks when a transaction intends to modify data but first needs to check for other changes.
4. **Intent Locks (IS, IX)**:
    - **Intent Shared (IS)**: Indicates that a transaction intends to read data.
    - **Intent Exclusive (IX)**: Indicates that a transaction intends to modify data.
5. **Schema Modification Lock (Sch-M)**:
    - Prevents other transactions from accessing a table while its schema is being modified (e.g., altering the table structure).
6. **Bulk Update Lock (BU)**:
    - Used during bulk insert operations, ensuring minimal locking of other transactions during large data imports.

These locks are implemented to ensure the proper isolation level and avoid data anomalies like dirty reads, lost updates, or phantom reads.

What are the different locks in SQL SERVER?

Depending on the transaction level, six types of locks can be acquired on data:

1. Intent Locks

The **intent lock** shows the future intention of SQL Server's lock manager to acquire locks on a specific unit of data for a particular transaction. SQL Server uses intent locks to queue exclusive locks, ensuring that these locks will be placed on the data elements in the order the transactions were initiated. Intent locks come in three flavors:

- **Intent Shared (IS):** Indicates that the transaction will read some (but not all) resources in the table or page by placing shared locks.
- **Intent Exclusive (IX):** Indicates that the transaction will modify some (but not all) resources in the table or page by placing exclusive locks.
- **Shared with Intent Exclusive (SIX):** Indicates that the transaction will read all resources, and modify some (but not all). This is accomplished by placing shared locks on the resources read and exclusive locks on the rows modified. Only one SIX lock is allowed per resource at one time, preventing other connections from modifying any data in the resource, although they can still read the data.

2. Shared Locks (S)

Shared locks (S) allow transactions to read data with `SELECT` statements. Other connections can read the data at the same time, but no transactions can modify the data until the shared locks are released.

3. Update Locks (U)

Update locks (U) are acquired just prior to modifying the data. If a transaction modifies a row, the update lock is escalated to an exclusive lock. If no modification is made, the lock is converted to a shared lock. Only one transaction can acquire update locks on a resource at a time. Update locks prevent multiple connections from having shared locks that may eventually want to modify a resource using an exclusive lock. Update locks are compatible with other shared locks but are not compatible with each other.

4. Exclusive Locks (X)

Exclusive locks (X) completely lock the resource from any type of access, including reads. These locks are issued when data is being modified, such as during `INSERT`, `UPDATE`, or `DELETE` statements.

5. Schema Locks

- **Schema Modification Locks (Sch-M):** Acquired when data definition language (DDL) statements such as `CREATE TABLE`, `CREATE INDEX`, `ALTER TABLE`, etc., are executed.
- **Schema Stability Locks (Sch-S):** Acquired when stored procedures are being compiled.

6. Bulk Update Locks (BU)

Bulk update locks (BU) are used when performing a bulk-copy of data into a table with the `TABLOCK` hint. These locks improve performance while bulk copying data into a table, but they reduce concurrency by disabling any other connections from reading or modifying data in the table.

Can we suggest locking hints to SQL SERVER?

Yes, we can provide locking hints in SQL Server to override its default lock decisions. For example, the **ROWLOCK** hint can be used with an `UPDATE` statement to force SQL Server to lock each row individually affected by the modification. However, this may not always be the best choice.

For instance, if the `UPDATE` statement affects 95% of the rows in a table, using `ROWLOCK` could result in SQL Server acquiring a large number of individual locks (e.g., 950 locks for 1000 rows). This can significantly increase memory usage and degrade performance compared to acquiring a single table lock. Therefore, it's important to carefully consider whether using such locking hints is appropriate, especially for large tables or bulk operations.

What is LOCK escalation?

Lock escalation is the process of converting low-level locks, such as row or page locks, into higher-level locks like table locks. This is done to prevent excessive memory usage, as each lock is a memory structure. If too many locks are held, it can lead to increased memory consumption. To avoid this, SQL Server escalates fine-grained locks (e.g., row locks, page locks) into more coarse-grained locks (e.g., table locks).

In SQL Server 6.5, the lock escalation threshold was configurable. However, from SQL Server 7.0 onwards, SQL Server dynamically manages lock escalation based on system resources and workload requirements.

What are the different ways of moving data between databases in SQL Server?

There are several methods for moving data between databases in SQL Server. The best method depends on your specific requirements, such as the volume of data, the complexity of the transfer, and the need for automation or consistency. Some of the most common options are:

- **BACKUP/RESTORE:** This method involves backing up a database and restoring it to another location. It's useful for transferring entire databases.
- **Detaching and Attaching Databases:** You can detach a database from one SQL Server instance and attach it to another, effectively moving the entire database.
- **Replication:** SQL Server replication allows data to be copied and distributed between databases. It's useful for maintaining synchronized copies of data across multiple servers.
- **DTS (Data Transformation Services):** DTS is a tool that can be used for moving and transforming data. It is commonly used for more complex data transfer tasks.
- **BCP (Bulk Copy Program):** BCP is a command-line tool used to export and import large amounts of data efficiently.
- **Log Shipping:** This involves transferring transaction log backups from one server to another and applying them to a secondary database, useful for disaster recovery scenarios.
- **INSERT...SELECT:** This is a SQL query-based method that allows data to be copied from one table to another within the same or different databases.
- **SELECT...INTO:** Similar to `INSERT...SELECT`, but it also creates the destination table if it does not already exist.
- **Creating INSERT Scripts:** You can generate SQL scripts that insert data into the destination database. This is useful for smaller datasets or when you need to replicate data manually.

What is the difference between a HAVING CLAUSE and a WHERE CLAUSE?

The **HAVING** clause and the **WHERE** clause are both used to filter data in SQL, but they are applied at different stages in the query execution process:

- **WHERE Clause:**
  - The `WHERE` clause filters rows **before** any grouping or aggregation is performed.
  - It applies conditions to individual rows in a table.
  - Typically used with SELECT queries without `GROUP BY`.
  - Example: `SELECT * FROM Employees WHERE Salary > 50000;`
- **HAVING Clause:**
  - The `HAVING` clause filters rows **after** the `GROUP BY` function has been applied, and is used with aggregate functions like `SUM`, `COUNT`, `AVG`, etc.
  - It applies conditions to the groups formed by `GROUP BY`.
  - Example: `SELECT Department, AVG(Salary) FROM Employees GROUP BY Department HAVING AVG(Salary) > 60000;`

What is the difference between UNION and UNION ALL SQL syntax?

The **UNION** and **UNION ALL** SQL syntax are used to combine the results from two or more queries, but they differ in how they handle duplicate records:

- **UNION:**
  - Combines the result sets of two queries.
  - Removes **duplicate records**, returning only distinct rows.
  - Example: `SELECT column_name FROM table1 UNION SELECT column_name FROM table2;`
- **UNION ALL:**
  - Combines the result sets of two queries.
  - **Includes all records**, even duplicates.
  - Example: `SELECT column_name FROM table1 UNION ALL SELECT column_name FROM table2;`

Note: The selected records from each query must have the same number of columns and compatible data types for the syntax to work.

What are the different types of triggers in SQL SERVER?

There are two main types of triggers in SQL Server:

1. INSTEAD OF Triggers

- **INSTEAD OF triggers** are fired in place of the triggering action (INSERT, UPDATE, DELETE).
- For example, if an INSTEAD OF UPDATE trigger exists on a table (e.g., Sales), and an UPDATE statement is executed, the trigger will execute instead of performing the UPDATE directly on the table.
- **Key feature:** INSTEAD OF triggers are executed automatically **before** Primary Key and Foreign Key constraints are checked.
- **Use case:** They can be created on **views** to manage updates to underlying tables.

2. AFTER Triggers

- **AFTER triggers** (also known as **FOR triggers**) are executed **after** the SQL action (INSERT, UPDATE, DELETE) is completed.
- These triggers are the traditional type that existed in SQL Server before the introduction of INSTEAD OF triggers.
- **Key feature:** AFTER triggers are executed **after** the Primary Key and Foreign Key constraints are checked.
- **Use case:** Typically used for actions that need to occur after the data is changed, such as logging or enforcing business rules.

If we have multiple AFTER Triggers on a table, how can we define the sequence of the triggers?

If a table has multiple **AFTER triggers**, you can define the execution order of the triggers using the stored procedure `sp_settriggerorder`. This stored procedure allows you to specify which trigger should execute first and which should execute last.

Syntax:

```
sp_settriggerorder
    @table_name = 'table_name',
    @trigger_name = 'trigger_name',
    @order = 'First' | 'Last'
```

- **@table_name:** Name of the table where the triggers are defined.
- **@trigger_name:** Name of the trigger whose execution order you want to control.
- **@order:** You can set this to `'First'` or `'Last'` to define the sequence of trigger execution. The default is `'None'`, which means no specific order is enforced.

This ensures that when multiple AFTER triggers are defined on a table, they will execute in the specified order.

What is SQL Injection?

SQL injection is a form of attack on a database-driven website in which an attacker executes unauthorized SQL commands by exploiting insecure code on a system connected to the internet, bypassing firewalls and gaining access to sensitive data. This attack allows the attacker to steal information from a database that would otherwise not be accessible, and potentially gain access to an organization's host computers through the computer hosting the database.

SQL injection attacks are often easy to avoid by implementing strong input validation and using secure coding practices. When an attacker inputs malicious SQL code into input fields, they can manipulate the SQL query to execute commands that were not intended.

Example of SQL Injection:

Consider the following SQL query:

```
SELECT email, passwd, login_id, full_name FROM members WHERE email = 'x'
```

An attacker may input:

```
x'; DROP TABLE members;
```

This causes the following query to be executed:

```
SELECT email, passwd, login_id, full_name FROM members WHERE email = 'x'; DROP TABLE members;
```

In this case, the attacker is able to delete the entire `members` table from the database, resulting in data loss.

What is the difference between Stored Procedure (SP) and User Defined Function (UDF)?

Here are some major differences between a **Stored Procedure (SP)** and a **User Defined Function (UDF)**:

1. **Data Modification:**
    - **UDF:** You cannot change any data with a UDF.
    - **SP:** Stored Procedures can modify data (e.g., using `INSERT`, `UPDATE`, `DELETE`).
2. **Use in XML FOR Clause:**
    - **UDF:** Cannot be used in the `XML FOR` clause.

- **SP:** Stored Procedures can be used in the `XML FOR` clause.

3. **Output Parameters:**
    - **UDF:** Does not return output parameters.
    - **SP:** Stored Procedures can return output parameters.
4. **Error Handling:**
    - **UDF:** If there is an error, UDF stops executing.
    - **SP:** Stored Procedures continue execution even if an error occurs, ignoring it and moving to the next statement.
5. **Changes to Server Environment:**
    - **UDF:** Cannot make permanent changes to the server environment.
    - **SP:** Stored Procedures can change some of the server environment settings.

How can you raise custom errors from stored procedure?

To raise custom errors from a stored procedure, the `RAISERROR` statement is used. This statement can produce an ad-hoc error message or retrieve a custom message stored in the `sysmessages` table. Below is the syntax of the `RAISERROR` statement:

```
RAISERROR ({msg_id | msg_str}, severity, state, [argument, ...,n]) [WITH option [, ...n]]
```

Components of RAISERROR:

1. **msg_id:** The ID for an error message stored in the `sysmessages` table.
2. **msg_str:** A custom message that is not in the `sysmessages` table.
3. **severity:** The severity level of the error (values between 0 and 25). Levels 0-18 are used by users, while 19-25 are restricted to `sysadmin` role members.
4. **state:** A value indicating the invocation state of the error (valid values 0–127).
5. **argument:** One or more variables that customize the message. For example, the current process ID (`@@SPID`) can be included.
6. **WITH option:** Optional clauses that modify behavior:
    - **LOG:** Logs the error in the SQL Server error log and the NT application log.
    - **NOWAIT:** Sends the message immediately to the client.
    - **SETERROR:** Sets `@@ERROR` to the unique ID for the message or 50,000.

Example Usage:

Here's how you can raise a custom error with a severity of 10 and a state of 1:

```
RAISERROR ('An error occurred updating the Nonfatal table', 10, 1)
```

This raises the custom error message: "An error occurred updating the Nonfatal table."

Example with Error Handling:

Below is an example of raising a custom error within a stored procedure:

```
USE tempdb
GO
ALTER PROCEDURE ps_NonFatal_INSERT
@Column2 int = NULL
AS
DECLARE @ErrorMsgID int
INSERT INTO NonFatal VALUES (@Column2)
SET @ErrorMsgID = @@ERROR
IF @ErrorMsgID <> 0
BEGIN
    RAISERROR ('An error occurred updating the NonFatal table', 10, 1)
END
```

In this example, if an error occurs during the `INSERT` statement, the custom message is raised to the client.

What is DBCC?

DBCC (Database Consistency Checker Commands) is a set of commands used to check the logical and physical consistency of a SQL Server database. These commands help detect and fix problems within the database. DBCC commands are grouped into four categories:

1. Maintenance Commands

These commands are used for maintenance tasks in SQL Server. Examples include:

- **DBCC DBREINDEX**: Rebuilds the indexes of a table.
- **DBCC DBREPAIR**: Repairs issues related to databases.

2. Miscellaneous Commands

These commands enable specific functionalities like locking or removing DLLs from memory. Examples include:

- **DBCC ROWLOCK**: Enables row-level locking.
- **DBCC TRACEO**: Used for enabling or disabling trace flags.

3. Status Commands

These commands are used to check the status of the database. Examples include:

- **DBCC OPENTRAN**: Displays information about open transactions in a database.
- **DBCC SHOWCONTIG**: Displays fragmentation information for the data and indexes of a specified table.

4. Validation Commands

These commands perform validation operations on the database. Examples include:

- **DBCC CHECKALLOC**: Checks the consistency of the allocation structures in the database.
- **DBCC CHECKCATALOG**: Validates the database catalog.

Example: DBCC SHOWCONTIG

Here is an example of using **DBCC SHOWCONTIG** to check the fragmentation of a table. In this case, the `Customer` table is checked for fragmentation. If the **scan density** is 100%, it means the data is contiguous. A scan density of 95.36% is considered decent, indicating minimal fragmentation.

Note: For a complete list of all DBCC commands, refer to MSDN. It is common for DBA interviews to require knowledge of specific DBCC commands and their functions.

What is the purpose of Replication?

Replication is a method used to keep data synchronized across multiple databases. SQL Server replication involves two main components:

1. Publisher

- The **Publisher** is the database server that makes its data available for replication. It contains the original data that will be replicated to other servers.

2. Subscriber

- The **Subscriber** is the database server that receives the data from the Publisher. Subscribers maintain copies of the data from the Publisher and can be used for querying, reporting, or backup purposes.

Replication ensures that data is consistent across different systems, improving data availability and reliability in distributed environments.

What are the different types of replication supported by SQL SERVER?

SQL Server supports three types of replication:

1. Snapshot Replication

- Snapshot replication takes a snapshot of the database at a specific point in time and moves it to another database. After the initial data load, the data can be refreshed periodically. The main disadvantage is that the entire dataset must be copied every time the snapshot is refreshed, which can be resource-intensive.

2. Transactional Replication

- In transactional replication, data is copied initially like in snapshot replication, but subsequent updates are synchronized by replicating only the changes (transactions) rather than the entire database. This type of replication can run continuously or on a periodic basis, ensuring real-time or near real-time synchronization.

3. Merge Replication

- Merge replication allows data from multiple sources to be combined into a single central database. After the initial data load, changes can be made on both the Publisher and the Subscriber. When they are online, the replication engine detects and merges the changes, updating the data accordingly. This type is useful for scenarios where both ends can modify the data independently.

What is BCP utility in SQL SERVER?

BCP (Bulk Copy Program) is a command-line utility in SQL Server used to import and export large volumes of data between a SQL Server database and data files. It allows for efficient bulk data transfer, making it suitable for moving large datasets quickly. BCP can be used to export data from SQL Server to a file or to import data from a file into SQL Server. It is commonly used for data migration, backup, and loading large datasets into SQL Server.

What is a Cursor?

A **Cursor** in a database is a control structure that allows for traversal over the rows or records in a table. It acts like a pointer, referencing one row at a time in a result set. Cursors are particularly useful for performing operations on each row individually, such as retrieval, addition, or deletion of database records. They are commonly used when a set-based operation is not feasible or when operations need to be performed row by row. However, cursors can be slower than set-based operations, so they should be used judiciously.

What are local and global variables and their differences?

Local Variables
Local variables are variables that are declared and used within a specific function or block of code. They are not accessible outside of that function, and once the function execution ends, the local variable is destroyed. They exist only within the scope of the function where they are declared.

Global Variables
Global variables are variables that are declared outside of any function or block and can be accessed throughout the entire program, in any function or block. These variables retain their values across different function calls and remain in memory as long as the program runs.

Differences

- **Scope:** Local variables are confined to the function they are declared in, while global variables can be accessed from anywhere in the program.
- **Lifetime:** Local variables are created when the function is called and destroyed when the function exits, whereas global variables exist for the duration of the program's execution.
- **Accessibility:** Local variables cannot be accessed by other functions, but global variables can be accessed by any function in the program.

What is an index?

An **index** is a database object that improves the speed of data retrieval operations on a table at the cost of additional space and decreased performance on data modification operations. It works similarly to an index in a book, allowing the database engine to quickly locate the data without scanning the entire table. Indexes are particularly useful for large tables, as they significantly improve query performance by ordering or sorting the data based on specific columns.

Types of Index:

- **Clustered Index:**
  A **Clustered Index** sorts and stores the data in the table based on the values of the indexed column(s). Since the data is physically stored in the order of the index, only one clustered index can exist per table. Columns with Text, nText, and Image data types cannot be used in clustered indexes.

  Example:

  ```
  SET STATISTICS IO ON
  SELECT * FROM Employee WHERE EmpID = 20001;
  ```

- **Non-Clustered Index:**
  A **Non-Clustered Index** creates a separate structure from the table and stores pointers to the data rows. SQL Server supports up to 999 non-clustered indexes per table, and each index can contain up to 1023 columns. Non-clustered indexes cannot include Text, nText, or Image data types.

  Example:

  ```
  CREATE NONCLUSTERED INDEX NCL_ID ON Employee(DeptID);
  SET STATISTICS IO ON
  SELECT * FROM Employee WHERE DeptID = 20001;
  ```

Why do I need an index in a database?

An **index** is an essential database object used to enhance the speed of data retrieval operations. It works by creating a data structure on one or more columns (up to 16 column combinations), which allows SQL Server to quickly locate data, reducing the number of comparisons needed. While indexes significantly improve query performance, they can add overhead to **Data Manipulation Language (DML)** operations such as **INSERT**, **DELETE**, and **UPDATE** because the index must also be updated. Therefore, the decision to create an index depends on the balance between the frequency of data retrieval and modification operations.

Need of Index in Database:

- **Faster Data Retrieval:** Indexes are primarily used to speed up data retrieval from large tables, making searches, filters, and joins more efficient.

Syntax:

- **Create Index:**

  ```
  CREATE INDEX index_name ON table_name(column_name);
  ```

- **Drop Index:**

```
DROP INDEX index_name;
```

Types of Index:

1. **Clustered Index:**
   A clustered index sorts the data in the table based on the indexed columns. There can only be one clustered index per table.
2. **Non-Clustered Index:**
   A non-clustered index creates a separate structure from the table and contains pointers to the data, allowing multiple non-clustered indexes on a table.

What is a query in a database?

A **query** in a database refers to a request made to the database management system (DBMS) to retrieve, manipulate, or modify data. SQL (Structured Query Language) is the language used for querying databases, and it provides commands to create, read, update, and delete data (often referred to as CRUD operations). While **SELECT** queries are the most complex in SQL, queries for data modification, such as **INSERT**, **UPDATE**, and **DELETE**, are relatively simple but come with their own challenges.

SQL Query for Data Manipulation:

- **INSERT**: Adds new data rows to a table.

Syntax for INSERT:

```
INSERT INTO table_name (column_names)
VALUES (values_for_columns);
```

Example:

```
INSERT INTO employee (ID, SURNAME, FIRSTNAME, EMAIL, COUNTRY, PHONE)
VALUES (111, 'vithal', 'wadje', 'vithal.wadje@yahoo.com', 'India', '+914545455454');
```

The DBMS ensures that data integrity is maintained during these operations and handles simultaneous updates by multiple users, making sure their changes do not interfere with one another.

What are query types in a database?

SQL commands in a database are categorized into four main types: **DDL**, **DML**, **DCL**, and **TCL**. These categories serve different purposes and help in organizing and managing the database structure, data, security, and transactions.

DDL - Data Definition Language

DDL commands are responsible for defining and managing the structure of database objects, such as tables, views, and schemas. These commands are used to create, modify, or delete the structure of database objects.

- **Create**: Creates a new database object.
- **Alter**: Modifies an existing database object.
- **Drop**: Deletes an existing database object.

Example:

```
CREATE DATABASE DB2;
GO
CREATE TABLE tblDemo (Id INT PRIMARY KEY, Name CHAR(20));
GO
DROP DATABASE DB2;
```

DML - Data Manipulation Language

DML commands are used to manipulate data stored within database objects. These commands help in inserting, updating, and deleting data.

- **Insert**: Adds new records to a table.
- **Delete**: Removes records from a table.
- **Update**: Modifies existing records in a table.
- **Insert Into**: Inserts bulk data into a table.

Example:

```
INSERT INTO tblDemo VALUES(1, 'Abhishek');
GO
DELETE FROM tblDemo WHERE Id = 4;
GO
UPDATE tblDemo SET Name = 'Sunny' WHERE Id = 6;
GO
```

DCL - Data Control Language

DCL commands are used for managing access and security in a database. These commands control permissions and roles assigned to users for database objects.

- **Grant**: Provides users with specific permissions on a database or object.
- **Revoke**: Removes specific permissions from users.

Example:

```
GRANT CREATE ON DATABASE Kansiris TO UserName;
REVOKE CREATE ON DATABASE Kansiris FROM UserName;
```

TCL - Transaction Control Language

TCL commands manage transactions within the database, ensuring that changes are committed or rolled back properly.

- **Commit**: Saves changes made during a transaction permanently.
- **Rollback**: Reverts the database to its last committed state.
- **Save Tran**: Saves the current transaction at a specific point for possible rollback.

Example:

```
BEGIN TRAN
INSERT INTO tblStudents VALUES('Sumit');
COMMIT;

BEGIN TRAN
INSERT INTO tblStudents VALUES('Kajal');
SAVE TRAN A;
INSERT INTO tblStudents VALUES('Rahul');
SAVE TRAN B;
INSERT INTO tblStudents VALUES('Ram');
SAVE TRAN C;

SELECT * FROM tblStudents;

ROLLBACK TRAN B;
COMMIT;
```

Summary:

- **DDL**: Manages database structure (e.g., `CREATE`, `ALTER`, `DROP`).
- **DML**: Manipulates data (e.g., `INSERT`, `UPDATE`, `DELETE`).
- **DCL**: Manages security and user access (e.g., `GRANT`, `REVOKE`).
- **TCL**: Manages transactions (e.g., `COMMIT`, `ROLLBACK`, `SAVE TRAN`).

What is a join in SQL Server?

A **join** in SQL Server is used to combine data 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 by specifying the relationship among the tables. Joins are crucial when you need to fetch data that is stored across different tables.

Types of Joins in SQL Server

1. **Inner Join** An **INNER JOIN** returns rows when there is a match in both tables. If there is no match, the row is not returned.

   Syntax:

```
SELECT <column_list>
FROM <left_table> [INNER] JOIN <right_table>
ON <join_condition>
```

Example:

```
SELECT Table1.ID, Table1.Name, Table2.Name
FROM Table1
INNER JOIN Table2
ON Table1.ID = Table2.ID
```

2. **Left Join (Left Outer Join)** A **LEFT JOIN** (or **LEFT OUTER JOIN**) returns all rows from the left table, and the matched rows from the right table. If there is no match, the result will contain `NULL` values for columns from the right table.

   Syntax:

   ```
   SELECT <column_list>
   FROM <left_table> LEFT JOIN <right_table>
   ON <join_condition>
   ```

   Example:

   ```
   SELECT Table1.ID, Table1.Name, Table2.Name
   FROM Table1
   LEFT JOIN Table2
   ON Table1.ID = Table2.ID
   ```

3. **Right Join (Right Outer Join)** A **RIGHT JOIN** (or **RIGHT OUTER JOIN**) is similar to the left join but returns all rows from the right table, and the matched rows from the left table.

   Syntax:

   ```
   SELECT <column_list>
   FROM <left_table> RIGHT JOIN <right_table>
   ON <join_condition>
   ```

4. **Full Join (Full Outer Join)** A **FULL JOIN** (or **FULL OUTER JOIN**) returns all rows when there is a match in either the left or right table. If there is no match, the result will contain `NULL` values for columns from the table that doesn't have a matching row.

   Syntax:

   ```
   SELECT <column_list>
   FROM <left_table> FULL JOIN <right_table>
   ON <join_condition>
   ```

5. **Cross Join** A **CROSS JOIN** returns the Cartesian product of the two tables, meaning it will return every combination of rows from the two tables.

   Syntax:

   ```
   SELECT <column_list>
   FROM <left_table> CROSS JOIN <right_table>
   ```

Example of Creating Tables and Using Joins

Now, let's create three tables and apply an **INNER JOIN**:

1. **Creating the tables:**

   ```
   CREATE TABLE Table1 (ID INT, Name VARCHAR(20));
   CREATE TABLE Table2 (ID INT, Name VARCHAR(30));
   CREATE TABLE Table3 (ID INT, Name VARCHAR(40));
   ```

2. **Using an INNER JOIN:**

   To retrieve data from two tables based on a common column (ID), you can use the following query:

   ```
   SELECT Table1.ID, Table1.Name AS Table1_Name, Table2.Name AS Table2_Name
   FROM Table1
   INNER JOIN Table2
   ON Table1.ID = Table2.ID;
   ```

In summary, joins in SQL Server are essential for combining data from multiple tables, and the most commonly used types are **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, **FULL JOIN**, and **CROSS JOIN**. The choice of join depends on the type of result you want based on the relationships between the tables.

What are different types of joins in SQL Server?

Joins are useful for bringing data together from different tables based on their database relations. First we will see how the join operates between tables. Then we will explore the Order of Execution when both a join and a where condition exist. Finally we will move our exploration to the importance of the Join order.

A Join condition defines a way two tables are related in a query by:

- Specifying the column to be used for the Join from each table. In joining foreign keys in a table and its associated key in the other table.
- To use the logical operator in comparing values from the columns.

There are three type of joins available based on the way we join columns of two different tables.

```
Full Join
```

```
Inner Join
```

```
Left outer Join
```

```
Right outer Join
```

Full Join - A full join is somewhat different from the Cartesian product. A Cartesian product will get all the possible row combinations of the two joining tables. A Full Join takes the matching columns plus all table rows from the left table that does not match the right and all table rows in the right that does not match the left. It applies null for unmatched rows on the other end when doing so. The following example shows the full join between Table_A and Table_C

What is a Self-Join?

A **Self-Join** is a query that joins a table with itself. This is useful when you need to compare values within the same table, such as matching records based on certain conditions.

Example:

```
SELECT a.EmployeeName AS Employee, b.EmployeeName AS Manager
FROM Employees a
JOIN Employees b ON a.ManagerID = b.EmployeeID;
```

In this example, the `Employees` table is joined with itself, where `a` and `b` are aliases for the same table, and we're comparing an employee with their manager.

What is a Cross-Join?

A **Cross Join** produces a Cartesian product, where each row in the first table is joined with every row in the second table. The result is the multiplication of the number of rows from both tables.

Example:

```
SELECT *
FROM Products
CROSS JOIN Categories;
```

In this case, if `Products` has 3 rows and `Categories` has 4, the result will contain 12 rows (3 * 4). If a `WHERE` clause is used, the query behaves like an **INNER JOIN**.

What Are User Defined Functions?

User Defined Functions (UDFs) are custom functions created to encapsulate reusable logic. They help avoid writing the same logic multiple times and can be called whenever needed.

Types of User Defined Functions

There are three types of UDFs in SQL Server:

1. **Scalar Functions**: Return a single value.

```
CREATE FUNCTION GetEmployeeAge (@EmployeeID INT)
RETURNS INT
AS
BEGIN
  RETURN (SELECT Age FROM Employees WHERE EmployeeID = @EmployeeID);
END;
```

2. **Inline Table-Valued Functions**: Return a table and have a single `SELECT` statement.

```
CREATE FUNCTION GetEmployeeDetails (@DepartmentID INT)
RETURNS TABLE
AS
RETURN (SELECT * FROM Employees WHERE DepartmentID = @DepartmentID);
```

3. **Multi-statement Table-Valued Functions**: Return a table and can contain multiple SQL statements.

```
CREATE FUNCTION GetEmployeesByDepartment (@DepartmentID INT)
RETURNS @EmployeeTable TABLE (EmployeeID INT, EmployeeName VARCHAR(100))
AS
BEGIN
  INSERT INTO @EmployeeTable
  SELECT EmployeeID, EmployeeName FROM Employees WHERE DepartmentID = @DepartmentID;
  RETURN;
END;
```

What is Collation?

Collation refers to the set of rules that define how character data is sorted and compared. It affects how characters are compared, ordered, and handled in different languages or regions.

Key Points:

- Determines case sensitivity, accent sensitivity, and sorting rules.
- Can be set at the database, column, or query level.

Types of Collation Sensitivity

1. **Case Sensitivity**: Differentiates between uppercase and lowercase characters (e.g., `A ≠ a`).
2. **Accent Sensitivity**: Differentiates characters based on accent marks (e.g., `e ≠ é`).
3. **Kana Sensitivity**: Differentiates between Hiragana and Katakana characters in Japanese.
4. **Width Sensitivity**: Differentiates between single-byte and double-byte characters (e.g., `A ≠ A`).

Advantages and Disadvantages of Stored Procedures

Advantages:

- Modular programming: Can be written once and called multiple times.
- Faster execution due to reduced query parsing.
- Improved security as the logic is stored on the server.
- Reduces network traffic.

Disadvantages:

- Limited to execution within the database.
- Consumes server memory for storage.
- Harder to debug and manage for large applications.

What is Online Transaction Processing (OLTP)?

OLTP (Online Transaction Processing) refers to systems that manage transaction-based applications. OLTP systems are used for data entry, processing, and retrieval, focusing on simplicity, speed, and efficiency.

Example:

- **Bank Transactions**: Daily processing of deposits, withdrawals, and transfers in banking systems. OLTP systems ensure fast and accurate processing.

What is a Clause in SQL?

A **Clause** in SQL is a part of a SQL statement or query that specifies a condition or limits the result set. Clauses are used to filter or define how data should be retrieved, updated, or manipulated.

Examples:

1. **WHERE Clause** - Filters rows based on a condition.

```
SELECT * FROM Employees WHERE Age > 30;
```

2. **HAVING Clause** - Filters records after grouping (used with aggregate functions).

```
SELECT Department, COUNT(*) FROM Employees GROUP BY Department HAVING COUNT(*) > 10;
```

What is a Recursive Stored Procedure?

A **Recursive Stored Procedure** is a stored procedure that calls itself, allowing it to repeat a set of actions until it reaches a specific boundary or termination condition. This is useful for problems that can be broken down into smaller, similar sub-problems (e.g., calculating factorials or processing hierarchical data).

Example:

```
CREATE PROCEDURE Factorial (@Number INT, @Result INT)
AS
BEGIN
    IF @Number = 1
        RETURN @Result
    ELSE
        EXEC Factorial @Number - 1, @Result * @Number
END;
```

What are UNION, MINUS, and INTERSECT Commands?

- **UNION**: Combines the result sets of two queries and removes duplicate rows. Both queries must have the same number of columns, with similar data types.

```
SELECT column1 FROM table1
UNION
SELECT column1 FROM table2;
```

- **MINUS**: Returns rows from the first query that are not present in the second query. It is available in some database systems like Oracle (in SQL Server, `EXCEPT` is used).

```
SELECT column1 FROM table1
MINUS
SELECT column1 FROM table2;
```

- **INTERSECT**: Returns only the rows that are common to both queries.

```
SELECT column1 FROM table1
INTERSECT
SELECT column1 FROM table2;
```

What is an Alias Command in SQL?

An **Alias** is a temporary name given to a table or column in SQL to simplify queries. Aliases are often used in complex queries to make the SQL more readable.

Example:

```
SELECT st.StudentID, Ex.Result
FROM student AS st, Exam AS Ex
WHERE st.StudentID = Ex.StudentID;
```

Here, `st` is the alias for the `student` table, and `Ex` is the alias for the `Exam` table.

What is the Difference Between TRUNCATE and DROP Statements?

- **TRUNCATE**: Removes all rows from a table but keeps the structure intact. The operation cannot be rolled back.

```
TRUNCATE TABLE Employees;
```

- **DROP**: Completely removes a table (or another object like a view) from the database, including its structure. The operation cannot be rolled back.

```
DROP TABLE Employees;
```

What Are Aggregate and Scalar Functions in SQL?

- **Aggregate Functions**: Perform calculations on a set of values and return a single result. Examples include:
  - `MAX()`, `COUNT()`, `SUM()`, `AVG()`, etc.

  Example:

```
SELECT MAX(Salary) FROM Employees;
```

- **Scalar Functions**: Return a single value based on input values. Examples include:
  - `UCASE()`, `NOW()`, `LEN()`, etc.

  Example:

```
SELECT UCASE(EmployeeName) FROM Employees;
```

What is an Inner Join in SQL?

An **Inner Join** in SQL returns only the rows where there is a match in both tables. It excludes rows where no match is found.

Syntax:

```
SELECT *
FROM Table1
INNER JOIN Table2
ON Table1.ID = Table2.ID;
```

Example:

The following query retrieves employee names along with their manager's names from the **employee** table. The join is made within the same table (self-join), matching **EmployeeID** with **ManagerID**.

```
SELECT e1.Employee_Name AS EmployeeName, e2.Employee_Name AS ManagerName
FROM employee e1 (nolock), employee e2 (nolock)
WHERE e1.EmployeeID = e2.ManagerID;
```

In this case:

- Only employees who have a manager will be listed.
- If an employee has no manager, they will not appear in the result.

What is an Outer Join in SQL?

An **Outer Join** is a type of join that returns all rows from one table and the matching rows from the other. If there is no match, NULL values are returned for the missing side. There are three types of outer joins:

1. **Left Outer Join**
2. **Right Outer Join**
3. **Full Outer Join**

Left Outer Join

A **Left Outer Join** (or simply **Left Join**) returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for the columns from the right table.

Syntax:

```
SELECT *
FROM Table1
LEFT OUTER JOIN Table2
ON Table1.ID = Table2.ID;
```

Right Outer Join

A **Right Outer Join** (or **Right Join**) returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for the columns from the left table.

Syntax:

```
SELECT *
FROM Table1
RIGHT OUTER JOIN Table2
ON Table1.ID = Table2.ID;
```

What is Full Join in SQL?

A **Full Outer Join** returns all records from both tables. When there is no match, NULL values are returned for the missing side. It is a combination of both **Left Outer Join** and **Right Outer Join**.

Syntax:

```
SELECT *
FROM Table1
FULL OUTER JOIN Table2
ON Table1.ID = Table2.ID;
```

Example:

```
SELECT e.empId, e.empName, e1.empAdd
FROM emp e
FULL OUTER JOIN emp_add e1
ON e.empId = e1.empId;
```

In this case:

- All employee records are returned, whether or not they have an address.
- All address records are returned, whether or not they belong to an employee.

What is Left Join in SQL Server?

A **LEFT JOIN** (or **LEFT OUTER JOIN**) in SQL Server is used to retrieve data from two tables where all the rows from the left table (first table) are returned, along with the matching rows from the right table (second table). If no matching row exists in the right table, the result will contain **NULL** values for the columns from the right table.

Syntax:

```
SELECT *
FROM Table1
LEFT JOIN Table2
ON Table1.ID = Table2.ID;
```

Example:

To fetch employee details along with their department name, including those employees who might not belong to any department, you would use the following query:

```
SELECT e.EmployeeID, e.Employee_Name, d.Department_Name
FROM employee e (nolock)
LEFT JOIN department d (nolock)
ON e.DepartmentID = d.DepartmentID;
```

In this case:

- All employee rows are included.
- If an employee doesn't have a corresponding department, the department name will be displayed as **NULL**.

What is Right Join in SQL Server?

A **RIGHT JOIN** (or **RIGHT OUTER JOIN**) in SQL Server is used to retrieve data from two tables where all the rows from the right table (second table) are returned, along with the matching rows from the left table (first table). If no matching row exists in the left table, the result will contain **NULL** values for the columns from the left table.

Syntax:

```
SELECT *
FROM Table1
RIGHT JOIN Table2
ON Table1.ID = Table2.ID;
```

Example:

To fetch department details along with employee names (if available), including all departments even if no employees belong to them, you would use the following query:

```
SELECT e.EmployeeID, e.Employee_Name, d.Department_Name
FROM employee e (nolock)
RIGHT JOIN department d (nolock)
ON e.DepartmentID = d.DepartmentID;
```

Here:

- All department rows are included.
- If a department has no employees, the employee details will be displayed as **NULL**.

What is Database Engine in SQL Server?

The **SQL Server Database Engine** is the core service for storing, processing, and securing data in SQL Server. It is responsible for managing and executing queries, transactions, and other database operations. This engine runs as a background service, typically starting when the operating system is booted, depending on the configuration set during installation.

Key Components:

1. **SQL Server Database Engine**: Manages the storage, retrieval, and updating of data.
2. **SQL Server Agent**: Automates tasks such as running jobs, managing alerts, and scheduling backups.
3. **Other SQL Server Services**: These include components like SQL Server Reporting Services (SSRS), SQL Server Integration Services (SSIS), and SQL Server Analysis Services (SSAS).

Managing SQL Server Services:

To manage SQL Server services, you can use the **SQL Server Configuration Manager**:

- Start > All Programs > Microsoft SQL Server > Configuration Tools > SQL Server Configuration Manager.

This tool helps in starting, stopping, pausing, or restarting SQL Server services and ensures the database engine and related services are properly configured.

What are the Analysis Services in SQL Server?

SQL Server Analysis Services (SSAS) is a data analysis tool that helps turn raw data into actionable information by providing fast and easy access to that information for decision makers. SSAS provides capabilities for Online Analytical Processing (OLAP) and data mining, allowing users to analyze large amounts of data from multiple sources in real time.

Key Features of SSAS:

1. **OLAP (Online Analytical Processing)**: SSAS allows the design and management of multidimensional structures (cubes) that contain data aggregated from relational databases. It provides quick querying capabilities, enabling faster decision-making.
2. **Data Mining**: SSAS supports data mining algorithms that help uncover patterns and insights from data sources.
3. **Fast Query Responses**: SSAS is designed to provide fast query results, even for large datasets, by pre-aggregating and indexing data in multidimensional cubes.

Components of SSAS Architecture:

1. **Server Architecture**: SSAS operates as a Windows service through the `Msmdsrv.exe` application. It includes:
    - **Security**: Manages user access and permissions.
    - **XMLA Listener**: Handles communication over XML for Analysis (XMLA) protocol.
    - **Query Processor**: Executes queries and returns results to clients.
2. **Client Architecture**: SSAS uses a thin client model, where all queries and calculations are processed by the server. Communication between clients and the server is done through SOAP packets.

What are the Integration Services in SQL Server?

SQL Server Integration Services (SSIS) is a platform for building data integration and workflow solutions, focusing on data extraction, transformation, and loading (ETL) processes, commonly used for data warehousing.

Key Uses of SSIS:

1. **ETL Operations**: SSIS facilitates extracting data from multiple sources, transforming it to fit the desired structure, and loading it into data warehouses or data marts.
2. **Data Integration**: SSIS is used to merge data from various data stores, automating data loading and transformation processes.

Architecture of SSIS:

1. **SSIS Designer**: A graphical tool used to build and design ETL workflows and data integration logic.
2. **Runtime Engine**: Executes the tasks defined in the SSIS package.
3. **Tasks and Executables**: Components that define the operations to be performed in a package, such as data transformations or data loading.
4. **Data Flow Engine and Components**: Manages the flow of data during the ETL process, ensuring data is properly transformed and loaded into the destination.
5. **API/Model**: Provides a programmatic interface for integrating SSIS with other applications.
6. **Integration Services Service**: Manages and executes SSIS packages.
7. **SQL Server Import and Export Wizard**: A tool for simple data migrations and transformations.

SSIS simplifies and automates data management tasks, making it an essential tool for large-scale data integration and transformation projects.

What are the data quality services in SQL Server?

SQL Server Data Quality Services (DQS) is a feature introduced in SQL Server 2012 that provides tools and features for managing data quality. It helps in tasks such as data cleansing, standardization, enrichment, and de-duplication.

Key Features of Data Quality Services:

1. **Data Cleansing**: DQS enables the identification and correction of data errors or inconsistencies using a knowledge base and reference data services.
2. **Matching**: DQS allows you to identify and merge duplicate records, improving the accuracy and consistency of your data.
3. **Reference Data Services**: DQS can connect to cloud-based reference data providers to enrich data with accurate and up-to-date information.
4. **Profiling and Monitoring**: DQS integrates data profiling capabilities that help in analyzing the integrity of the data before and after performing cleansing operations.
5. **Knowledge Base**: DQS uses a knowledge base that contains rules and patterns about data, enabling it to automatically correct or standardize data.

Components of Data Quality Server:

- **DQS_MAIN**: Contains stored procedures, DQS engine, and published knowledge bases.
- **DQS_PROJECTS**: Stores data needed for knowledge base management and DQS project activities.
- **DQS_STAGING_DATA**: An intermediate staging database used for processing source data before exporting the cleaned data.

What are the reporting services in SQL Server?

SQL Server Reporting Services (SSRS) is a comprehensive platform for building and delivering reports. It provides tools for report creation, management, and deployment, and offers a scalable architecture for generating reports from various data sources.

Key Components of SQL Server Reporting Services:

1. **Processors**: Components that process the reports, execute queries, and retrieve data.
2. **Extensions**: SSRS includes extensions to interact with different data sources and deliver reports in various formats.

Tools and Components in SSRS Architecture:

- **Report Builder**: A tool used for creating ad-hoc reports with a user-friendly interface.
- **Report Designer**: A more advanced tool used to create reports in Visual Studio, offering additional features for report customization.
- **Report Manager**: The web-based interface for managing and viewing reports.
- **Report Server**: The server that processes and delivers reports.
- **Report Server Database**: Stores metadata, reports, and subscription information.
- **Data Sources**: Connections to external databases or other data providers from which reports are generated.

SSRS enables organizations to generate, manage, and distribute reports across various formats, such as PDF, Excel, and HTML, and supports interactive features like drill-down and parameterized reports.

What are the master data services in SQL Server?

Master Data Services (MDS) is a feature in SQL Server designed to help organizations manage and maintain master data across various business functions. MDS provides a centralized platform to ensure data consistency, accuracy, and integrity for key business data.

Key Features of Master Data Services:

1. **Master Data Hub**: MDS acts as a central repository for storing and managing master data, addressing operational and analytical challenges related to data governance and integrity.
2. **Components of MDS**:
    - **Master Data Services Configuration Manager**: Used for configuring the MDS database, Web application, and Web service.
    - **Web Application (Master Data Manager)**: Provides the interface for data stewardship and allows users to manage master data, apply business rules, and review changes.
3. **Data Stewardship**: Business users with appropriate permissions can create, review, and update master data. They can also manage changes and define business rules that validate the data.
4. **Model Objects**:
    - **Model**: A container for all master data, which includes entities, members, and attributes.
    - **Entity**: Represents a table and contains **members**, which are individual records or rows.
    - **Members**: The actual master data (e.g., customer records) that are managed in MDS. Each member has attributes (like columns in a table).
5. **Master Data Maintenance**: Users can create, edit, and update **leaf members** (individual data points) and **consolidated members** (grouped data). These actions help ensure that master data is kept accurate and up to date.

What is replication in SQL Server?

Replication in SQL Server refers to the process of copying and distributing data from one database to another. It ensures that the data is synchronized across multiple servers, providing redundancy and high availability.

Key Features of Replication:

1. **Data Synchronization**: Replication ensures that changes to data on one server are propagated to other servers in the replication set.
2. **High Availability and Redundancy**: By maintaining multiple copies of the same data on different servers, replication helps protect against server failures, ensuring that data remains accessible.
3. **Scalability and Load Balancing**: Replication can be used to increase the read capacity of a system. By having read replicas, the load can be distributed across multiple servers, improving performance.
4. **Distributed Applications**: Replication can be used to store copies of data in different data centers, improving the availability and locality of data for distributed applications.

Types of Replication:

- **Snapshot Replication**: Captures the data at a specific point in time and applies it across all subscribers.

Example: Snapshot Replication

1. **Step 1**: Open the **Replication** node in SQL Server Management Studio (SSMS) and select **Local Publications**.
2. **Step 2**: Right-click **Local Publications** and select **New Publication**.
3. **Step 3**: Follow the prompts in the wizard to configure and create the publication. The next steps will guide you through selecting the articles (data) to replicate, setting up the distributor, and configuring the subscribers.

How to select data from an SQL Server table?

To select data from an SQL Server table, you use the `SELECT` statement. Here's how to select all columns or specific columns and apply different conditions and sorting:

1. Select All Rows and Columns

To select all rows and columns from a table, you can use the following query:

```
SELECT * FROM HumanResources.Employee;
```

This query retrieves all data from the `Employee` table in the `HumanResources` schema.

2. Select Specific Columns

If you want to select specific columns from a table, list them by name instead of using `*`. For example:

```
SELECT BusinessEntityID, NationalIDNumber, LoginID, JobTitle FROM HumanResources.Employee;
```

This query retrieves only the specified columns from the `Employee` table.

3. Generate Query Automatically in SQL Server

You can also generate the `SELECT` query automatically in SQL Server Management Studio (SSMS).

- Right-click the table (e.g., `HumanResources.Employee`) in **Object Explorer**.
- Select **Script Table as** > **SELECT To** > **New Query Editor Window**. SQL Server will generate the `SELECT` statement for you.

4. Selecting Distinct Rows

To retrieve only distinct (unique) rows, use the `DISTINCT` keyword:

```
SELECT DISTINCT JobTitle FROM HumanResources.Employee;
```

This query will return only unique job titles from the `Employee` table.

5. Filtering Rows with WHERE Clause

To filter rows based on a condition, use the `WHERE` clause:

```
SELECT * FROM HumanResources.Employee WHERE JobTitle = 'Sales Representative';
```

This query retrieves all rows where the `JobTitle` is 'Sales Representative'.

6. Sorting Rows with ORDER BY

To sort the result set, use the `ORDER BY` clause:

```
SELECT * FROM HumanResources.Employee ORDER BY HireDate DESC;
```

This query will sort the employees by `HireDate` in descending order.

What is a check in SQL?

A **Check Constraint** ensures that values in a column satisfy a specified condition. It helps enforce **Domain Integrity**, meaning that the values in a column must meet certain criteria.

Syntax:

```
CREATE TABLE TableName (
    Column1 DataType CHECK (Condition),
    Column2 DataType
);
```

Example:

```
CREATE TABLE emp (
    empId INT CHECK (empId > 10),
    empName VARCHAR(15)
);
```

In this example, the `empId` column is constrained to only accept values greater than 10.

Inserting Invalid Data:

If you try to insert a value that does not meet the check condition, an error will occur:

```
INSERT INTO emp VALUES (8, 'John Doe');
```

This will result in an error because the `empId` must be greater than 10.

Dropping a Check Constraint:

To remove a **Check Constraint**, you first need to know the constraint's name. You can use the following command:

```
EXEC sp_help 'emp';
```

This will provide details about the `emp` table, including the names of any constraints.

What is a default in SQL?

The **Default Constraint** in SQL is used to assign a default value to a column when a new row is inserted, and no specific value is provided for that column. This ensures that the column will have a predefined value if no explicit value is supplied during insertion.

The Default Constraint is not the same as a **Not Null** constraint. It does not prevent NULL values unless explicitly defined by other constraints. Instead, it allows the column to accept NULL values but ensures that, if no value is entered, a default value will be automatically assigned.

Key Points of the Default Constraint:

- It sets a default value only when a new row is created, and no value is provided for that column.
- It does not restrict NULL entries unless combined with the Not Null constraint.
- It is useful for providing a fallback value in case a user does not specify a value during data insertion.

Example of Default Constraint:

```
CREATE TABLE Student (
    StudId SMALLINT PRIMARY KEY,
    StudName VARCHAR(50) NOT NULL,
    Class TINYINT DEFAULT 1
);
```

In this example, if no value is provided for the `Class` column during an insert, the value `1` will be assigned by default.

How to create a database using SQL?

To create a new database, you can use the `CREATE DATABASE` statement. This will create a database structure that can hold various objects such as tables, views, queries, and reports.

Syntax:

```
CREATE DATABASE DatabaseName;
```

Example:

```
CREATE DATABASE Student;
```

Alternatively, you can also create a database using SQL Server Management Studio (SSMS) by:

1. Right-clicking the **Databases** node in Object Explorer.
2. Selecting **New Database** and filling out the necessary details in the wizard.

What is a constraint in SQL?

Constraints in SQL are rules that ensure the validity and integrity of data within a table. These rules enforce data consistency and help prevent errors when inserting, updating, or deleting records. SQL Server supports several types of constraints, each serving a different purpose for maintaining data quality and relationships between tables.

Types of Constraints:

1. **Primary Key Constraint**:
    - A primary key uniquely identifies each record in a table. It ensures that the column(s) marked as primary key contain unique, non-null values.
    - **Example:**

```
CREATE TABLE Student (
    StudId SMALLINT PRIMARY KEY,
    StudName VARCHAR(50),
    Class TINYINT
);
```

2. **Foreign Key Constraint**:
    - A foreign key enforces a relationship between columns in two tables. It ensures that the value in one table matches a value in another table's primary key.
    - **Example:**

```
CREATE TABLE TotalMarks (
    StudentId SMALLINT,
    TotalMarks SMALLINT,
    FOREIGN KEY (StudentId) REFERENCES Student(StudId)
);
```

3. **Not Null Constraint**:
    - The Not Null constraint ensures that a column does not accept NULL values. It guarantees that data is always present for a specified column.
    - **Example:**

```
CREATE TABLE Student (
    StudId SMALLINT PRIMARY KEY,
    StudName VARCHAR(50) NOT NULL,
    Class TINYINT
);
```

4. **Unique Constraint**:
    - The Unique constraint ensures that all values in a column (or a group of columns) are different. Unlike the primary key, a unique column can accept NULL values.
    - **Example:**

```
CREATE TABLE Student (
    StudId SMALLINT PRIMARY KEY,
    StudName VARCHAR(50) UNIQUE,
    Class TINYINT
);
```

5. **Default Constraint**:
    - The Default constraint sets a default value for a column if no value is provided during an insert operation. It does not prevent NULL values unless combined with a Not Null constraint.
    - **Example:**

```
CREATE TABLE Student (
    StudId SMALLINT PRIMARY KEY,
    StudName VARCHAR(50) NOT NULL,
    Class TINYINT DEFAULT 1
);
```

6. **Check Constraint**:
    - The Check constraint ensures that all values in a column satisfy a specified condition. If the data does not meet the condition, the insert or update operation is rejected.
    - **Example:**

```
CREATE TABLE Student (
    StudId SMALLINT PRIMARY KEY,
    StudName VARCHAR(50) NOT NULL,
    Class TINYINT CHECK (Class BETWEEN 1 AND 12)
);
```

Example Tables:

To demonstrate constraints, let's create two tables: `Student` and `TotalMarks`.

```
CREATE TABLE Student (
    StudId SMALLINT,
    StudName VARCHAR(50),
    Class TINYINT
);

CREATE TABLE TotalMarks (
    StudentId SMALLINT,
    TotalMarks SMALLINT
);
```

Data Integrity:

- Data integrity refers to the accuracy and consistency of data within a database. It involves ensuring that the data entered is accurate and valid, adhering to the defined constraints and business rules. This is enforced by constraints such as `PRIMARY KEY`, `FOREIGN KEY`, `CHECK`, and others.

Auto Increment:

- The **Auto Increment** feature automatically generates a unique value for a column (usually the primary key) whenever a new record is inserted. In SQL Server, this is implemented using the `IDENTITY` keyword, while in Oracle, the `AUTO_INCREMENT` keyword is used.

Clustered vs. Non-Clustered Index:

- **Clustered Index**: Defines the order in which data is physically stored in a table. A table can have only one clustered index.
- **Non-Clustered Index**: Creates a separate structure from the table, which references the table's data. A table can have multiple non-clustered indexes.

Data Warehouse:

- A **Data Warehouse** is a centralized repository that stores data from multiple sources. It is used for reporting, analysis, and business intelligence. Data within a data warehouse is often processed into a subset called **Data Marts**, which focus on specific business areas.

How do I define constraints in SQL?

Constraints in SQL are rules and restrictions applied to columns or tables to ensure the integrity, accuracy, and reliability of the data stored. By defining constraints, you control what data can be entered into a table, thus maintaining the consistency and validity of the data.

Types of Constraints:

SQL Server supports two primary categories of constraints:

1. **Column-level constraints**: These are defined when creating or modifying a column in a table.
2. **Table-level constraints**: These are defined after the table has been created, usually using the `ALTER` command.

Six Types of Constraints in SQL Server:

1. **NOT NULL Constraint**:
    - Ensures that a column cannot have a `NULL` value. This constraint is useful when certain columns must always contain data.

    Example:

    ```
    CREATE TABLE My_Constraint (
        IID INT NOT NULL,
        Name NVARCHAR(50) CONSTRAINT Cons_NotNull NOT NULL,
        Age INT NOT NULL
    );
    ```

    Table Level:

```
ALTER TABLE My_Constraint
ALTER COLUMN IID INT NOT NULL;
```

2. **CHECK Constraint**:
   - Enforces a condition to ensure that the data inserted into a column satisfies a specific condition. If the condition is not met, the data is rejected.

   Example:

   ```
   CREATE TABLE My_Constraint (
       Salary INT CHECK (Salary > 5000)
   );
   ```

   Table Level:

   ```
   ALTER TABLE My_Constraint
   ADD CONSTRAINT Check_Constraint CHECK (Age > 50);
   ```

3. **DEFAULT Constraint**:
   - Specifies a default value for a column when no value is provided during insertion.

   Example:

   ```
   CREATE TABLE My_Constraint (
       Status NVARCHAR(50) DEFAULT 'Active'
   );
   ```

4. **UNIQUE Constraint**:
   - Ensures that all values in a column are unique, meaning no duplicate values are allowed.

   Example:

   ```
   CREATE TABLE My_Constraint (
       ID INT UNIQUE
   );
   ```

5. **PRIMARY KEY Constraint**:
   - Uniquely identifies each record in a table. A primary key constraint automatically enforces both uniqueness and non-nullability on the column.

   Example:

   ```
   CREATE TABLE My_Constraint (
       ID INT PRIMARY KEY
   );
   ```

6. **FOREIGN KEY Constraint**:
   - Enforces a relationship between two tables. A foreign key in one table points to the primary key in another table, ensuring referential integrity.

   Example:

   ```
   CREATE TABLE Orders (
       OrderID INT PRIMARY KEY,
       CustomerID INT,
       CONSTRAINT FK_Customer FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
   );
   ```

- **Column-level constraints** are defined directly on individual columns, while **table-level constraints** are defined after the table is created.
- Each type of constraint helps to enforce specific rules, such as uniqueness, referential integrity, and data consistency, ensuring the accuracy and integrity of data in the database.

What is the meaning of Not Null in SQL?

The **NOT NULL** constraint in SQL is used to ensure that a column cannot have a `NULL` value. When a column is marked as **NOT NULL**, it must always contain a value when a record is inserted or updated. This constraint is useful for ensuring that important data, such as a student's name or an employee's ID, is always present.

Example of NOT NULL Constraint:

Consider a table of students where the `StudentName` column cannot have `NULL` values.

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    StudentName NVARCHAR(100) NOT NULL
);
```

In this example, the `StudentName` column cannot have `NULL` entries. Any attempt to insert a record without a student name will result in an error.

How to Add a NOT NULL Constraint to an Existing Table:

To add the `NOT NULL` constraint to an existing column, you can use the `ALTER TABLE` statement:

```
ALTER TABLE Students
MODIFY StudentName NVARCHAR(100) NOT NULL;
```

This statement will modify the `StudentName` column to enforce that it cannot accept `NULL` values.

How to alter a table schema in SQL Server?

To modify an existing table schema in SQL Server, you can use the `ALTER TABLE` statement. This allows you to add, modify, or drop columns, constraints, and other table properties.

Syntax for ALTER TABLE:

```
ALTER TABLE table_name
ADD column_name data_type [constraints];
```

Example of Altering a Table:

1. **Adding a New Column:**

```
ALTER TABLE Stock
ADD Quantity SMALLINT UNSIGNED NOT NULL;
```

This statement adds a new `Quantity` column to the `Stock` table and ensures it cannot contain `NULL` values.

2. **Modifying an Existing Column:**

```
ALTER TABLE Stock
MODIFY ID SMALLINT UNSIGNED NOT NULL;
```

This statement modifies the `ID` column to ensure it is non-nullable.

3. **Adding a Primary Key:**

```
ALTER TABLE Stock
ADD PRIMARY KEY (ID);
```

This statement adds a primary key constraint to the `ID` column, ensuring that each `ID` value is unique.

These operations are used when you need to adjust the structure of your database tables after they have been created.

How to create index in SQL Server?

In SQL Server, indexes are used to improve the speed of data retrieval operations. They are especially useful for large tables with frequent SELECT queries and WHERE clauses. However, they can negatively affect performance on INSERT, UPDATE, and DELETE operations because they require extra processing to maintain the index.

Types of Indexes

1. **Simple Index**: Created on a single column.
    - **Example**:

```
CREATE INDEX i_select ON emp(empName);
```

2. **Composite Index**: Created on multiple columns to optimize queries that involve multiple columns.
    - **Example**:

```
CREATE INDEX i_select ON emp(empId, empName);
```

3. **Unique Index**: Ensures that all values in the indexed column(s) are unique. It is used to enforce data integrity and prevent duplicate values.
    - **Example**:

```
CREATE UNIQUE INDEX i_unique ON emp(empId);
```

How to Get Unique Records in SQL?

To retrieve unique records from a table, we can use the **DISTINCT** keyword or enforce a unique constraint on a column.

Using DISTINCT

The `DISTINCT` keyword is used in a `SELECT` query to eliminate duplicate records in the result set.

- **Example**:

```
SELECT DISTINCT empName FROM emp;
```

This will return only unique employee names from the `emp` table.

Unique Constraint

A **Unique Constraint** ensures that a column contains unique values across all rows in the table. It allows a maximum of one `NULL` value per column.

- **Syntax**:

```
CREATE TABLE Table_Name (
    Column_Name Datatype CONSTRAINT Constraint_Name UNIQUE
);
```

- **Example**:

```
CREATE TABLE MY_Tab (
    IId INT CONSTRAINT Unique_Cons UNIQUE,
    Name NVARCHAR(50)
);
```

This example creates a table `MY_Tab` with a unique constraint on the `IId` column, ensuring that all values in `IId` are unique.

How to create a date column in SQL Server?

In SQL Server, you can create a `datetime` or `smalldatetime` column to store date and time values. The `datetime` type has a larger range and more precision than `smalldatetime`.

Datetime Data Type

- **Range**: January 1, 1753, to December 31, 9999
- **Precision**: Up to 0.003 seconds

Smalldatetime Data Type

- **Range**: January 1, 1900, to June 6, 2079
- **Precision**: 1 second

Creating a Table with a Date Column

To create a table with a `datetime` column and insert date values:

```
CREATE TABLE tbDate (
    col DATETIME
);

GO

INSERT INTO tbDate VALUES ('8:00 AM');
GO

INSERT INTO tbDate VALUES ('March 24, 2008');
GO
```

Formatting Date Values

You can use the `CONVERT` function to format `datetime` values in different styles. Below is a list of styles that can be used when converting dates from `datetime` to character data:

Style ID	Style Type
0 or 100	mon dd yyyy hh:miAM (or PM)
101	mm/dd/yy
102	yy.mm.dd
103	dd/mm/yy
104	dd.mm.yy
105	dd-mm-yy
106	dd mon yy
107	Mon dd, yy
108	hh:mm:ss
9 or 109	mon dd yyyy hh:mi:ss:mmmAM (or PM)
110	mm-dd-yy
111	yy/mm/dd
112	Yymmdd
13 or 113	dd mon yyyy hh:mm:ss:mmm(24h)
114	hh:mi:ss:mmm(24h)
20 or 120	yyyy-mm-dd hh:mi:ss(24h)
21 or 121	yyyy-mm-dd hh:mi:ss.mmm(24h)
126	yyyy-mm-ddThh:mm:ss.mmm(no spaces)
130	dd mon yyyy hh:mi:ss:mmmAM
131	dd/mm/yy hh:mi:ss:mmmAM

Example Usage of `CONVERT`

```
SELECT CONVERT(VARCHAR, col, 103) AS FormattedDate
FROM tbDate;
```

This would format the date in `dd/mm/yy` style, based on the value stored in the `col` column.

What is ACID fundamental? What are transactions in SQL SERVER?

In SQL Server, a transaction is a sequence of operations that are executed as a single unit of work. This ensures that the transaction adheres to the **ACID** properties, which guarantee reliable processing of database transactions:

ACID Properties

1. **Atomicity**
    - **Definition**: A transaction is atomic, meaning it is indivisible. Either all operations within the transaction are completed successfully, or none are.
    - **Example**: If a bank transfer involves deducting money from one account and adding it to another, the entire process must succeed or fail together. If one operation fails, the entire transaction is rolled back.
2. **Consistency**
    - **Definition**: A transaction must take the database from one consistent state to another. It ensures that all data integrity rules (constraints, triggers, etc.) are maintained during the transaction.
    - **Example**: A transaction must maintain referential integrity, such as ensuring that no foreign key violations occur.
3. **Isolation**
    - **Definition**: Modifications made by one transaction should not be visible to others until the transaction is complete. Each transaction is isolated from others to prevent data anomalies.
    - **Example**: If two transactions are running concurrently, one transaction should not see uncommitted changes from the other transaction.
4. **Durability**
    - **Definition**: Once a transaction is committed, its changes are permanent, even in the event of a system failure.
    - **Example**: If a transaction is committed to the database, the changes (e.g., inserted, updated, or deleted records) will remain intact even if the system crashes after the commit.

Transactions in SQL Server

- **Definition**: A transaction is a sequence of SQL commands executed as a single unit. SQL Server ensures that these transactions follow the ACID properties.
- **Transaction Control Commands**:
    - **BEGIN TRANSACTION**: Marks the start of a transaction.
    - **COMMIT**: Saves all changes made during the transaction to the database.
    - **ROLLBACK**: Reverts all changes made during the transaction if an error occurs or if the transaction is explicitly rolled back.

Example of a Transaction in SQL Server

```
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountID = 1;

UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountID = 2;

COMMIT; -- Commit transaction if all operations succeed
-- ROLLBACK; -- Rollback transaction if an error occurs
```

- If any of the updates fail, a `ROLLBACK` can be issued to revert the changes and maintain data integrity.

What is a candidate key?

A **candidate key** is any column or combination of columns in a table that can uniquely identify each row in that table. A table can have multiple candidate keys, and one of them is chosen as the **primary key**. The rest are known as **alternate keys**.

Example:

- In a `Supplier` table, both `supplierid` and `suppliername` might uniquely identify a supplier. Hence, they are both **candidate keys**. However, you would typically choose `supplierid` as the **primary key** for uniqueness and efficiency.

How do GROUP and ORDER BY Differ?

The **ORDER BY** and **GROUP BY** clauses in SQL serve different purposes but can often be used together. Here's how they differ:

ORDER BY

- **Purpose**: It is used to sort the result set based on one or more columns.
- **Function**: It arranges the data in either ascending or descending order.
- **Usage**: It does not alter the data itself but only the display order.
- **Example**:

```
SELECT SalesOrderID, ProductID, OrderQty * UnitPrice AS ExtendedPrice
FROM Sales.SalesOrderDetail
ORDER BY SalesOrderID;
```

- This query will return all rows from the `SalesOrderDetail` table, sorted by `SalesOrderID` in ascending order.

GROUP BY

- **Purpose**: It groups rows that have the same values into summary rows. This allows you to perform aggregation functions (like SUM, COUNT, AVG, etc.) on grouped data.
- **Function**: It reduces the number of rows by combining data based on the grouping columns.
- **Usage**: Used when you want to aggregate data or summarize information based on certain columns.
- **Example**:

```
SELECT SalesOrderID, SUM(OrderQty * UnitPrice) AS TotalPrice
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID;
```

- This query groups the rows by `SalesOrderID` and calculates the total price for each order by summing `OrderQty * UnitPrice`.

Differences

- **ORDER BY** is used to **sort** the result set by specific columns.
- **GROUP BY** is used to **group** data based on one or more columns and perform aggregation on other columns (e.g., SUM, AVG, COUNT).

Both clauses can be used together. After using `GROUP BY` to summarize data, you can use `ORDER BY` to sort the results, as shown below:

```
SELECT SalesOrderID, SUM(OrderQty * UnitPrice) AS TotalPrice
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
ORDER BY TotalPrice;
```

- In this query, `GROUP BY` groups the results by `SalesOrderID` and calculates the total price for each order, and `ORDER BY` sorts those results by `TotalPrice`.

Key Differences Summary

- **ORDER BY**: Sorts the final result by specified columns.
- **GROUP BY**: Creates unique combinations of columns and aggregates data, often with the use of functions like `SUM`, `COUNT`, `AVG`, etc.

Compare SQL & PL/SQL

Criteria	SQL	PL/SQL
What it is	Single query or command execution	Full programming language
What it comprises	Data source for reports, web pages	Application language to build, format, and display reports, web pages
Characteristic	Declarative in nature	Procedural in nature
Used for	Manipulating data	Creating applications

SQL is primarily used for querying and manipulating data directly, whereas PL/SQL is a full programming language that allows for procedural logic, creating applications, and handling complex logic with conditions and loops.

What is BCP? When is it used?

BCP (Bulk Copy Program) is a utility tool in SQL Server that allows users to copy large amounts of data between SQL Server instances and external data files, typically in CSV or other formats. It is often used for:

- Exporting large datasets from tables and views.

- Importing large datasets from external files into SQL Server tables or views.

Difference from BULK INSERT:

- **BCP** is used for exporting and importing data directly from or to files in bulk, while **BULK INSERT** is used within SQL Server to load data into a table directly from a file.

When is the UPDATE_STATISTICS command used?

The **UPDATE STATISTICS** command is used to update statistics about the distribution of values in indexed columns and other columns involved in query optimization. It ensures that SQL Server uses the most accurate and up-to-date data when creating query execution plans. This command is used when:

- After processing a large amount of data (inserts, deletes, updates).
- When deleting a large number of rows or making substantial modifications to the table.
- To ensure that SQL Server can accurately optimize queries and improve performance by adjusting query plans based on the latest data.

Explain the steps needed to Create a scheduled job?

To create a scheduled job in SQL Server, follow these steps:

1. **Connect to the Database**: Open SQL Server Management Studio (SSMS) and connect to the desired SQL Server instance.
2. **Locate SQL Server Agent**: In Object Explorer, find the **SQL Server Agent** node. Right-click on **Jobs** and select **New Job**.
3. **Job Details**: In the **New Job** window, provide a name and description for the job.
4. **Add Steps**: Under the **Steps** tab, define the actions that the job will perform. You can specify SQL statements or stored procedures.
5. **Set Schedule**: Under the **Schedules** tab, define when the job should run (e.g., daily, weekly, or on-demand).
6. **Notifications**: Optionally, configure notifications to alert when the job completes, fails, or if any other condition occurs.

When are we going to use TRUNCATE and DELETE?

Here's a comparison of **TRUNCATE** and **DELETE** commands:

1. **Type of Command**:
    - **TRUNCATE** is a **DDL** (Data Definition Language) command.
    - **DELETE** is a **DML** (Data Manipulation Language) command.
2. **Triggers**:
    - **TRUNCATE** does not activate triggers.
    - **DELETE** can activate triggers.
3. **Speed**:
    - **TRUNCATE** is faster than **DELETE** because it doesn't log individual row deletions in the transaction log (no rollback of individual rows).
    - **DELETE** logs every row deletion, making it slower and allowing for rollback.
4. **Where Clause**:
    - **TRUNCATE** cannot be used with a **WHERE** clause.
    - **DELETE** allows a **WHERE** clause to selectively delete rows.
5. **Foreign Key Constraints**:
    - **TRUNCATE** cannot be used on a table with foreign key constraints.
    - **DELETE** can be used on tables with foreign key constraints (if the constraints are not violated).

Use TRUNCATE when you need to remove all rows from a table quickly and efficiently, especially when no foreign key constraints or triggers are involved. **Use DELETE** when you need to delete specific rows or when you want to activate triggers or maintain the ability to rollback individual deletions.

Explain correlated query work?

A **correlated subquery** is a type of subquery in which the inner query (subquery) depends on the outer query for its values. The subquery is evaluated once for each row processed by the outer query.

How it works:

1. The outer query processes one row at a time.
2. For each row, the inner subquery is executed. It refers to the outer query's current row and uses its values in the subquery.
3. The subquery will be re-executed for each row from the outer query.
4. The correlation condition (e.g., `Emp1.Salary`) in the subquery ensures it evaluates with each row of the outer query.

Example:

```
SELECT Emp1.Name, Emp1.Salary
FROM Employees Emp1
WHERE Emp1.Salary >
    (SELECT AVG(Emp2.Salary)
     FROM Employees Emp2
     WHERE Emp1.Department = Emp2.Department);
```

In this case, for each row in the outer query (for each employee), the subquery calculates the average salary for that employee's department.

When is the Explicit Cursor Used?

An **explicit cursor** is used when a developer needs to process a result set row by row. This is often necessary when the application logic requires iterative processing, such as when multiple operations need to be performed on each row.

Steps involved:

1. **Declare the Cursor**: The cursor is explicitly defined by the developer with a `DECLARE` statement.
2. **Open the Cursor**: The `OPEN` statement is used to execute the query and position the cursor.
3. **Fetch Rows**: The `FETCH` statement retrieves individual rows one by one.
4. **Close the Cursor**: The `CLOSE` statement is used to release the resources associated with the cursor.
5. **Cursor Attributes**: Attributes like `%FOUND`, `%NOFOUND`, `%ROWCOUNT`, and `%ISOPEN` help in managing cursor operations.

Example:

```
DECLARE employee_cursor CURSOR FOR
    SELECT Name, Salary FROM Employees;
OPEN employee_cursor;
FETCH NEXT FROM employee_cursor INTO @Name, @Salary;
WHILE @@FETCH_STATUS = 0
BEGIN
    -- Processing logic here
    FETCH NEXT FROM employee_cursor INTO @Name, @Salary;
END;
CLOSE employee_cursor;
DEALLOCATE employee_cursor;
```

Find What is Wrong in this Query?

The issue in the original query is the misuse of the `WHERE` clause to filter groups. The `WHERE` clause is meant to filter individual rows, not groups. To filter groups, the `HAVING` clause should be used instead.

Incorrect Query:

```
SELECT subject_code, AVG(marks)
FROM students
WHERE AVG(marks) > 75
GROUP BY subject_code;
```

This query tries to use `AVG(marks)` in the `WHERE` clause, which is invalid because aggregate functions cannot be used in `WHERE` clauses directly.

Corrected Query:

```
SELECT subject_code, AVG(marks)
FROM students
GROUP BY subject_code
HAVING AVG(marks) > 75;
```

The `HAVING` clause correctly filters the results after the `GROUP BY` operation has been applied, allowing the average marks to be compared.

Write the Syntax for STUFF function in an SQL server?

The syntax for the **STUFF** function in SQL Server is:

```
STUFF (String1, Position, Length, String2)
```

- **String1**: The original string in which characters will be replaced.
- **Position**: The starting position in **String1** where the replacement will begin.
- **Length**: The number of characters to remove from **String1** starting at **Position**.
- **String2**: The string that will replace the characters in **String1**.

Example:

```
SELECT STUFF('Hello World', 7, 5, 'SQL');
-- Output: 'Hello SQL'
```

In this example, "World" is replaced by "SQL" starting from the 7th character.

Name some commands that can be used to manipulate text in T-SQL code.

Here are some T-SQL commands used for text manipulation:

- `CHARINDEX(findTextData, textData, [startingPosition])` : Finds the position of a substring in a string.
- `LEFT(character_expression, integer_expression)` : Extracts a specified number of characters from the left side of a string.
- `LEN(textData)` : Returns the length of a string, excluding trailing spaces.
- `LOWER(character_expression)` : Converts a string to lowercase.
- `LTRIM(textData)` : Removes leading spaces from a string.
- `PATINDEX(findTextData, textData)` : Finds the position of a pattern in a string.
- `REPLACE(textData, findTextData, replaceWithTextData)` : Replaces occurrences of a substring with another string.
- `REPLICATE(character_expression, integer_expression)` : Repeats a string a specified number of times.
- `REVERSE(character_expression)` : Reverses the characters in a string.
- `RTRIM(textData)` : Removes trailing spaces from a string.
- `SPACE(numberOfSpaces)` : Returns a string with a specified number of spaces.
- `STUFF(textData, start, length, insertTextData)` : Deletes characters in a string and inserts new ones at a specified position.
- `SUBSTRING(textData, startPosition, length)` : Extracts a part of a string.
- `UPPER(character_expression)` : Converts a string to uppercase.

What are the three ways that Dynamic SQL can be executed?

Dynamic SQL in SQL Server can be executed in the following three ways:

1. **Writing a query with parameters**: A SQL query with dynamic parts can be written using parameters to pass dynamic values.
2. **Using `EXEC`** : The `EXEC` command can execute dynamic SQL by passing the SQL string to it.

   Example:

   ```
   EXEC('SELECT * FROM ' + @tableName);
   ```

3. **Using `sp_executesql`** : A system stored procedure that allows execution of dynamically built SQL strings and provides better parameterization.

   Example:

   ```
   EXEC sp_executesql N'SELECT * FROM @tableName', N'@tableName NVARCHAR(50)', @tableName;
   ```

In what version of SQL Server were synonyms released? How do synonyms work and explain its use cases?

Synonyms were introduced in **SQL Server 2005**.

A **synonym** is an alias or reference to another database object such as a table, view, stored procedure, or function. Synonyms can be used to simplify migrations, testing, and to reference objects across different databases, servers, or schemas without modifying application code.

How Synonyms Work:

- A synonym allows an object to be referenced by a different name.
- The synonym can reference an object in a different database or server, enabling seamless referencing without altering SQL queries.

Use Cases:

1. **Database Migrations**: You can use synonyms to point to different underlying objects during a migration without changing the code that references those objects.
2. **Simplifying Queries**: By using synonyms, applications can reference objects without needing to know their exact location.
3. **Re-architecture or Upgrade Projects**: Synonyms are useful when migrating from one database structure to another, allowing the code to remain intact while the underlying structure changes.

If you are a SQL Developer, how can you delete duplicate records in a table with no primary key?

To delete duplicate records in a table with no primary key, you can use the `SET ROWCOUNT` command. The approach works by limiting the number of rows affected by the `DELETE` command to one duplicate row at a time. After deleting one duplicate row, you can reset `ROWCOUNT` to zero to remove the restriction.

Example:

```
SET ROWCOUNT 1;
DELETE FROM YourTable
WHERE [ColumnName] IN (SELECT [ColumnName] FROM YourTable GROUP BY [ColumnName] HAVING COUNT(*) > 1);
SET ROWCOUNT 0;
```

In this example, only one duplicate record will be deleted at a time, and `SET ROWCOUNT 0` ensures that the restriction is removed after the operation.

Is it possible to import data directly from T-SQL commands without using SQL Server Integration Services? If so, what are the commands?

Yes, it is possible to import data directly from T-SQL commands without using SQL Server Integration Services. The following commands can be used for importing data:

- **BCP (Bulk Copy Program)**: A command-line utility that enables the insertion of large amounts of data from a file into a SQL Server table.
- **BULK INSERT**: A T-SQL command used to load data from a file into a table or view. This command is typically used for importing large datasets efficiently.
- **OPENROWSET**: A function that can be used in the `FROM` clause of a query to reference data from an external data source, such as a file or another SQL Server instance.
- **OPENDATASOURCE**: Allows ad hoc access to external data sources by providing connection information inline as part of a four-part object name.
- **OPENQUERY**: Executes a pass-through query against a linked server and retrieves the result set as if it were a table in the query.
- **Linked Servers**: By setting up a linked server, SQL Server can execute commands on external OLE DB data sources (like Oracle, another SQL Server instance, etc.).

What is the native system stored procedure to execute a command against all databases?

The native system stored procedure to execute a command against all databases is **`sp_MSforeachdb`**. It allows you to execute a command across all databases in the SQL Server instance. The `@Command` parameter is used to specify the T-SQL command, and the `?` placeholder represents the current database name during execution.

Example:

```
EXEC sp_MSforeachdb 'USE [?]; SELECT name FROM sys.tables';
```

Alternatively, you can use a **cursor** to iterate over each database and execute specific commands.

How can a SQL Developer prevent T-SQL code from running on a production SQL Server?

A SQL Developer can prevent T-SQL code from running on a production SQL Server by using **IF** logic along with the `@@SERVERNAME` function. This allows the developer to compare the server's name against a specific string, and if it matches the production environment, a `RETURN` statement can halt the execution of the code before any other logic runs.

Example:

```
IF @@SERVERNAME = 'ProductionServer'
BEGIN
    RETURN
END
```

This way, the code will not execute if the server is identified as a production server.

How do you maintain database integrity where deletions from one table will automatically cause deletions in another table?

To maintain database integrity, you can create a **foreign key** with the **ON DELETE CASCADE** option. This ensures that when a record in the parent table is deleted, corresponding records in the child table are automatically deleted as well.

Example:

```
CREATE TABLE ParentTable (
    ParentID INT PRIMARY KEY,
    Name VARCHAR(100)
);

CREATE TABLE ChildTable (
    ChildID INT PRIMARY KEY,
    ParentID INT,
    Name VARCHAR(100),
    FOREIGN KEY (ParentID) REFERENCES ParentTable(ParentID) ON DELETE CASCADE
);
```

With the above setup, when a record from the `ParentTable` is deleted, any corresponding record in the `ChildTable` that references it will be automatically removed.

What port does SQL Server run on?

The standard port for SQL Server is **1433**. This is the default port used for client connections to the database server. However, it can be configured to run on a different port if needed.

What is the SQL CASE statement used for? Explain with an example

The SQL **CASE** statement is used to add conditional logic to a query, allowing you to perform **if-else** checks within a **SELECT** clause. It helps in returning values based on different conditions.

Example:

```
SELECT Employee_Name,
       CASE
           WHEN Location = 'alex' THEN Bonus * 2
           WHEN Location = 'robin' THEN Bonus * 5
           ELSE Bonus
       END AS "New Bonus"
FROM Intellipaat_employee;
```

In this example, the **CASE** statement checks the `Location` column. If it matches 'alex', the bonus is doubled, and if it matches 'robin', the bonus is multiplied by 5. If neither condition is met, the original bonus is returned.

What are the risks of storing a Hibernate-managed object in cache?

The primary risk of storing a Hibernate-managed object in cache is that the object may outlive the session from which it was retrieved. As a result, any lazily loaded properties may not get loaded later when accessed, since they are bound to the session and may not be available outside of it.

To overcome this problem, you can cache the **object's ID** and **class** instead of the whole object. When the object is needed, it can be retrieved within the current session context, ensuring that lazily loaded properties are available as needed.

When is the use of UPDATE_STATISTICS command?

The **UPDATE_STATISTICS** command is used to ensure that SQL Server uses the most up-to-date statistics when generating query plans. Accurate statistics are essential for the SQL Server optimizer to make informed decisions about the most efficient query plan.

However, frequent updates of statistics may lead to query recompilation, which has a performance cost. Therefore, it's essential to balance the frequency of updating statistics with the impact on performance.

Syntax:

```
UPDATE STATISTICS table_or_indexed_view_name
[ { index_or_statistics_name | ( index_or_statistics_name [, ...n] ) } ]
[ WITH { FULLSCAN | SAMPLE number { PERCENT | ROWS } | RESAMPLE
       [ ON PARTITIONS ( { | } [, …n] ) ] } ]
[ , ... ]
[ ALL | COLUMNS | INDEX ]
[ NORECOMPUTE ]
[ INCREMENTAL = { ON | OFF } ]
[ STATS_STREAM = stats_stream ]
[ ROWCOUNT = numeric_constant ]
[ PAGECOUNT = numeric_constant ]
```

This command can be customized to choose the level of sampling or whether it should use a full scan, as well as specify incremental updates and other options. The command helps in maintaining query plan accuracy by keeping statistics current.

What is SQL Profiler?

Microsoft SQL Server Profiler is a tool that provides a graphical user interface for SQL Trace. It allows users to monitor and capture data about events occurring in an instance of SQL Server or Analysis Services. This data can be saved to a file or table for later analysis.

SQL Profiler allows you to:

- Monitor specific events of interest.
- Filter events to reduce trace size and minimize server overhead.
- Analyze the captured data for troubleshooting, performance tuning, or auditing purposes.

Using SQL Profiler, you can focus on particular events, ensuring that only relevant data is collected and analyzed.

What command using Query Analyzer will give you the version of SQL Server and operating system?

To get the version of SQL Server and the operating system, you can run the following command in SQL Server Query Analyzer:

```
SELECT SERVERPROPERTY('productversion'), SERVERPROPERTY('productlevel'), SERVERPROPERTY('edition');
```

This query returns:

- The version of SQL Server (`productversion`).
- The patch level of the SQL Server (`productlevel`).
- The edition of SQL Server (`edition`).

What does it mean to have QUOTED_IDENTIFIER ON? What are the implications of having it OFF?

When **SET QUOTED_IDENTIFIER** is ON:

- Identifiers (such as table names and column names) can be enclosed in double quotation marks (`" "`).
- String literals must be enclosed in single quotation marks (`' '`).

When **SET QUOTED_IDENTIFIER** is OFF:

- Identifiers cannot be enclosed in double quotation marks and must follow standard Transact-SQL rules for naming.
- String literals can still be enclosed in single quotation marks.

The main implication of having **SET QUOTED_IDENTIFIER OFF** is that it limits flexibility in naming, especially when working with reserved keywords or non-standard characters in identifiers.

What is the STUFF function and how does it differ from the REPLACE function in SQL?

- **STUFF function**: This function is used to insert a string into another string by removing a specified number of characters starting from a given position and replacing them with a new string.

  Syntax:

  ```
  STUFF(character_expression, start, length, replaceWith_expression)
  ```

  Example:

  ```
  SELECT STUFF('Intellipaat', 3, 3, 'abc')
  ```

This will return `'Iabcllipaat'`, where it replaces the characters from position 3 to 5 with `'abc'`.

- **REPLACE function**: This function replaces all occurrences of a specified substring with another substring.

  Syntax:

  ```
  REPLACE(string_expression, string_pattern, string_replacement)
  ```

  Example:

  ```
  SELECT REPLACE('Abcabcabc', 'bc', 'xy')
  ```

  This will return `'Axyaxyaxy'`, where it replaces all occurrences of `'bc'` with `'xy'`.

The key difference is that **STUFF** modifies the string at a specific position and for a specified length, while **REPLACE** alters all occurrences of a substring in the entire string.

How to get @@ERROR and @@ROWCOUNT at the same time

To retrieve both `@@ERROR` and `@@ROWCOUNT` at the same time, store them in local variables within the same statement. This ensures that their values are not reset by subsequent statements.

Here's how to achieve this:

```
SELECT @RC = @@ROWCOUNT, @ER = @@ERROR
```

This ensures that both `@@ROWCOUNT` and `@@ERROR` are captured immediately after an operation without being reset by any other statement.

What is de-normalization in SQL database administration? Give examples

De-normalization is the process of introducing redundant data into a database to improve query performance and readability. It is used to reduce the complexity of queries and increase data retrieval speed, often at the cost of storage efficiency and data integrity.

Examples of de-normalization:

- **Materialized views**: These store aggregated or computed data, such as storing the count of "many" objects in a one-to-many relationship.
- **Pre-calculated fields**: For example, storing the total order value directly in an order table instead of calculating it on the fly.
- **Linking attributes**: Combining data from multiple tables into a single table to reduce joins.

De-normalization is often used to optimize performance in specific scenarios, such as reporting and web applications, where speed is a higher priority than storage efficiency.

Can you explain about buffer cache and log cache in SQL Server?

- **Buffer Cache**: The buffer cache is a memory pool where SQL Server stores data pages. When a page is requested, SQL Server first checks the buffer cache. The goal is to avoid disk reads, enhancing performance. An ideal buffer cache should have a high hit rate (e.g., 95% or higher), meaning most data requests are served from memory, reducing disk access.
  - If the buffer cache hit ratio falls below 90%, it suggests that more physical memory may be needed for optimal performance.
- **Log Cache**: The log cache is a memory pool used to store transaction log pages. Log data is written to the log cache before being written to disk. This helps to reduce the synchronization between data and log buffers, improving the overall performance of write operations.

Describe how to use Linked Server

A **Linked Server** in SQL Server allows for connecting to different OLE DB sources, such as Excel, other SQL Servers, or even non-SQL data sources. Here are the steps to create and use a Linked Server, specifically for an MS-Excel workbook:

1. **Open SQL Server Management Studio**.
2. **Expand Server Objects** in the Object Explorer.
3. **Right-click on Linked Servers** and click on **New Linked Server**.
4. On the **General** page:
   - Enter a name for the Linked Server in the first text box.
   - Select **Other Data Source**.
   - Choose **Microsoft Jet 4.0 OLE DB Provider** from the Provider list.
   - In the **Data Source** box, enter the full path and file name of the Excel file.
   - In the **Provider String**, type the Excel version (e.g., **Excel 8.0** for Excel 2000, 2002, or 97).
5. Click **OK** to create the Linked Server.

This setup allows SQL Server to query Excel data directly by treating it as a linked database.

Explain how to send email from SQL database

SQL Server has the **Database Mail** feature that allows sending emails directly from the database without requiring MAPI clients. Here's how to set it up:

1. **Enable Database Mail**: Ensure the SQL Server Mail account is configured correctly.
2. **Create the email script**: The script uses the `sp_send_dbmail` stored procedure to send the email.

Example:

```
USE [YourDB]
EXEC msdb.dbo.sp_send_dbmail
    @recipients = 'xyz@intellipaat.com; abc@intellipaat.com; pqr@intellipaat.com',
    @body = 'A warm wish for your future endeavor',
    @subject = 'This mail was sent using Database Mail';
GO
```

This script sends an email to the recipients with the specified subject and body content.

How to make remote connection in database?

To enable remote connections in SQL Server, follow these steps:

1. **Use SQL Server Surface Area Configuration Tool**: This tool helps enable remote connections.
2. **Access Remote Connections**:
    - Open the Surface Area Configuration for Services and Connections.
    - Navigate to `SQLEXPRESS/Database Engine/RemoteConnections`.
3. **Enable Remote Connections**:
    - Select the option: **Local and Remote Connections**.
    - Choose **Using TCP/IP only** under Local and Remote Connections.
4. **Apply Settings**: Click the **OK** or **Apply** button to save the configuration.

This process ensures that SQL Server allows remote connections via TCP/IP.

What is the purpose of OPENXML clause in SQL Server stored procedure?

The `OPENXML` clause is used to parse and retrieve XML data in SQL Server. It allows XML data to be read and inserted into a relational database in an efficient way. The primary purpose of `OPENXML` is to enable querying and inserting XML data into SQL tables.

- **Usage**: It requires specifying the XPath to the XML element for extracting specific data.
- **Example**: Here's a procedure to retrieve XML data:

```
DECLARE @index int;
DECLARE @xmlString varchar(8000);

SET @xmlString = '<Persons><Person><id>15201</id><Name>abc</Name><PhoneNo>9343463943</PhoneNo></Person><Person><id>15202</id><Name>x

EXEC sp_xml_preparedocument @index OUTPUT, @xmlString;

SELECT *
FROM OPENXML(@index, 'Persons/Person')
WITH (id varchar(10), Name varchar(100) 'Name', PhoneNo varchar(50) 'PhoneNo');

EXEC sp_xml_removedocument @index;
```

- **Output**:

```
id      Name    PhoneNo
15201   abc     9343463943
15202   xyz     9342673212
```

This example parses the XML data and extracts the relevant information into a relational format.

How to store PDF file in SQL Server?

There are two main ways to store a PDF file in SQL Server:

1. **Using a BLOB column**:

- Create a column in your table with the data type `varbinary(max)` or `image` (deprecated, use `varbinary`).
- Store the binary content of the PDF file in this column.

Example:

```
CREATE TABLE Documents (
    DocID INT PRIMARY KEY,
    DocContent VARBINARY(MAX)
);
```

You can then use `INSERT` or `UPDATE` statements to store the binary data of the PDF into the `DocContent` column.

2. **Storing the file path**:
   - Store the file path (link to the PDF) in the database.
   - This method involves storing the PDF file on the filesystem and just referencing the file location in the database.

Explain the use of keyword WITH ENCRYPTION. Create a Store Procedure with Encryption.

The `WITH ENCRYPTION` keyword in SQL Server is used to encrypt the source code of a stored procedure, function, or view. When applied, it prevents users from viewing the original code of the procedure in the system catalog views or database tables. This helps in protecting the intellectual property or logic within the procedure.

Example:

```
CREATE PROCEDURE salary_sum WITH ENCRYPTION
AS
SELECT sum(salary) FROM employee WHERE emp_dept LIKE 'Develop';
GO
```

In this example, the `WITH ENCRYPTION` clause ensures that the stored procedure's SQL code will not be visible to users who do not have access to system tables or database files. However, privileged users such as administrators can still access the encrypted procedure.

What is lock escalation?

Lock escalation is the process of converting smaller, more granular locks (like row or page locks) into a larger, more coarse-grained lock (like a table lock). This process helps to conserve memory, as managing a large number of small locks can be resource-intensive.

- **Purpose**: The main purpose of lock escalation is to improve system performance by reducing the number of locks and memory usage. Too many locks could lead to higher memory consumption.
- **Behavior**: Starting from SQL Server 7.0, lock escalation is dynamically managed by SQL Server. When SQL Server detects a large number of smaller locks, it will automatically escalate them to higher-level locks to optimize performance.

What is Failover clustering overview?

Failover clustering is a high-availability solution that ensures database services remain available even if a system fails. In a failover cluster, two or more servers work together to maintain continuous data access.

- **Primary and Failover Servers**: The primary server runs the basic services, while the secondary server (failover system) takes over in case the primary server fails.
- **Monitoring**: The primary server is monitored to check if it's functioning properly. If it fails, the failover system automatically assumes control to continue providing services.

Failover clustering helps to maintain system uptime by preventing disruptions in case of a server failure.

What is Builtin/Administrator?

The **Builtin/Administrator** account is a system account used primarily during setup and for disaster recovery. It is generally used for joining a machine to a domain and should be disabled immediately after setup.

- **Purpose**: It allows access to administrative tasks and recovery procedures.
- **Usage**: This account should not be used for day-to-day operations and should remain disabled to ensure system security.
- **Automatic Enabling**: If necessary, the Builtin/Administrator account can be automatically enabled for disaster recovery scenarios.

What XML Support Does SQL Server Extend?

SQL Server extends comprehensive XML support both server-side and client-side. Below is a breakdown of the major XML capabilities it offers:

Server-Side XML Support

1. **Creation of XML Fragments**:
    - SQL Server allows the creation of XML data directly from relational data using the `FOR XML` clause in a `SELECT` query. This helps convert query results into XML format.
    - Example:

```
SELECT EmpName, EmpID FROM Employees FOR XML PATH('Employee');
```

2. **Shredding XML Data**:
    - SQL Server provides the ability to shred (parse) XML data, meaning you can break it down and store it in relational tables. This can be done using the `OPENXML` clause, which allows you to convert XML data into a relational result set.
3. **Storing XML Data**:
    - SQL Server can store XML data in columns of type `XML`. This allows you to store and manage XML documents directly within the database.

Client-Side XML Support

1. **XML Views**:
    - SQL Server offers bidirectional mapping between XML schemas and relational tables. This feature enables you to create XML views that map XML data to relational data structures.
2. **Creation of XML Templates**:
    - SQL Server allows the creation of dynamic sections in XML using templates. These templates can be used to structure the XML output dynamically based on the query results.

Additional Features

- **FOR XML Clause**:
    - SQL Server can return XML documents using the `FOR XML` clause, enabling seamless integration of XML data with relational queries.
- **OPENXML Clause**:
    - With the `OPENXML` clause, SQL Server allows you to extract data from XML documents and convert it into a relational result set.
- **XPath Queries**:
    - SQL Server 2000 and later versions support XPath queries, enabling efficient querying and navigation of XML data stored in SQL Server.

This XML support makes SQL Server a powerful tool for managing, manipulating, and integrating both XML and relational data.

Difference between Primary Key and Foreign Key

The **Primary Key** and **Foreign Key** are both used to enforce data integrity, but they serve different purposes in a relational database.

Primary Key

- **Uniqueness**: A primary key uniquely identifies each record in a table.
- **Null Values**: A primary key cannot accept null values.
- **Number of Keys**: Only one primary key is allowed per table.
- **Index**: By default, a primary key is associated with a clustered index. Data in the table is physically organized according to this index.
- **Auto Increment**: Primary keys often use an auto-increment feature to automatically generate unique values.
- **Relationship**: A primary key in one table can be referenced as a foreign key in another table to establish relationships.

Foreign Key

- **Relation to Primary Key**: A foreign key is a field in one table that refers to the primary key of another table.
- **Null Values**: Foreign keys can accept multiple null values, which means a record in the child table does not necessarily have to reference a parent record.
- **Number of Keys**: A table can have multiple foreign keys.
- **Index**: Foreign keys do not automatically create an index. Indexing must be manually created if needed.
- **Clustering**: Foreign keys do not have a clustered index by default.
- **Orphan Records**: If a foreign key is null, it may create "orphan records," which are records without a valid parent in the referenced table.

Unique Key

- **Uniqueness**: A unique key enforces the uniqueness of a column but allows for null values.
- **Null Values**: Unique keys can have null values.
- **Number of Keys**: A table can have more than one unique key.
- **Index**: A unique key is typically implemented as a unique non-clustered index.
- **Relationship**: Unique keys do not relate to other tables via foreign keys.
- **Auto Increment**: Unique keys do not support auto-increment like primary keys.

In summary, **Primary Key** is for uniquely identifying records and cannot have null values, while a **Foreign Key** establishes relationships between tables, allowing null values. **Unique Keys** are similar to primary keys but allow for multiple null values and can exist multiple times in a table.

SQL Queries

SQL Query to find second highest salary of Employee

There are multiple methods to find the second highest salary of an employee. Below are some common approaches using subqueries and joins.

Method 1: Using Subquery with COUNT

```
SELECT *
FROM employees emp1
WHERE 1 = (SELECT COUNT(DISTINCT(emp2.salary))
           FROM employees emp2
           WHERE emp2.salary > emp1.salary);
```

This query selects all records from `emp1` where the number of distinct salaries greater than `emp1.salary` is 1, effectively finding the second highest salary.

Method 2: Using `TOP` with `ORDER BY`

```
SELECT TOP 2 salary
FROM employees emp
ORDER BY salary DESC;
```

This query selects the top 2 salaries ordered in descending order. The second highest will be in the second row of the result.

Method 3: Using `MAX()` with a `NOT IN` Subquery

```
SELECT MAX(Salary)
FROM Employee
WHERE Salary NOT IN (SELECT MAX(Salary) FROM Employee);
```

This query finds the maximum salary that is not the highest (i.e., the second highest salary).

SQL Query to find Max Salary from each department

To find the maximum salary for each department, use the `MAX()` function with `GROUP BY` on `DeptID`.

```
SELECT DeptID, MAX(Salary)
FROM Employee
GROUP BY DeptID;
```

This will return the maximum salary for each department by grouping records based on `DeptID`.

With Department Name (Including departments with no employees)

```
SELECT DeptName, MAX(Salary)
FROM Employee e
RIGHT JOIN Department d ON e.DeptId = d.DeptID
GROUP BY DeptName;
```

This query uses a `RIGHT OUTER JOIN` to include departments without employees. It joins the `Employee` table with the `Department` table and returns the department name and the maximum salary for each department.

Write SQL Query to display the current date

SQL has a built-in function called `GETDATE()` that returns the current date and time in Microsoft SQL Server. Other databases have similar functions (e.g., `SYSDATE` in Oracle, `CURRENT_DATE` in MySQL).

```
SELECT GETDATE();
```

This query returns the current timestamp from the SQL Server.

Write an SQL Query to check whether date passed to Query is the date of given format or not.

SQL has `IsDate()` function which is used to check if the passed value is a date or not, based on the specified format. It returns `1` (true) if the value is a valid date and `0` (false) otherwise. This function works in SQL Server (MSSQL) and may not be available in other databases like Oracle or MySQL, but similar functions may exist in those systems.

```
SELECT ISDATE('1/08/13') AS "MM/DD/YY";
```

This will return `0` because the date passed is not in the correct format for SQL Server.

Write an SQL Query to print the name of the distinct employee whose DOB is between 01/01/1960 to 31/12/1975.

This SQL query uses the `BETWEEN` clause to select records whose `DOB` (Date of Birth) is between the two specified dates.

```
SELECT DISTINCT EmpName
FROM Employees
WHERE DOB BETWEEN '01/01/1960' AND '31/12/1975';
```

This will return the distinct names of employees whose date of birth falls between 01/01/1960 and 31/12/1975.

Write an SQL Query to find the number of employees according to gender whose DOB is between 01/01/1960 to 31/12/1975.

This query uses `COUNT()` and `GROUP BY` to get the number of employees grouped by gender, whose date of birth falls between the given range.

```
SELECT COUNT(*), sex
FROM Employees
WHERE DOB BETWEEN '01/01/1960' AND '31/12/1975'
GROUP BY sex;
```

This will return the count of employees for each gender who were born between 01/01/1960 and 31/12/1975.

Write an SQL Query to find an employee whose Salary is equal or greater than 10000

To find employees with a salary equal to or greater than 10000:

```
SELECT EmpName FROM Employees WHERE Salary >= 10000;
```

Write an SQL Query to find the name of employee whose name starts with ‘M’

You can use the `LIKE` operator to find employees whose name starts with 'M':

```
SELECT * FROM Employees WHERE EmpName LIKE 'M%';
```

Find all Employee records containing the word "Joe", regardless of whether it was stored as JOE, Joe, or joe.

You can use the `UPPER()` function to make the search case-insensitive and find all records containing "Joe":

```
SELECT * FROM Employees WHERE UPPER(EmpName) LIKE '%JOE%';
```

Write an SQL Query to find the year from date.

To extract the year from the current date in SQL Server, use the `YEAR()` function:

```
SELECT YEAR(GETDATE()) AS "Year";
```

How can you create an empty table from an existing table?

To create an empty table with the same structure as an existing table, you can use the following query:

```
SELECT * INTO studentcopy FROM student WHERE 1=2;
```

This creates the `studentcopy` table with the same structure as `student`, but without copying any rows.

How to fetch common records from two tables?

To fetch common records (intersection) between two tables, use the `INTERSECT` operator:

```
SELECT studentID FROM student
INTERSECT
SELECT studentID FROM Exam;
```

How to fetch alternate records from a table?

Step 1: Fetching Even Row Numbers

To fetch records with even row numbers, use the following query:

```
SELECT studentId
FROM (SELECT rowno, studentId FROM student)
WHERE MOD(rowno, 2) = 0;
```

Step 2: Fetching Odd Row Numbers

To fetch records with odd row numbers, use the following query:

```
SELECT studentId
FROM (SELECT rowno, studentId FROM student)
WHERE MOD(rowno, 2) = 1;
```

How to select unique records from a table?

Step 1: Using DISTINCT Keyword

The `DISTINCT` keyword is used to return only distinct (unique) values from a table.

```
SELECT DISTINCT StudentID, StudentName
FROM Student;
```

What is the command used to fetch first 5 characters of the string?

Step 1: Using SUBSTRING Function

The `SUBSTRING` function can be used to extract a portion of a string starting from a specified position. To fetch the first 5 characters:

```
SELECT SUBSTRING(StudentName, 1, 5) AS studentname
FROM student;
```

Step 2: Using RIGHT Function

The `RIGHT` function can be used to fetch the last 5 characters of a string. To fetch the first 5 characters, it will depend on string length, so this example fetches the

rightmost 5 characters:

```
SELECT RIGHT(StudentName, 5) AS studentname
FROM student;
```

Which operator is used in query for pattern matching?

Step 1: LIKE Operator for Pattern Matching

The `LIKE` operator is used in SQL to search for a specified pattern in a column.

- **%**: Matches zero or more characters.
- **_ (Underscore)**: Matches exactly one character.

Step 2: Example Queries

1. To find students whose name starts with 'a':

```
SELECT *
FROM Student
WHERE studentname LIKE 'a%';
```

2. To find students whose name starts with 'ami' followed by any one character:

```
SELECT *
FROM Student
WHERE studentname LIKE 'ami_';
```

Write SQL Query to find duplicate rows in a database? and then write SQL query to delete them?

Step 1: Find Duplicate Rows

```
SELECT *
FROM emp a
WHERE rowid = (SELECT MAX(rowid)
               FROM EMP b
               WHERE a.empno = b.empno);
```

Step 2: Delete Duplicate Rows

```
DELETE FROM emp a
WHERE rowid != (SELECT MAX(rowid)
                FROM emp b
                WHERE a.empno = b.empno);
```

There is a table which contains two column Student and Marks, you need to find all the students, whose marks are greater than average marks i.e. list of above average students.

Step 1: Find Students with Marks Greater Than Average

```
SELECT student, marks
FROM table
WHERE marks > (SELECT AVG(marks) FROM table);
```

Finding Employees Who Are Also Managers

To find employees who are also managers, you can perform a **self-join** on the **Employee** table. Here's how you can do it:

Step 1: Using Self-Join to Find Employees and Their Managers

In the self-join, you treat the **Employee** table as if it's two separate tables: one for employees (`e`) and one for managers (`m`). You join the tables using the `mgr_id` column (which references the `emp_id` of the manager).

```
SELECT e.name AS employee_name, m.name AS manager_name
FROM Employee e
JOIN Employee m ON e.mgr_id = m.emp_id;
```

This query will return the **employee name** along with their **manager name**.

Step 2: Include Employees Without Managers (Using LEFT JOIN)

If you also want to include employees who don't have a manager (i.e., their `mgr_id` is `NULL`), you can use a **LEFT JOIN** instead of an **INNER JOIN**. This will return all employees, including those without a manager.

```
SELECT e.name AS employee_name, m.name AS manager_name
FROM Employee e
LEFT JOIN Employee m ON e.mgr_id = m.emp_id;
```

Explanation:

- **INNER JOIN**: Only returns employees who have a manager (i.e., `mgr_id` is not `NULL`).
- **LEFT JOIN**: Returns all employees, including those without a manager. For employees without a manager, the `manager_name` will be `NULL` .

Conclusion:

- The first query (`INNER JOIN`) returns only employees who have managers.
- The second query (`LEFT JOIN`) returns all employees, including those without a manager, with `NULL` for the manager name.

Will Index be Used for Query with Two Columns in a Composite Index?

When you have a **composite index** on multiple columns, like `EmpId` , `EmpFirstName` , and `EmpSecondName` , and you write a `SELECT` query using only two of those columns in the `WHERE` clause (for example, `EmpId` and `EmpFirstName`), **the index can still be used** under certain conditions.

- **Primary Columns of the Index**: In a composite index, the order of the columns matters. If the columns used in the `WHERE` clause are part of the index and come **before** any columns not used, then the index can be utilized. The index is most efficient when the query uses the leftmost columns of the composite index.

  For example:

  ```
  SELECT *
  FROM Employee
  WHERE EmpId = 2 AND EmpFirstName = 'Radhe';
  ```

  In this case, since `EmpId` is the **first column** of the composite index and `EmpFirstName` is the **second column**, the index can be used to quickly locate the records.

- **Secondary Columns of the Index**: If you query using columns that are not the leading columns of the composite index, SQL Server might not be able to fully utilize the index for performance, as the index is built to be used in the order of the columns defined.

  For instance, if this query only includes `EmpSecondName` , SQL Server may not use the index optimally unless the query includes all preceding columns (like `EmpId` and `EmpFirstName`).

In this example, since `EmpId` and `EmpFirstName` are part of the **leading columns** of the composite index, the index **will be used**.

What is the Default Join in SQL?

The **default join** in SQL is the **INNER JOIN**. This type of join returns only the rows that have matching values in both tables. If there is no match between the tables, those rows will not appear in the result.

Example Query:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2 ON table1.column_name = table2.column_name;
```

This query will retrieve the rows where there is a match between `table1.column_name` and `table2.column_name` . If there is no match in one of the tables, that row will be excluded from the result.

Describe All the Joins with Examples in SQL

SQL Joins are used to combine rows from two or more tables based on a related column between them. Below are the different types of joins:

SQL LEFT JOIN

The **LEFT JOIN** (also called **LEFT OUTER JOIN**) returns all rows from the **left table** (`table1`), along with the matching rows from the **right table** (`table2`). If there is no match, the result from the right table will contain `NULL` values.

Syntax:

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2 ON table1.column_name = table2.column_name;
```

Example:

```
SELECT Employees.EmployeeID, Employees.EmployeeName, Departments.DepartmentName
FROM Employees
LEFT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

This query will return all employees, along with their department name. If an employee is not assigned to a department, the `DepartmentName` will be `NULL` .

SQL RIGHT JOIN

The **RIGHT JOIN** (also called **RIGHT OUTER JOIN**) is the opposite of the **LEFT JOIN**. It returns all rows from the **right table** (`table2`), with the matching rows from the **left table** (`table1`). If there is no match, the result from the left table will contain `NULL` values.

Syntax:

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2 ON table1.column_name = table2.column_name;
```

Example:

```
SELECT Employees.EmployeeID, Employees.EmployeeName, Departments.DepartmentName
FROM Employees
RIGHT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

This query will return all departments, along with employee names. If a department has no employees, the `EmployeeName` will be `NULL` .

SQL FULL OUTER JOIN

The **FULL OUTER JOIN** returns all rows from both the **left table** (`table1`) and the **right table** (`table2`). If there is no match between the tables, the result will contain `NULL` values in the columns from the table without a match.

Syntax:

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2 ON table1.column_name = table2.column_name;
```

Example:

```
SELECT Employees.EmployeeID, Employees.EmployeeName, Departments.DepartmentName
FROM Employees
FULL OUTER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

This query will return all employees and all departments. If an employee is not assigned to a department, the `DepartmentName` will be `NULL` . Similarly, if a department has no employees, the `EmployeeName` will be `NULL` .

Summary of Joins:

1. **LEFT JOIN**: Returns all rows from the left table and matching rows from the right table; if no match, returns `NULL` for the right table's columns.
2. **RIGHT JOIN**: Returns all rows from the right table and matching rows from the left table; if no match, returns `NULL` for the left table's columns.
3. **FULL OUTER JOIN**: Returns all rows from both tables; where there is no match, returns `NULL` for columns from the table without a match.

What is Union and Union All? Explain the Differences

SQL UNION

The **UNION** operator is used to combine the result sets of two or more `SELECT` statements into a single result. The key characteristics of **UNION** are:

- **Distinct Results**: By default, the `UNION` operator eliminates duplicate values from the combined result.
- **Same Number of Columns**: Each `SELECT` statement must have the same number of columns.
- **Similar Data Types**: The corresponding columns in each `SELECT` statement must have compatible data types.
- **Same Column Order**: The columns must appear in the same order in each `SELECT` statement.

Example:

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

This will return a result set with distinct values from both `table1` and `table2`.

SQL UNION ALL

The **UNION ALL** operator is similar to **UNION**, but there is one key difference:

- **Allows Duplicate Values**: Unlike **UNION**, **UNION ALL** includes duplicate values in the result set.
- **Performance**: **UNION ALL** is generally faster than **UNION** because it does not perform the operation of eliminating duplicates.

Example:

```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

This will return all rows, including duplicates, from both `table1` and `table2`.

Differences Between UNION and UNION ALL:

1. **Duplicate Rows**:
    - **UNION** removes duplicate rows.
    - **UNION ALL** keeps duplicate rows.
2. **Performance**:
    - **UNION** can be slower because it checks for and removes duplicates.
    - **UNION ALL** is faster since it does not perform duplicate removal.
3. **Use Cases**:
    - Use **UNION** when you need only distinct rows.
    - Use **UNION ALL** when you need all rows, including duplicates, or when performance is a priority and duplicates don't matter.