**Recovery Methods Used by DBAs**

**1. Transaction Log-Based Recovery**

All DML operations are first recorded in the **Transaction Log** (.ldf file in SQL Server). This allows the database to:

* Rollback incomplete transactions
* Recover to a previous state using **point-in-time restore**

**Example (SQL Server):**

RESTORE DATABASE MyDB

FROM DISK = 'D:\Backup\MyDB.bak'

WITH STOPAT = '2025-07-17 14:30:00'

Requires that the database uses the **Full Recovery Model** and that backups are available.

**2. Point-in-Time Recovery**

This technique involves:

1. Restoring a full backup
2. Applying the transaction log backups up to a specific point before the unwanted DML was executed

This ensures minimal data loss and full recovery.

**3. Database Snapshots (SQL Server Enterprise)**

Snapshots are read-only static views of a database. If a problem occurs, data can be recovered directly from the snapshot.

**Recovery Example:**

INSERT INTO Employees

SELECT \* FROM EmployeeSnapshot.dbo.Employees

**4. Auditing Triggers**

Creating triggers that capture deleted or updated records into backup tables before the operation is finalized.

**Example:**

CREATE TRIGGER trg\_BackupBeforeDelete

ON Employees

FOR DELETE

AS

BEGIN

INSERT INTO DeletedEmployeesBackup

SELECT \* FROM DELETED

END

This approach enables manual recovery from backup tables.

**5. Temporal Tables (SQL Server 2016+)**

Temporal tables automatically store **historical versions** of data and allow queries over time.

**Example:**

SELECT \* FROM Employees

FOR SYSTEM\_TIME AS OF '2025-07-17 14:00:00'

**Dynamic Query and EXEC Keyword**

A **dynamic query** is an SQL statement that is **built as a string at runtime**, instead of being hard-coded.

It allows flexibility to modify table names, column names, conditions, or even entire statements based on input or logic

SELECT \* FROM Employees WHERE Dept\_Id = 10 /// static

DECLARE @SQL NVARCHAR(MAX) ///dynamic

DECLARE @Dept INT = 10

SET @SQL = 'SELECT \* FROM Employees WHERE Dept\_Id = ' + CAST(@Dept AS NVARCHAR)

EXEC(@SQL)

**Can execute dynamic SQL using:**

**1. EXEC or EXECUTE**

Used to run the query directly from a string

DECLARE @SQL NVARCHAR(MAX)

SET @SQL = 'SELECT \* FROM Employees WHERE Dept\_Id = 10'

EXEC(@SQL)

**2. sp\_executesql (Safer & Parameterized)**

Allows parameterized queries to **prevent SQL Injection**

DECLARE @SQL NVARCHAR(MAX)

DECLARE @Dept INT = 10

SET @SQL = N'SELECT \* FROM Employees WHERE Dept\_Id = @id'

EXEC sp\_executesql @SQL, N'@id INT', @id = @Dept

----------------------------------------------------------------------------------------------------------------------

**case - iif - wait for - choose**

**1. CASE Statement**

The CASE statement works like an IF...ELSE structure. It returns a value based on conditions.

**Syntax:**

SELECT

FirstName,

Salary,

CASE

WHEN Salary >= 5000 THEN 'High'

WHEN Salary >= 3000 THEN 'Medium'

ELSE 'Low'

END AS SalaryLevel

FROM Employees;

**Use Case:**

Used in SELECT, UPDATE, ORDER BY, etc., to apply conditional logic.

**2. IIF() Function**

Introduced in SQL Server 2012, IIF() is a shorthand for simple IF...ELSE.

**Syntax:**

SELECT FirstName, IIF(Salary > 4000, 'Good', 'Needs Raise') AS Status

FROM Employees;

**Use Case:**

Used when you have **simple binary conditions** (true/false). Easier than CASE for quick checks.

**3. WAITFOR Statement**

WAITFOR is used to **pause query execution** for a specific time.

**Syntax:**

-- Wait for 5 seconds

WAITFOR DELAY '00:00:05';

**Use Case:**

Used in testing, simulating delays, retry mechanisms, or scheduled logic.

You can also use it with a specific time:

WAITFOR TIME '14:30:00'; -- Waits until 2:30 PM

**4. CHOOSE() Function**

CHOOSE() returns a value from a list based on an index (1-based).

**Syntax:**

SELECT CHOOSE(2, 'Red', 'Green', 'Blue') AS Color;

-- Output: Green

You can also use it in a query:

SELECT Name, CHOOSE(Level, 'Low', 'Medium', 'High') AS LevelName

FROM Employees;

**Use Case:**

Useful when mapping numbers to descriptive names or labels.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **TRY...CATCH Approach** | **IF NOT EXISTS Approach** |
| **Error Logging** | Logged as an actual **error** in the log | Not logged as error (handled logically) |
| **Control** | You react **after** the error happens | You **prevent** the error from happening |
| **Performance** | Slightly **slower** if error occurs (rollback involved) | Usually **faster**, avoids exception processing |
| **Clarity** | May seem simpler for short code | More explicit logic, easier to debug |
| **Recommended Use** | Useful for **unexpected errors** | Better for **expected constraints**, like foreign keys |

**what's between two approaches logging , performance level**

Use **IF NOT EXISTS** when you **expect** the constraint and want to handle it cleanly.

Use **TRY...CATCH** for **unexpected runtime errors** (e.g., timeout, permission issues, unknown bugs)

------------------------------------------------------------------------------------------------------------

**What is trigger and why use it**

A **trigger** is a **special type of stored procedure** that **automatically executes** (or "fires") when certain **events** happen on a table or view.

You **don’t call a trigger manually** — the database runs it automatically when an INSERT, UPDATE, or DELETE occurs.

**Why Use Triggers?**

**1. Automatic Enforcement of Business Rules**

* Example: Prevent deleting a manager if they still supervise employees.

**2. Auditing and Logging**

* Track changes made to important tables.
* Example: Log who changed a salary and when.

**3. Cascading Actions**

* Automatically update or delete related data.
* Example: If a department is deleted, remove related logs or archive them.

**4. Data Validation**

* Prevent invalid data from being inserted or updated.

**What is permissions and how w grant and revoke it**

**Permissions** control **what actions a user can perform** on a database object like a table, view, or procedure.

They are like **access rights** — without permissions, a user can’t read, insert, update, or delete data.

**How to GRANT Permissions**

Use the GRANT statement to **give** a user permission.

**Syntax:**

GRANT permission\_name ON object\_name TO user\_name

**Example:**

GRANT SELECT, INSERT ON Employees TO User1

➡ This gives **User1** the right to **read and insert** into the Employees table.

**How to REVOKE Permissions**

Use the REVOKE statement to **remove** a previously granted permission.

**Syntax:**

REVOKE permission\_name ON object\_name FROM user\_name

**Example:**

REVOKE INSERT ON Employees FROM User1

➡ Now **User1** can no longer insert into the Employees table, but can still select if that permission wasn't revoked.

|  |  |  |
| --- | --- | --- |
| **Command** | **Purpose** | **Example** |
| GRANT | Give permission | GRANT SELECT ON Employees TO User1; |
| REVOKE | Remove permission | REVOKE SELECT ON Employees FROM User1; |