Self Study

1. Common Recovery Methods

SQL database recovery depends on the issue (e.g., data loss, corruption, or crash). Below is a simple and concise guide to common recovery methods in SQL Server:

Backup Restoration:

- Identify the latest full or differential backup.
- Use RESTORE DATABASE command to restore the backup.
- Verify restored data with DBCC CHECKDB.

Transaction Log Recovery:

- Restore transaction log backups using RESTORE LOG.
- Apply logs to recover changes since the last backup.
- Ensure the log chain is unbroken (no gaps).

Point-in-Time Recovery (PITR):

- Restore to a specific time using RESTORE DATABASE with STOPAT.
- Requires full recovery model and transaction log backups.
- Validate the restored database state.

Database Repair Tools:

- Run DBCC CHECKDB to detect and repair corruption.
- Use REPAIR_REBUILD or REPAIR_ALLOW_DATA_LOSS for fixes (data loss possible).
- Back up the database before repairs.

Crash Recovery:

- SQL Server automatically recovers on restart using ARIES.
- Check error logs for recovery issues.
- Ensure the database is in a consistent state post-recovery.

File-Level Recovery:

- Restore corrupted data or log files from backups.
- Use RESTORE DATABASE with REPLACE to overwrite damaged files.
- Restart SQL Server and verify database access.

High Availability (HA) Failover:

- Fail over to a secondary replica in Always On Availability Groups.
- Promote the secondary using ALTER AVAILABILITY GROUP.
- Rebuild the failed instance after recovery.

2. What is the dynamic query ?

In SQL Server, a dynamic query is a SQL statement that is constructed and executed at runtime, typically using string concatenation or parameterization. Unlike static queries, which are hardcoded and fixed at design time, dynamic queries are built dynamically based on variables, user input, or conditions, allowing for flexible query execution.

A SQL query whose text is generated programmatically during execution, often using variables or logic to construct the query string.

The query is built as a string and executed using commands like EXEC or sp_executesql.

Used to create flexible queries that adapt to runtime conditions, such as variable table names, column names, or user-defined criteria.

Examples of Dynamic Queries

Basic Dynamic Query with EXEC:

```
DECLARE @TableName NVARCHAR(128) = 'Employees';
DECLARE @SQL NVARCHAR(200) = 'SELECT * FROM ' + @TableName;
EXEC (@SQL);
```

Constructs a query to select all columns from a dynamically specified table.
Output: Retrieves all records from the Employees table.

Dynamic Query with sp_executesql (Parameterized):

```
DECLARE @SQL NVARCHAR(200);
DECLARE @ColumnName NVARCHAR(100) = 'Name';
DECLARE @TableName NVARCHAR(128) = 'Employees';
SET @SQL = N'SELECT ' + QUOTENAME(@ColumnName) + ' FROM ' +
QUOTENAME(@TableName);

EXEC sp_executesql @SQL;
```

Uses sp_executesql for better performance and parameterization support.
QUOTENAME prevents SQL injection by escaping special characters.

Dynamic Query with Parameters:

```
DECLARE @SQL NVARCHAR(200);
DECLARE @Value INT = 100;
SET @SQL = N'SELECT * FROM Employees WHERE Salary > @Salary';

EXEC sp_executesql @SQL, N'@Salary INT', @Salary = @Value;
```

Uses parameterized queries to safely pass values, reducing SQL injection risks.

When to Use Dynamic Queries ?

Use Cases:

- Building queries where table/column names are not known until runtime (e.g., dynamic schema exploration).
- Creating flexible search queries based on user input (e.g., optional filters).
- Automating repetitive tasks, like maintenance scripts across multiple tables.

Avoid When:

- Static queries can achieve the same result with less complexity.
- User input is not properly sanitized, increasing security risks.
- Performance is critical, and query plan caching is a priority.

3. What is a trigger and why use it ?

A trigger is a special type of stored procedure that automatically executes in response to specific events on a table, such as INSERT, UPDATE, or DELETE operations. Triggers are used to enforce business rules, maintain data integrity, or automate tasks without requiring explicit user intervention.

Types:

DML Triggers: Fire on Data Manipulation Language events (INSERT, UPDATE, DELETE).

AFTER (FOR): Executes after the event (e.g., after a row is inserted).

INSTEAD OF: Executes instead of the event, overriding the default action.

DDL Triggers: Fire on Data Definition Language events (e.g., CREATE, ALTER, DROP).

Logon Triggers: Fire on user logon events.

Example (DML Trigger):

```
CREATE TRIGGER trg_AfterInsert_Department
ON Department
AFTER INSERT
AS
BEGIN
   INSERT INTO AuditLog (Action, Details)
   SELECT 'Insert', 'New Dept_Id: ' + CAST(Dept_Id AS NVARCHAR(10))
```

```
    FROM inserted;
END;
```

Logs new department insertions into an AuditLog table.

Key Considerations

Triggers add overhead, as they execute automatically for every affected row, potentially slowing DML operations. Avoid complex logic or recursive triggers to minimize performance impact.

Triggers generate additional transaction log entries for their operations (e.g., INSERT into an audit table).

Consider constraints, stored procedures, or application logic for simpler tasks to avoid overusing triggers.

Example Use Case

Scenario: Log all updates to the Department table and prevent updates if a condition is not met.

```
CREATE TRIGGER trg_AfterUpdate_Department
ON Department
AFTER UPDATE
AS
BEGIN

    -- Prevent updates if a condition fails
    IF EXISTS (SELECT 1 FROM inserted WHERE Dept_Name = '')
    BEGIN

        RAISERROR ('Department name cannot be empty', 16, 1);
        ROLLBACK;

    END
```

```
    ELSE
    BEGIN

        -- Log the update

        INSERT INTO AuditLog (Action, Details)
        SELECT 'Update', 'Dept_Id: ' + CAST(Dept_Id AS NVARCHAR(10)) +
' updated'
        FROM inserted;

    END

END;
```

4. What are permissions and how do we grant and revoke them ?

In SQL Server, permissions define what actions a user, role, or application can perform on database objects (e.g., tables, views, stored procedures) or the database itself. Permissions are part of SQL Server's security model, controlling access and operations to ensure data security and integrity.

Permissions are rules that specify what actions (e.g., SELECT, INSERT, UPDATE, DELETE, EXECUTE) a principal (user, role, or group) can perform on a securable (database, schema, table, etc.).

Types of Permissions:

Object-Level: Apply to specific objects like tables, views, or stored procedures (e.g., SELECT, INSERT, EXECUTE).

Schema-Level: Apply to all objects within a schema (e.g., ALTER on a schema).

Database-Level: Apply to database-wide operations (e.g., CREATE TABLE, BACKUP DATABASE).

Server-Level: Apply to server-wide operations (e.g., CREATE DATABASE, SHUTDOWN).

Key Principals:

Users: Individual accounts in a database.

Roles: Groups of permissions that can be assigned to users (e.g., db_datareader, db_owner).

Logins: Server-level accounts that map to database users.

Purpose: Ensure only authorized users or roles can access or modify data, maintaining security and compliance.

How to Grant Permissions

Command: Use the GRANT statement to assign permissions to a principal.

Syntax:

```
GRANT <permission> [ON <securable>] TO <principal> [WITH GRANT OPTION];
```

Where:

<permission>: Action (e.g., SELECT, INSERT, EXECUTE, ALTER).

<securable>: Object (e.g., TABLE::Department, SCHEMA::dbo).

<principal>: User, role, or login (e.g., UserName, db_datareader).

WITH GRANT OPTION: Allows the principal to grant the same permission to others.

Examples:

Grant SELECT on a Table: Allows UserName to read data from the Department table.

```
GRANT SELECT ON Department TO UserName;
```

Grant EXECUTE on a Stored Procedure: Allows UserName to execute the MyProcedure stored procedure.

```
GRANT EXECUTE ON dbo.MyProcedure TO UserName;
```

Grant Schema-Level Permissions: Grants SELECT and INSERT on all objects in the dbo schema to RoleName.

```
GRANT SELECT, INSERT ON SCHEMA::dbo TO RoleName;
```

Grant Database-Level Permission: Allows UserName to create tables in the database.

```
GRANT CREATE TABLE TO UserName;
```

Grant with GRANT OPTION: Allows UserName to grant SELECT permission to others.

```
GRANT SELECT ON Department TO UserName WITH GRANT OPTION;
```

How to Revoke Permissions

Command: Use the REVOKE statement to remove previously granted permissions from a principal.

Syntax:

```
REVOKE [GRANT OPTION FOR] <permission> [ON <securable>] FROM
<principal> [CASCADE];
```

GRANT OPTION FOR: Removes the ability to grant the permission to others without revoking the permission itself.

CASCADE: Also revokes permissions granted by the principal to others (if they used WITH GRANT OPTION).

Examples:

Revoke SELECT on a Table: Removes UserName's ability to read from the Department table.

```
REVOKE SELECT ON Department FROM UserName;
```

Revoke EXECUTE on a Stored Procedure: Removes UserName's ability to execute MyProcedure.

```
REVOKE EXECUTE ON dbo.MyProcedure FROM UserName;
```

Revoke Schema-Level Permissions: Removes SELECT permission on all objects in the dbo schema for RoleName.

```
REVOKE SELECT ON SCHEMA::dbo FROM RoleName;
```

Revoke GRANT OPTION: Removes UserName's ability to grant SELECT to others but keeps their SELECT permission.

```
REVOKE GRANT OPTION FOR SELECT ON Department FROM UserName;
```

Revoke with *CASCADE*: Removes SELECT permission from UserName and any principals they granted it to.

```
REVOKE SELECT ON Department FROM UserName CASCADE;
```