Question Answering Assignment

1. What's the difference between full, differential and transactional back up ?

1. Full Backup

Captures the entire database at a specific point in time, including all data, objects, and schema (tables, indexes, stored procedures, etc.).

Contains all data in the database, regardless of previous backups.

Serves as the baseline for other types of backups (differential and transaction log).

Restoring a full backup restores the database to the exact state it was in when the backup was taken.

Use Case: Initial backup or periodic complete snapshots of the database.

Pros:

- Comprehensive and standalone; no other backups are needed to restore.
- Simplest restore process.

Cons:

- Takes the most time and storage space compared to other backup types.
- Can impact performance if taken frequently on large databases.

2. Differential Backup

Captures only the data that has changed since the last full backup.

Smaller and faster than a full backup because it only includes changes (new or modified data).

Requires the last full backup to restore the database (you restore the full backup first, then apply the differential).

Each differential backup grows larger over time as more data changes since the last full backup.

Use Case: Used to reduce backup time and storage needs between full backups.

Pros:

- Faster to create and restore than a full backup.
- Uses less storage than a full backup.

Cons:

- Requires the corresponding full backup for restoration.
- Cumulative changes can make later differential backups larger.

3. Transaction Log Backup

Backs up the transaction log, which records all database transactions (inserts, updates, deletes) since the last transaction log backup or full backup.

Only available in the Full or Bulk-Logged recovery models (not Simple recovery model).

Allows point-in-time recovery, meaning you can restore the database to a specific moment (e.g., just before a failure or error).

Requires a full backup and any subsequent transaction log backups to restore.

Clears the transaction log, preventing it from growing too large.

Use Case: Used for frequent backups (e.g., every 15 minutes) to minimize data loss in case of failure and to enable precise recovery.

Pros:

- Enables point-in-time recovery.
- Small and quick to back up, as it only captures transaction log entries.
- Helps manage transaction log size.

Cons:

- Requires a sequence of transaction log backups and the last full backup (and any differential backups) for restoration.
- More complex restore process, as multiple files may need to be applied in order.

2. What is permission and What's the difference between grant and deny and used on what level ?

What is a Permission?

A permission is a rule that grants or denies a specific action (e.g., SELECT, INSERT, EXECUTE) on a database object to a user, group, or role.

Scope: Permissions can be applied at various levels, such as:

- **Server level:** Controls access to server-wide operations (e.g., CREATE DATABASE, SHUTDOWN).
- **Database level:** Controls access to database-specific operations (e.g., CREATE TABLE, BACKUP DATABASE).
- **Schema level:** Controls access to objects within a schema (e.g., tables, views).
- **Object level:** Controls access to specific objects (e.g., a single table or stored procedure).

Managed by: Permissions are managed using T-SQL statements like GRANT, DENY, and REVOKE.

Difference Between GRANT and DENY ?

GRANT:

Explicitly allows a user or role to perform a specific action on an object or resource.

Enables the specified permission. For example, GRANT SELECT allows a user to read data from a table.

Usage Example:

```
GRANT SELECT ON dbo.MyTable TO User1;
```

This allows User1 to query dbo.MyTable.

If a user has not been granted a permission, they cannot perform that action unless they inherit it via a role or higher-level permission.

DENY:

Explicitly prohibits a user or role from performing a specific action, even if they would otherwise have access (e.g., through a role or inherited permission).

Overrides any GRANT at the same or lower level. For example, if a user is granted SELECT through a role but denied SELECT explicitly, they cannot perform the action.

Usage Example:

```
DENY SELECT ON dbo.MyTable TO User1;
```

This prevents User1 from querying dbo.MyTable, even if they belong to a role with SELECT permission.

DENY takes precedence over GRANT, making it a powerful way to restrict access.

3. What's sql profiler and when to use it ?

What is SQL Server Profiler?

A graphical tool included with Microsoft SQL Server that allows database administrators and developers to capture and analyze events occurring in a SQL Server instance. It is used for monitoring and troubleshooting database activity by tracing operations such as SQL queries, stored procedures, transactions, and other server events. SQL Server Profiler provides detailed insights into the performance, behavior, and interactions within a SQL Server database.

It has several features. It captures real-time events (e.g., SQL statements, stored procedure executions, logins, errors). Displays events in a trace, which can be saved for later analysis or replayed. Allows filtering to focus on specific databases, users, applications, or events. Provides performance metrics, such as query duration, CPU usage, and I/O operations.

SQL Server Profiler is available in SQL Server Management Studio (SSMS) under the "Tools" menu.

When to Use SQL Server Profiler?

SQL Server Profiler is useful in various scenarios, particularly for troubleshooting, performance tuning, and auditing. Use Cases:

1. **Performance Troubleshooting**: Identify slow-running queries or stored procedures by analyzing execution times, CPU usage, and I/O metrics. Detect performance bottlenecks, such as excessive locking, blocking, or resource-intensive queries.
2. **Debugging Application Issues**: Capture the exact SQL statements or stored procedure calls sent by an application to diagnose errors or unexpected behavior. Identify parameterized queries or missing parameters causing failures.

3. **Auditing and Security Monitoring**: Track user activity, such as logins, logouts, or permission changes, to ensure security compliance. Monitor specific database actions (e.g., INSERT, UPDATE, DELETE) to detect unauthorized access or changes.
4. **Analyzing Workload Patterns**: Understand database usage patterns, such as peak query times or frequently executed statements, to optimize resource allocation. Replay captured traces on a test server to simulate production workloads for testing or tuning.
5. **Deadlock and Locking Analysis**: Capture deadlock events to identify conflicting queries and resolve contention issues. Analyze locking behavior to optimize transaction isolation levels or query design.
6. **Baseline Performance Monitoring**: Create a baseline of normal database activity to compare against periods of poor performance.

4. What is the trigger and why use it and on what level and what makes it different from normal Stord procedure ?

What is the 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 selected 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;
```

What makes it different from normal Stord procedure ?

In SQL Server, a trigger is a special type of stored procedure that automatically runs in response to specific database events like INSERT, UPDATE, or DELETE on a table, or DDL events like CREATE. A stored procedure is a precompiled set of T-SQL statements executed manually via EXEC or by an application.

Key Differences:

Execution: Triggers run automatically when their associated event occurs; stored procedures are called explicitly.

Event-Driven: Triggers are tied to specific table or database events; stored procedures are not.

Parameters: Triggers use virtual inserted and deleted tables to access modified data, not parameters; stored procedures accept input/output parameters.

Return Values: Triggers don't return values to the caller; stored procedures can return data or result sets.

Transaction: Triggers operate within the triggering event's transaction and can rollback; stored procedures manage transactions explicitly.

Use Case: Triggers enforce rules, audit changes, or maintain data integrity automatically (e.g., log inserts). Stored procedures handle complex logic, reporting, or batch tasks on demand (e.g., generate reports).

When to Use:

Triggers: For automatic actions like auditing or enforcing rules (e.g., update a log table on INSERT).

Stored Procedures: For reusable, on-demand tasks like data processing or reporting.

Best Practices: Keep triggers lightweight to avoid performance issues; use parameters and error handling in stored procedures for flexibility and robustness.