|  |  |  |
| --- | --- | --- |
| **Feature** | **Trigger** | **Stored Procedure** |
| **Definition** | A trigger is a special type of stored procedure that automatically executes in response to certain events on a particular table or view. | A stored procedure is a compiled collection of one or more SQL statements that can be executed as a single unit to perform a specific task. |
| **Execution Timing** | Automatically executed (fired) when a specific event occurs (e.g., INSERT, UPDATE, DELETE). | Manually executed by a user or application when called. |
| **Invocation** | Cannot be called directly; it is invoked automatically by the database engine in response to data modification events. | Can be called explicitly using the EXEC command or from within other procedures or applications. |
| **Return Type** | Does not return a value or result set directly; used primarily for performing actions based on data changes. | Can return a value (using RETURN statement) and can also return multiple result sets. |
| **Parameters** | Cannot accept parameters. Triggers operate on the inserted and deleted tables, which hold the affected rows. | Can accept input parameters, allowing for more flexible and reusable code. |
| **Scope of Action** | Limited to the table/view on which it is defined; cannot perform operations on other tables unless explicitly coded. | Can perform complex operations, including multi-table queries, and can call other stored procedures or functions. |
| **Use Cases** | Typically used for enforcing business rules, maintaining data integrity, and auditing changes. | Used for encapsulating complex logic, batch processing, and reusable code modules. |
| **Performance Impact** | Can lead to performance issues if not carefully implemented, as they run automatically on data modifications. | Generally better for performance as they can be optimized and reused without the overhead of automatic execution. |
| **Error Handling** | Limited error handling capabilities; if an error occurs, the triggering action may fail without the ability to recover. | Can implement structured error handling using TRY...CATCH blocks, allowing for more robust error management. |
| **Debugging** | More challenging to debug due to automatic invocation; can use logging within the trigger to track execution. | Easier to debug since they can be executed in isolation and tested with various parameters. |
| **How to Write** | ```sql CREATE TRIGGER trigger\_name ON table\_name A F T E R | INSTEAD OF event AS BEGIN -- Trigger logic here END ``` |
| **How to Use** | Automatically executed when the defined event occurs on the specified table. | Called using EXEC procedure\_name or EXEC procedure\_name @param1, @param2. |

**procedures** and **functions** in SQL Server:

|  |  |  |
| --- | --- | --- |
| **Feature** | **Stored Procedure** | **Function** |
| **Definition** | A stored procedure is a compiled collection of one or more SQL statements that can be executed as a single unit to perform a specific task. | A function is a reusable piece of code that can accept parameters, perform an operation, and return a single value or a table. |
| **Return Type** | Can return multiple result sets and can return a value using the RETURN statement. | Must return a value (scalar) or a table (table-valued function) and uses the RETURN statement for a single return value. |
| **Invocation** | Executed using the EXEC command or called from another procedure or application. | Called as part of a SQL expression, typically within a SELECT, WHERE, or JOIN clause. |
| **Parameters** | Can accept input parameters and can also have output parameters to return values. | Can accept input parameters but cannot have output parameters. Must return a value or result set. |
| **Transaction Support** | Supports transaction control (BEGIN, COMMIT, ROLLBACK), allowing for complex operations. | Does not support transaction control; operations are generally atomic. |
| **Side Effects** | Can modify database state (e.g., perform INSERT, UPDATE, DELETE operations). | Should not have side effects; designed primarily to compute and return values without modifying database state. |
| **Usage Scenarios** | Used for encapsulating complex logic, batch processing, and performing administrative tasks. | Used for computations, encapsulating reusable logic, and returning values as part of larger queries. |
| **Error Handling** | Can implement structured error handling using TRY...CATCH blocks for robust error management. | Limited error handling; errors can cause the function to fail without structured handling. |
| **Performance** | Generally more efficient for complex operations that require multiple statements. | Can be less efficient in complex operations due to the restriction of returning only one value or table. |
| **Debugging** | Easier to debug since they can be executed in isolation with various parameters. | More challenging to debug due to limitations in error handling and invocation context. |

!

|  |  |  |
| --- | --- | --- |
| **Feature** | **Inline Function** | **View** |
| **Definition** | An inline function is a user-defined function that returns a single table result set and can be used in SQL statements like a regular table. | A view is a virtual table that is based on the result of a SELECT statement, allowing users to access data from one or more tables in a simplified manner. |
| **Return Type** | Returns a table result set. | Represents a result set from a SELECT statement but does not return a value directly. |
| **Parameters** | Can accept parameters, allowing for dynamic behavior based on input. | Cannot accept parameters; it is a static representation of the data. |
| **Usage** | Can be used in SELECT, JOIN, WHERE, and other SQL statements just like a table. | Can be queried like a table in SELECT statements but cannot be used in certain contexts (e.g., cannot be used in an INSERT or UPDATE statement directly). |
| **Execution** | Executes the underlying SELECT statement each time it is called. | Executes the underlying SELECT statement when the view is queried; the execution plan is cached. |
| **Performance** | Performance can vary depending on the complexity of the function; may not be optimized like views. | Generally performs well, especially if indexed views are used; the optimizer can cache execution plans. |
| **Use Cases** | Used for encapsulating reusable logic, particularly for complex calculations or filtering criteria. | Used to simplify complex queries, present data in a specific format, or provide security by limiting access to certain columns or rows. |
| **Updateability** | Inline functions can be used in data manipulation (INSERT, UPDATE, DELETE) if they return a single table. | Views can be updatable under certain conditions; if they are based on a single table without aggregations, they can be used for data manipulation. |
| **Security** | Can encapsulate complex logic and be used in permissions to restrict access to sensitive data. | Can provide a security layer by allowing users to access specific columns or rows without giving them direct access to the underlying tables. |
| **Debugging** | Debugging can be more challenging due to parameter handling and execution context. | Easier to debug since views can be queried directly and tested independently. |

|  |  |  |
| --- | --- | --- |
| **Feature** | **DROP** | **DELETE** |
| **Definition** | Removes an entire database object (table, view, index, etc.) from the database. | Removes rows from a table based on specified criteria. |
| **Scope** | Affects the entire object; cannot be rolled back unless in a transaction. | Affects specific rows; can be rolled back if wrapped in a transaction. |
| **Data Removal** | All data and the structure of the object are permanently removed. | Data is removed, but the structure of the table remains intact. |
| **Triggers** | Does not activate any triggers. | Activates DELETE triggers defined on the table. |
| **Use Case** | Used when you want to permanently remove a table or other object. | Used when you want to remove specific rows while keeping the table structure. |
| **Performance** | Generally faster since it removes the entire object and its data. | Slower than DROP for large datasets, as it logs each row deletion. |
| **Syntax Example** | DROP TABLE table\_name; | DELETE FROM table\_name WHERE condition; |

**SELECT vs. SELECT INTO**

|  |  |  |
| --- | --- | --- |
| **Feature** | **SELECT** | **SELECT INTO** |
| **Definition** | Retrieves data from one or more tables and displays it in the result set. | Creates a new table and populates it with data from an existing table based on the query results. |
| **Existing Tables** | Does not modify or create any tables. | Creates a new table if it does not already exist; can cause an error if the table exists. |
| **Use Case** | Used for querying data without altering the database structure. | Used to make a copy of data into a new table for backup, reporting, or transformation. |
| **Syntax Example** | SELECT \* FROM table\_name WHERE condition; | SELECT \* INTO new\_table\_name FROM table\_name WHERE condition; |
| **Performance** | Generally faster for simply retrieving data. | Performance may vary based on the size of the dataset being copied to a new table. |
| **Result Set** | Returns a result set that can be further processed or displayed. | Creates a new table and returns a confirmation of the operation rather than a result set. |

**5. DDL, DML, DCL, and DQL**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | **DDL (Data Definition Language)** | **DML (Data Manipulation Language)** | **DCL (Data Control Language)** | **DQL (Data Query Language)** |
| **Definition** | Deals with the structure (schema) of the database. | Deals with the manipulation of data in existing tables. | Deals with the permissions and access control. | Deals with querying data from the database. |
| **Commands** | CREATE, ALTER, DROP, TRUNCATE | INSERT, UPDATE, DELETE, MERGE | GRANT, REVOKE, DENY | SELECT |
| **Function** | Defines and modifies database schema and objects. | Manages data within schema objects. | Controls access to data and database operations. | Retrieves data from the database. |
| **Transactions** | Not transactional; changes are immediately applied. | Usually transactional; can be rolled back or committed. | Not directly transactional, but impacts data access. | Not transactional; primarily used for data retrieval. |
| **Use Cases** | Creating tables, altering table structures, deleting tables. | Inserting new records, updating existing records, deleting records. | Granting or revoking permissions on database objects. | Fetching data from one or multiple tables. |

**6. Table-Valued Functions vs. Multi-Statement Functions**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Table-Valued Functions** | **Multi-Statement Functions** |
| **Definition** | A function that returns a table data type. | A function that can perform multiple statements and return a single value. |
| **Return Type** | Returns a table as output. | Returns a single scalar value (e.g., INT, VARCHAR). |
| **Use Cases** | Used for returning rows from queries, similar to a table. | Used for complex logic where multiple operations are performed. |
| **Performance** | Generally faster since it returns a result set directly. | May be slower due to the execution of multiple statements. |
| **Parameters** | Can accept parameters to filter or modify the result set. | Can accept parameters to affect the logic within the function. |
| **Execution** | Can be used in FROM clause of SQL statements. | Cannot be used in the FROM clause, used in expressions. |

**7. VARCHAR(50) vs. VARCHAR(MAX)**

|  |  |  |
| --- | --- | --- |
| **Feature** | **VARCHAR(50)** | **VARCHAR(MAX)** |
| **Definition** | A variable-length string data type with a maximum length of 50 characters. | A variable-length string data type with a maximum length of 2^31-1 (approximately 2 billion characters). |
| **Storage** | Allocates space for a maximum of 50 characters; if fewer characters are used, only the required space is allocated. | Allocates space as needed, can store very large strings up to the defined limit. |
| **Performance** | Generally faster for small to medium-sized strings due to fixed upper limit. | May have slower performance when dealing with very large data sizes due to memory management. |
| **Use Cases** | Suitable for fields with a known maximum length (e.g., names, short descriptions). | Suitable for large text data, such as documents, descriptions, or logs. |
| **Usage Limitations** | Cannot store more than 50 characters; may lead to truncation. | Can handle very large strings without truncation, but needs to be managed carefully for performance. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Feature** | | **SQL Server Authentication** | | | | | | | **Windows Authentication** | | | | | | |
| **Definition** | | | | | A method of user authentication that requires a username and password defined within SQL Server. | | | | | | | A method that uses Windows credentials to authenticate users, leveraging the security of the Windows operating system. | | | | | |
| **User Management** | | | | | | | Users are managed within SQL Server; separate from the Windows user accounts. | | | | Users are managed by the Windows operating system and can utilize their existing Windows accounts for authentication. | | | | | | |
| **Security** | | | Less secure as it relies on passwords stored in SQL Server. | | | | | | | More secure due to Windows security policies, including password complexity and account lockout policies. | | | | | | | |
| **Usage** | Commonly used in environments where SQL Server is accessed remotely or when applications do not have access to Windows authentication. | | | | | | | | | | | | | | | | Preferred in organizations that utilize Active Directory and want to manage users centrally with Windows accounts. |
| **Connection String** | | | | | | Requires specifying username and password in the connection string. | | | | | | | Uses Windows credentials automatically; no need to provide a username or password. | | | | |
| **Roles and Permissions** | | | | | | | | Roles and permissions are defined in SQL Server and are independent of Windows security settings. | | | | | | | Roles and permissions can be tied to Windows group memberships, allowing for easier management of access. | | |
| **Scenario** | | | | Useful in mixed environments or applications that are not domain-joined. | | | | | | | | | | Ideal for intranet applications where users are part of a Windows domain. | | | |

**10. Identity vs. Unique Constraint**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Identity** | **Unique Constraint** |
| **Definition** | A property that automatically generates a unique value for a column, typically used for primary keys. | A constraint that ensures all values in a column or a combination of columns are unique across the table. |
| **Usage** | Commonly used for primary keys to ensure each record has a unique identifier without manual input. | Used when you want to enforce uniqueness on non-primary key columns, allowing for duplicate values in other columns. |
| **Value Generation** | Automatically increments (or decrements) the value for each new row added to the table. | Does not generate values; must be manually provided by the user during insertions. |
| **Data Type** | Typically used with integer or numeric data types. | Can be applied to any data type, including strings, dates, etc. |
| **Multiple Constraints** | Only one identity column is allowed per table. | Multiple unique constraints can be applied to different columns in the same table. |
| **Null Values** | Cannot contain null values as they must provide a unique identifier for each record. | Can contain null values; if a column allows nulls, multiple rows can have null as a value |