**Stored procedures**

Stored procedures in SQL Server are precompiled SQL code blocks that are stored and can be executed on demand. They allow you to group one or more SQL statements into a single unit of work, which can then be invoked by name.

SQL Server builds an execution plan when the stored procedure is called the first time and stores them in the cache memory. The plan is reused by SQL Server in subsequent executions of the stored procedure, allowing it to run quickly and efficiently.

Features of Stored Procedures:

* **Reduced Traffic:** A stored procedure reduces network traffic between the application and the database server, resulting in increased performance. It is because instead of sending several SQL statements, the application only needs to send the name of the stored procedure and its parameters.
* **Stronger Security:** The procedure is always secure because it manages which processes and activities we can perform. It removes the need for permissions to be granted at the database object level and simplifies the security layers.
* **Reusable:** Stored procedures are reusable. It reduces code inconsistency, prevents unnecessary rewrites of the same code, and makes the code transparent to all applications or users.
* **Easy Maintenance:** The procedures are easier to maintain without restarting or deploying the application.
* **Improved Performance:** Stored Procedure increases the application performance. Once we create the stored procedures and compile them the first time, it creates an execution plan reused for subsequent executions. The procedure is usually processed quicker because the query processor does not have to create a new plan.

Types of Stored Procedures in SQL Server (in general user-defined or system-stored):

* Simple Stored Procedures: Simple stored procedures consist of a single SQL statement or a group of SQL statements within a BEGIN...END block. They are straightforward and are often used for tasks such as querying or updating data in a database.
* Parameterized Stored Procedures: Parameterized stored procedures allow you to pass parameters to the stored procedure, enabling dynamic execution of SQL statements. Parameters can be used for filtering data, specifying values for inserts or updates, or controlling the behavior of the procedure.
* System Stored Procedures: System stored procedures are predefined procedures provided by SQL Server. They perform various administrative tasks, such as managing databases, configuring server settings, or retrieving system information. Examples include sp\_helpdb, sp\_configure, and sp\_who.
* Extended Stored Procedures: Extended stored procedures are custom procedures written in a programming language such as C or C++. They are used to extend the functionality of SQL Server beyond what is possible with T-SQL. Extended stored procedures can access external resources and perform operations not supported by standard SQL.
* CLR Stored Procedures: CLR (Common Language Runtime) stored procedures allow you to create stored procedures using .NET languages such as C# or VB.NET. These stored procedures run within the SQL Server process and provide access to the full range of .NET framework libraries and functionalities.
* Temporary Stored Procedures: Temporary stored procedures are created and exist only for the duration of a user session or a transaction. They are useful for performing temporary tasks or for creating ad-hoc procedures that do not need to be persisted.

Simple Stored Procedure:

CREATE PROCEDURE ProcedureName

AS

BEGIN

-- SQL statements

-- (e.g., SELECT, INSERT, UPDATE, DELETE)

END;

Example: Create a simple stored procedure to select all employees

CREATE PROCEDURE GetAllEmployees

AS

BEGIN

SELECT \* FROM Employees;

END;

Parameterized Stored Procedure:

CREATE PROCEDURE ProcedureName

@Parameter1 DataType,

@Parameter2 DataType

AS

BEGIN

-- SQL statements using parameters

END;

Example: Create a parameterized stored procedure to select employees by department

CREATE PROCEDURE GetEmployeesByDepartment

@DeptID INT

AS

BEGIN

SELECT \* FROM Employees WHERE DepartmentID = @DeptID;

END;

System Stored Procedure: System stored procedures do not need to be explicitly created. They are predefined procedures provided by SQL Server.

Example: Use the system stored procedure sp\_helpdb to display information about all databases

EXEC sp\_helpdb;

Extended Stored Procedure:

Syntax: Extended stored procedures are typically written in a programming language like C or C++ and then registered with SQL Server.

Example: Creating an extended stored procedure involves coding in a programming language and registering it with SQL Server.

CLR (Common Language Runtime) Stored Procedure:

CREATE PROCEDURE ProcedureName

AS EXTERNAL NAME AssemblyName.[Namespace.ClassName].[MethodName];

Example: Creating a CLR stored procedure involves writing code in a .NET language (e.g., C# or VB.NET), compiling it into an assembly, and then deploying it to SQL Server. Here's a simplified example:

using System;

using Microsoft.SqlServer.Server;

public class StoredProcedures

{

[SqlProcedure]

public static void HelloWorld()

{

SqlContext.Pipe.Send("Hello, world!");

}

}

And then, Deploy the assembly to SQL Server, and then create the stored procedure:

CREATE ASSEMBLY [HelloWorldAssembly]

FROM 'C:\Path\To\HelloWorldAssembly.dll'

WITH PERMISSION\_SET = SAFE;

CREATE PROCEDURE HelloWorldProcedure

AS EXTERNAL NAME [HelloWorldAssembly].[StoredProcedures].[HelloWorld];

Temporary Stored Procedure:

CREATE PROCEDURE #ProcedureName

AS

BEGIN

-- SQL statements

END;

Example:

-- Create a temporary stored procedure to select employees with a salary greater than a specified threshold

CREATE PROCEDURE #GetHighSalaryEmployees

@Threshold DECIMAL(10, 2)

AS

BEGIN

SELECT \* FROM Employees WHERE Salary > @Threshold;

END;

In SQL Server, when you see # in front of an object name, such as a stored procedure, it indicates that the object is temporary and stored in the temporary database (tempdb). Specifically, # denotes a local temporary stored procedure.

Temporary stored procedures are only available to the session that created them and are automatically dropped when the session ends. They are useful for storing temporary data or performing operations that are specific to a particular session or task.

When you see ## in front of an object name in SQL Server, such as a stored procedure, it indicates that the object is a global temporary object. Unlike local temporary objects (prefixed with #), global temporary objects are accessible across sessions within the same instance of SQL Server.

All together: creation of tables, insertion of data, and examples of each type of stored procedure:

-- Create Departments table

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(100)

);

INSERT INTO Departments (DepartmentID, DepartmentName)

VALUES

(1, 'Engineering'),

(2, 'Marketing'),

(3, 'Sales');

-- Create Employees table

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(100),

DepartmentID INT,

Salary DECIMAL(10, 2)

);

INSERT INTO Employees (EmployeeID, EmployeeName, DepartmentID, Salary)

VALUES

(1, 'John Doe', 1, 60000.00),

(2, 'Jane Smith', 2, 55000.00),

(3, 'Michael Johnson', 1, 62000.00),

(4, 'Emily Davis', 3, 58000.00),

(5, 'Chris Brown', 2, 54000.00);

-- 1. Simple Stored Procedure

CREATE PROCEDURE GetAllEmployees

AS

BEGIN

SELECT \* FROM Employees;

END;

-- 2. Parameterized Stored Procedure

CREATE PROCEDURE GetEmployeesByDepartment

@DeptID INT

AS

BEGIN

SELECT \* FROM Employees WHERE DepartmentID = @DeptID;

END;

-- 3. System Stored Procedure do not need to be explicitly created. They are predefined procedures provided by SQL Server.

-- 4. Extended Stored Procedure: Creating an extended stored procedure involves coding in a programming language like C or C++ and then registering it with SQL Server.

-- 5. CLR Stored Procedure: Creating a CLR stored procedure involves writing code in a .NET language, compiling it into an assembly, and then deploying it to SQL Server.

-- 6. Temporary Stored Procedure

CREATE PROCEDURE #GetHighSalaryEmployees

@Threshold DECIMAL(10, 2)

AS

BEGIN

SELECT \* FROM Employees WHERE Salary > @Threshold;

END;

Examples of calling stored procedures

-- Simple Stored Procedure

EXEC GetAllEmployees;

-- Parameterized Stored Procedure

EXEC GetEmployeesByDepartment 1;

-- Temporary Stored Procedure

EXEC #GetHighSalaryEmployees 55000.00;

Alter the stored procedure:

To alter an existing stored procedure in SQL Server, you can use the ALTER PROCEDURE statement followed by the modified procedure definition. Here's the syntax:

ALTER PROCEDURE ProcedureName

AS

BEGIN

-- Updated SQL statements

END;

Example: modify the GetEmployeesByDepartment stored procedure to include an additional parameter for filtering employees by salary range.

ALTER PROCEDURE GetEmployeesByDepartment

@DeptID INT,

@MinSalary DECIMAL(10, 2) = NULL, -- Default to NULL if not provided

@MaxSalary DECIMAL(10, 2) = NULL -- Default to NULL if not provided

AS

BEGIN

IF @MinSalary IS NOT NULL AND @MaxSalary IS NOT NULL

BEGIN

-- Filter employees by department and salary range

SELECT \*

FROM Employees

WHERE DepartmentID = @DeptID

AND Salary BETWEEN @MinSalary AND @MaxSalary;

END

ELSE

BEGIN

-- Filter employees by department only

SELECT \*

FROM Employees

WHERE DepartmentID = @DeptID;

END

END;

To run: EXEC GetEmployeesByDepartment 1, 55000, 65000

In this example, we added two new parameters @MinSalary and @MaxSalary to the GetEmployeesByDepartment stored procedure. The procedure now filters employees by both department ID and salary range, allowing for more flexibility in querying.

To simplify:

ALTER PROCEDURE GetEmployeesByDepartment

@DeptID INT,

@MinSalary DECIMAL(10, 2) = NULL,

@MaxSalary DECIMAL(10, 2) = NULL

AS

BEGIN

IF @MinSalary IS NOT NULL AND @MaxSalary IS NOT NULL

SELECT \*

FROM Employees

WHERE DepartmentID = @DeptID

AND Salary BETWEEN @MinSalary AND @MaxSalary;

ELSE

SELECT \*

FROM Employees

WHERE DepartmentID = @DeptID;

END;

Rename Stored Procedure:

EXEC sp\_rename 'GetEmployeesByDepartment','GetEmployeesByDepartment11'

Drop Stored Procedure:

To drop a stored procedure in SQL Server, you can use the DROP PROCEDURE statement followed by the name of the procedure you want to drop.

DROP PROCEDURE [schema\_name.]procedure\_name;

Example:

DROP PROCEDURE dbo.GetEmployeesByDepartment;

When we have several procedures, it is very important to list all procedures. Because sometimes the procedure names are the same in many databases. In that case, you can list all stored procedure in the current database as follows:

SELECT \* FROM sys.procedures;

The best way for listing all user-defined stored procedures in a database is to use the ROUTINES information schema view as below:

SELECT ROUTINE\_SCHEMA, ROUTINE\_NAME

FROM INFORMATION\_SCHEMA.ROUTINES

WHERE ROUTINE\_TYPE = 'PROCEDURE';

Disadvantages of Stored Procedures

* Debugging: Since debugging stored procedures is never simple, it is not advised to write and execute complex business logic using them. As a result, if we will not handle it properly, it can result in a failure.
* Dependency: Professional DBAs and database developers handle vast data sets in large organizations. And the application developers must depend on them because any minor changes must be referred to a DBA, who can fix bugs in existing procedures or build new ones.
* Expensive: Stored procedures are costly to manage in terms of DBAs because organizations would have to pay extra costs for specialist DBAs. A DBA is much more qualified to handle complex database procedures.
* Specific to a Vendor: Stored procedures written in one platform cannot run on another. Since procedures written in Oracle are more complicated, we will need to rewrite the entire procedure for SQL Server.

**Difference between Function and Procedure in SQL Server**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Function** | **Procedure** |
| Basic principles | Functions use the input provided to calculate the output of a program. | Procedures use the inputs provided to determine which tasks to complete in what order. |
| Try-Catch Blocks | The try-catch Blocks are not supported by functions. | Try-catch Blocks are supported by procedures. |
| SQL Query | Within an SQL query, we can invoke a function. | An SQL query cannot call a procedure. |
| SELECT | Function calls may be present in the SELECT statements. | Procedure calls cannot ever be made in the SELECT statements. |
| Return | To the caller function or code, a function would return the returned value or control. | In contrast, a procedure would return control but neither the caller function nor the code would receive any value from it. |
| Call | A procedure can be used to call a function. | No function can be used to call a procedure. |
| Compilation | When we call a function within a program, it gets compiled. | The procedures must be compiled once, but if more than one is needed, they can be called repeatedly; a compilation is not required each time. |

Note

1. Unlike the Stored Procedure, the Function returns only a single value.
2. Unlike the Stored Procedure, the Function accepts only input parameters.
3. Unlike the Stored Procedure, the Function is not used to Insert, Update, or Delete data in a database table(s).
4. Like the Stored Procedure, the Function can be nested up to 32 levels.
5. User Defined Function can have up to 1023 input parameters while a Stored Procedure can have up to 2100 input parameters.
6. User Defined Function can't return XML Data Type.
7. User Defined Function doesn't support Exception handling.
8. User Defined Function can call only Extended Stored Procedure.

**Triggers**

In SQL Server, triggers are special types of stored procedures that are automatically executed or fired in response to specific events or actions performed on a table or view in the database. These events can include INSERT, UPDATE, or DELETE operations.

Syntax:

CREATE TRIGGER trigger\_name

ON table\_name

AFTER INSERT, UPDATE, DELETE

AS

BEGIN

-- SQL statements to be executed when the trigger fires

END;

* CREATE TRIGGER: This statement is used to create a new trigger.
* trigger\_name: This is the name you assign to the trigger.
* ON table\_name: Specifies the table on which the trigger should be associated.
* AFTER INSERT, UPDATE, DELETE: Specifies the action or actions that will trigger the execution of the trigger. In this example, the trigger will fire after an INSERT, UPDATE, or DELETE operation.
* AS: Begins the body of the trigger.
* BEGIN and END: Encloses the SQL statements that should be executed when the trigger is fired.

Within the body of the trigger, you can include any SQL statements that you want to execute when the trigger is fired. These statements can manipulate data in the table or perform other tasks based on the event that triggered the trigger.

It's important to note that triggers can have a significant impact on performance and can make the database behavior less predictable, so they should be used only when necessary. Additionally, care should be taken to ensure that triggers do not cause unintended side effects or create recursive loops.

Types of triggers in SQL Server:

Triggers can be categorized based on the type of SQL operations they are associated with:

1. DML Triggers (Data Manipulation Language Triggers):

These triggers are fired in response to Data Manipulation Language (DML) operations such as INSERT, UPDATE, and DELETE on a table.

DML triggers can be further classified into:

* After Triggers: Fired after the DML operation (INSERT, UPDATE, DELETE) has occurred.
* Instead Of Triggers: Fired instead of the DML operation, allowing custom logic to be executed before or instead of the operation.

In other words, Instead Of trigger fires before the SQL Server starts the execution of the operation that fired it. This is much more different from the AFTER trigger, which fires after the action that caused it to fire. Remember, when we have a successfully executed INSTEAD OF insert/update/delete trigger on a table, it does not include the actual insert/update/delete to the table.

In real-time applications, Instead Of Triggers are used to correctly update a complex view.

1. DDL Triggers (Data Definition Language Triggers):

* These triggers are fired in response to Data Definition Language (DDL) operations such as CREATE, ALTER, and DROP statements affecting database objects like tables, views, stored procedures, etc.
* DDL triggers can be used to enforce policies, audit schema changes, or perform actions based on changes to the database structure.

1. Logon Triggers

* Logon triggers are to perform actions or enforce policies when a user logs in to the SQL Server instance. They are fired automatically on a LOGON event. They are DDL triggers and are created at the server level. You can define more than one LOGON trigger on a server.
* A LOGON trigger can be used in controlling server sessions by tracking login activity, restricting logins to the SQL Server, or limiting the number of sessions for a particular login.

Additionally, triggers can be further classified based on the number of times they are executed:

* Single-Statement Triggers:

These triggers are designed to handle a single row operation (e.g., a single INSERT, UPDATE, DELETE).

They are more efficient but may not handle batch operations effectively.

* Multi-Statement Triggers:

These triggers are capable of handling multiple row operations, such as batch INSERT, UPDATE, DELETE statements.

They are less efficient compared to single-statement triggers but offer more flexibility.

**Syntax and example:**

1. DML Triggers:

a. After Triggers:

CREATE TRIGGER trigger\_name

ON table\_name

AFTER INSERT, UPDATE, DELETE

AS

BEGIN

-- Trigger logic here

END;

Example: Suppose we have a table called Employee with columns EmployeeID, Name, and Salary. We want to create an after trigger that logs the details of any changes made to this table.

CREATE TABLE Employee (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100),

Salary DECIMAL(10,2)

);

INSERT INTO Employee (EmployeeID, Name, Salary)

VALUES (1, 'John Doe', 50000),

(2, 'Jane Smith', 60000);

After Trigger Example:

CREATE TRIGGER AfterEmployeeChanges

ON Employee

AFTER INSERT, UPDATE, DELETE

AS

BEGIN

IF EXISTS(SELECT \* FROM inserted)

BEGIN

PRINT 'Employee data has been modified:';

SELECT 'New EmployeeID: ' + CAST(EmployeeID AS VARCHAR(10)),

'New Name: ' + Name,

'New Salary: ' + CAST(Salary AS VARCHAR(20))

FROM inserted;

END;

IF EXISTS(SELECT \* FROM deleted)

BEGIN

PRINT 'Employee data has been deleted:';

SELECT 'Deleted EmployeeID: ' + CAST(EmployeeID AS VARCHAR(10)),

'Deleted Name: ' + Name,

'Deleted Salary: ' + CAST(Salary AS VARCHAR(20))

FROM deleted;

END;

END;

To test the AfterEmployeeChanges trigger created earlier, perform INSERT, UPDATE, or DELETE operations on the Employee table and observe the results.

Let's go through each operation:

* Insert Operation: Insert a new record into the Employee table:

INSERT INTO Employee (EmployeeID, Name, Salary) VALUES (3, 'Alice Johnson', 70000);

After executing this INSERT statement, the trigger will fire and display the message along with the details of the newly inserted employee.

* Update Operation: Update the salary of an existing employee in the Employee table:

UPDATE Employee SET Salary = 75000 WHERE EmployeeID = 1;

After executing this UPDATE statement, the trigger will fire and display the message along with the details of the updated employee.

* Delete Operation: Delete an existing record from the Employee table:

DELETE FROM Employee WHERE EmployeeID = 2;

After executing this DELETE statement, the trigger will fire and display the message along with the details of the deleted employee.

In the case of DELETE operation, if you delete multiple records from the Employee table, by using a single DELETE statement again and again, the trigger will fire once and will display the details of the last operation (the last deleted record).

If you want to log or display information about all the operations performed within a single trigger firing, you would need to store the details of each operation in a temporary table or use some other method to accumulate the information and display it at the end of the trigger logic.

b. Instead Of Triggers:

CREATE TRIGGER trigger\_name

ON view\_name

INSTEAD OF INSERT, UPDATE, DELETE

AS

BEGIN

-- Trigger logic here

END;

Example: Suppose we have a view EmployeeView that joins data from Employee and Department tables. We want to create an instead of trigger to handle INSERT, UPDATE, and DELETE operations on this view.

CREATE VIEW EmployeeView AS

CREATE VIEW EmployeeView

AS select \* from Employee

Instead Of Trigger Example:

CREATE TRIGGER InsteadOfEmployeeViewChanges

ON EmployeeView

INSTEAD OF INSERT, UPDATE, DELETE

AS

BEGIN

-- Trigger logic here

PRINT 'Instead of trigger fired for EmployeeView';

-- Handle INSERT, UPDATE, DELETE operations as required

END;

And to test: INSERT INTO dbo.EmployeeView VALUES (11,'Bob Dion',32500)

2. DDL Triggers:

Syntax:

CREATE TRIGGER trigger\_name

ON ALL SERVER | DATABASE

FOR CREATE\_TABLE, ALTER\_TABLE, DROP\_TABLE, etc.

AS

BEGIN

-- Trigger logic here

END;

There are two types of DDL triggers available in SQL Server:

* Database Scoped DDL Trigger
* Server Scoped DDL Trigger

The DDL triggers can be created in a specific database or at the server level. If we set the scope to server-level then it is applied to all the databases of that server.

Example for database level:

CREATE TRIGGER trRestrictCreateTable

ON DATABASE

FOR CREATE\_TABLE

AS

BEGIN

PRINT 'YOU CANNOT CREATE A TABLE IN THIS DATABASE'

ROLLBACK TRANSACTION

END

CREATE TRIGGER trRestrictAlterTable

ON DATABASE

FOR ALTER\_TABLE

AS

BEGIN

PRINT 'YOU CANNOT ALTER TABLES'

ROLLBACK TRANSACTION

END

CREATE TRIGGER trRestrictDropTable

ON DATABASE

FOR DROP\_TABLE

AS

BEGIN

PRINT 'YOU CANNOT DROP TABLES'

ROLLBACK TRANSACTION

END

And, all together:

CREATE TRIGGER trRestrictDDLEvents

ON DATABASE

FOR CREATE\_TABLE, ALTER\_TABLE, DROP\_TABLE

AS

BEGIN

PRINT 'You cannot create, alter or drop a table'

ROLLBACK TRANSACTION

END

##### **Drop a Database Scoped DDL trigger:**

DROP TRIGGER trRestrictCreateTable ON DATABASE

DROP TRIGGER trRestrictAlterTable ON DATABASE

DROP TRIGGER trRestrictDropTable ON DATABASE

To test:

Create Table University1(UniID int, UniName varchar);

Alter table University add location varchar;

drop table University

Note: ROLLBACK TRANSACTION is to undo and erase all data modifications made from the start of the transaction or to a savepoint. It also frees resources held by the transaction. This does not include changes made to local variables or table variables. These are not erased by this statement.