**Session Week 6**

**Stored Procedures:**

Stored procedures and triggers are precompiled objects that contain Transact-SQL or .Net logic that can be reused within the database. Each object provides different capabilities that can be used within SQL Server 2012.

Stored procedures are objects that are executed by clients to perform data modification or to query the data. The precompiled nature of stored procedures makes them efficient objects to access the data. Triggers are event driven and will execute in response to a specific event that occurs against an object. This enables you to enforce business logic or complex data integrity.

The logic that is contained within a stored procedure or trigger can include error handling to create robust objects that can validate information that is passed into the stored procedure. Furthermore, you can control the execution context of a stored procedure or trigger to execute under a different account for greater control over security.

Stored procedure and triggers are important database objects. Understanding how they are implemented will help you make better use of the database in which they operate in.

Stored procedure is a SQL Server object that is a named collection of precompiled T-SQL statements. This allows you to embed repetitive or complex T-SQL code that can be executed efficiently. Stored procedures provide support for local and global variables, control-of-flow language elements, and other advanced T-SQL programming logic.

This allows interaction that can include querying or modifying data, or calling other stored procedures and SQL Server objects to perform tasks.

Stored procedures must be called by a client application using the Execute statement. Once executed, the stored procedure will store a copy of its execution plan within the buffer cache. This means that subsequent execution of the same stored procedure is faster as the execution plan is contained within memory.

**STORED PROCEDURE TYPES:**

System Stored Procedures - pre-exists within a SQL Server instance and can be used to perform the following administrative functions:

* To query SQL Server and database metadata.
* To set up SQL Server objects, including replication and SQL Server Agent job, as a wide range of other SQL Server objects.

**User Defined Stored Procedures:**

User defined stored procedures allows you to embed custom repetitive or complex T-SQL code that support local and global variables and control-of-flow language elements. IN SQL Server 2012, a stored procedure can be of two types: T-SQL or Common Language Runtime (CLR).

Transact SQL user defined stored procedure might contain the statements needed to insert a new row into one or more tables based on information supplied by the client application. Or, the stored procedure might return data from the database to the client application.

A CLR user defined stored procedure uses the .NET Framework CLR methods that can take and return user-supplied parameters. This enables your stored procedures to access data and objects outside of SQL Server.

Extended stored procedures are legacy SQL Server objects that enable you to access resources outside of the SQL Server environment. An example can include making a call to an external application such as CommandExec. This feature will be removed in future version of SQL Server. Avoid using this feature in new development work and consider replacing with CLR user defined stored procedures.

**Benefits of Stored Procedures and Triggers**

Stored procedures and triggers are powerful SQL Server objects that can be used to enforce complex business logic against a SQL Server instance or database. The following provides examples where stored procedures and triggers are useful to an organization.

Stored procedures provide the following benefits.

* They can encapsulate reusable logic in single objects in a central place.
* They can hide the details of the objects that the stored procedure interacts with.
* They can act as a security mechanism as access to the database can be controlled through stored procedures.
* They are pre-compiled objects that store a copy of the execution plan in memory, therefore improving performance.
* Stored procedures can reduce network traffic by reducing the amount of Transact-SQL statements sent over the network.

**DML Triggers provide the following benefits.**

* They can cascade changes through related tables in the database. However, these changes can be executed more efficiently using cascading referential integrity constraints.
* They can guard against malicious or incorrect INSERT, UPDATE, and DELETE operations and enforce other restrictions that are more complex than those defined with CHECK constraints.
* They can evaluate the state of a table before and after a data modification and take actions based on that difference.
* Multiple DML triggers of the same type (INSERT, UPDATE, or DELETE) on a table allow multiple, different actions to take place in response to the same modification statement.

**DDL Triggers provide the following benefits.**

* You want the ability to prevent certain changes to your database schema.
* You want an update to occur in the database in response to a change in your database schema.
* You want to record changes or events in the database schema.

**Syntax for Stored Procedures**

The creation of stored procedures can be as simple or complex as you wish it to be. Stored procedures provide support for local and global variables, control-of-flow language elements, and other advanced Transact-SQL programming logic. A stored procedure can be created by defining a stored procedure name followed by the stored procedure logic. However, when working with parameters and variables, the syntax required to incorporate these parameters extends the Transact-SQL required to create the stored procedure.

**The partial syntax for creating a simple stored procedure is:**

CREATE {PROC|PROCEDURE} [ schema\_name . ] procedure\_name

AS

Transact-SQL\_statement

CREATE {PROC|PROCEDURE} is used to define the name of the stored procedure and the schema where the object will live.

Transact-SQL\_statementis defined after the AS clause that can use a SELECT, INSERT, UPDATE, DELETE, and MERGE statement to define the custom stored procedure logic.

Below is an example for creating a stored procedure named SalesDetails held in the dbo schema that uses a SELECT statement and returns Store\_Name, SalesOrderNumber, OrderDate, and TotalDue.

CREATE PROC dbo.SalesDetails

AS

SELECT S.Name AS STORE\_NAME, SO.SalesOrderNumber, SO.OrderDate, SO.TotalDue

FROM Sales.Store AS S

JOIN Sales.SalesOrderHeader AS SO

ON S.CustomerID = SO.CustomerID

GO

The following example shows how you can call the stored procedure to return back the result set. Like Views, you can use the sp\_helptext system stored procedure to find out the definition of the stored procedure.

EXEC dbo.SalesDetails

The syntax can be extended to include a number of options that can be defined within the stored procedure definition.

**Syntax for Altering and Dropping Stored Procedures:**

The process of changing a stored procedure is similar to creating a stored procedure as you must redefine the stored procedure logic with the change for the modification to the stored procedure to be successful. The process of deleting a stored procedure is performed by using the DROP PROCEDURE or DROP PROC statement. The table below provides examples of altering and dropping stored procedures.

**ALTER**

In order to modify a stored procedure, you must use ALTER PROCEDURE or ALTER PROC Transact-SQL statement.

The partial syntax for altering a simple stored procedure is as follows:

ALTER {PROC | PROCEDURE} [ schema\_name . ] procedure\_name

AS

Transact-SQL\_statement

ALTER {PROC | PROCEDURE} is used to define the name of the stored procedure and the schema of the stored procedure object.

Transact-SQL\_statement can be defined after the AS clause that can include a SELECT, INSERT, UPDATE, DELETE, and MERGE statement.

Below is an example for altering a stored procedure named SalesDetails held in the dbo schema created in the last topic that excluded the OrderDate and TotalDue columns.

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

ALTER PROCEDURE dbo.SalesDetails

AS

SELECT S.Name AS STORE\_NAME, SO.SalesOrderNumber

FROM Sales.Store AS S

JOIN Sales.SalesOrderHeader AS SO

ON S.CustomerID = SO.CustomerID

GO

The following example shows how you can call the stored procedure in the same way to return back the result set. Like Views, you can use the sp\_helptext system stored procedure to find out the definition of the stored procedure.

EXEC dbo.SalesDetails

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

**DROP**

The syntax for deleting a stored procedure is as follows:

DROP {PROC | PROCEDURE} [ schema\_name . ] procedure\_name

DROP {PROC | PROCEDURE} is used to define the name of the stored procedure and the schema of the stored procedure object.

Below is an example for removing a stored procedure named SalesDetails held in the dbo schema.

DROP PROC dbo.SalesDetails

**Parameterized Stored Procedures**

Parameterized stored procedures increase the flexibility of stored procedures by allowing you to pass information into the stored procedure in the form of an input parameter. You can also return information back to the stored procedure by using output parameters.

To use a stored procedure that uses input parameters, you must declare one or more variables as parameters within the CREATE PROCEDURE statement.

To use a stored procedure that uses output parameters, you should declare one or more variables as parameters with the OUTPUT clause within the CREATE PROCEDURE statement. You must also specify the OUTPUT clause as you use the EXECUTE statement to execute the stored procedure.

The partial syntax for creating a simple stored procedure is as follows:

CREATE {PROC |PROCEDURE} [

schema\_name .

]

procedure\_name

AS

Transact-SQL\_statement

When using parameters, the syntax can be extended to allow the use of parameters with the following syntax.

CREATE {PROC |PROCEDURE} [ schema\_name . ] procedure\_name

[{@parametername data type} [OUTPUT]]

AS

Transact-SQL\_statement

Below is an example for creating a stored procedure named SalesDetails held in the dbo schema that uses a select statement that returns Store\_Name, SalesOrderNumber, OrderDate, and TotalDue. The CREATE procedure uses an input parameter named @StoreName using the data type of Varchar(20). The @StoreName parameter passes the input value to a variable within the WHERE clause of the SELECT statement.

CREATE PROC dbo.SalesDetails

@StoreName Varchar(20)

AS

SELECT S.Name AS STORE\_NAME, SO.SalesOrderNumber, SO.OrderDate, SO.TotalDue

FROM Sales.Store AS S

JOIN Sales.SalesOrderHeader AS SO

ON S.CustomerID = SO.CustomerID

WHERE S.Name = @StoreName

GO

The following example calls the stored procedure with an input parameter to return back the result set where the Store Name is Manchester.

EXEC dbo.SalesDetails ‘Manchester’

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

**Triggers:**

Triggers are a special type of stored procedure that are event driven, rather than being driven by a call from a client application. Like stored procedures, triggers are precompiled objects that can encapsulate logic within a SQL Server trigger. They can also make calls to other SQL Server objects to perform other tasks.

Unlike stored procedures, triggers are created against a table or a view and will only execute when a predefined event, such as an INSERT, UPDATE, or DELETE statement is issued against the table or view. The trigger automatically becomes part of the transaction that fired the trigger. If a violation occurs, the trigger can then rollback the transaction that fired the trigger.

As a result, triggers can be used to provide data integrity mechanisms that may not be possible with other data integrity objects.

**There are two types of triggers that can be created:**

* Data Modification Language or DML triggers that can operate on tables or views.
* Data Definition triggers or DDL triggers that can operate at a SQL Server instance or database level.

**DML Trigger Types:**

**BEFORE or INSTEAD OF triggers** - are executed in place of a triggering action.

ex. HumanResources.Employee table:

The INSTEAD OF trigger is triggered every time you run a DELETE operation on the HumanResources.Employee table.

When executing the INSTEAD OF trigger, SQL Server rolls back the transaction if a user tries to delete an Employee. This is because, according to the business rules, deleting employees is not allowed. You can only have one INSTEAD OF trigger on the same table and same action query type. There can be at most three INSTEAD OF triggers associated with a table, one for each action query type: INSERT, UPDATE, and DELETE.

**AFTER triggers** are executed following a triggering action, such as INSERT, UPDATE, or DELETE.

Example: in the HumanResources.Employee table, the ModifiedDate column of that table gets updated with the current date for all the rows that are affected.

This saves you from updating the ModifiedDate column every time you update the table. There can be several AFTER triggers associated with a single table. An AFTER trigger fires only after the action query that invoked the AFTER trigger has modified the data.

Database - Database level DDL triggers are used to maintain the integrity of the SQL Server objects that exist within a database. For example, a DDL trigger can be created to audit when modifications are made to a table by setting the DDL Trigger to be invoked on the ALTER TABLE statement. The trigger can then write information to a table that holds information about when the modification occurred and who made the modification.

**Syntax for DML Triggers**

DML triggers implement complex business logic on data that is modified using INSERT, UPDATE, and DELETE statements against a table or view. Triggers have the ability to query hidden tables that store copies of the data during DML operations. The table below outlines the syntax and examples of implementing DML Triggers.

**The following is syntax for creating a DML trigger.**

CREATE TRIGGER [ schema\_name . ]trigger\_name

ON { table | view }

[ WITH <dml\_trigger\_option>[ ,...n ] ]

{ FOR | AFTER | INSTEAD OF }

{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }

[ WITH APPEND ]

[ NOT FOR REPLICATION ]

AS { sql\_statement [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] > }

<dml\_trigger\_option> ::=

[ ENCRYPTION ]

[ EXECUTE AS Clause ]

<method\_specifier> ::=

assembly\_name.class\_name.method\_name

An insert trigger fires when an insert is performed on the table or view to which the trigger is assigned. During an insert operation, not only is the data placed in the table, but also in a hidden table named inserted. This hidden table can be queried and an audit trail of data inserted into the table could be maintained in another table.

The following example creates a trigger on the Production.Products table named ProductInsert. This trigger queries the inserted table for any new products that are inserted in the Production.Products table and then inserts those records in the Admin.Products table.

CREATE TRIGGER ProductInsert ON Production.Products

AFTER INSERT AS BEGIN SET NOCOUNT ON;

INSERT INTO Admin.Products (ProductID, TransactDate, Quantity)

SELECT inserted.ProductID, GetDate(), inserted.Quantity FROM inserted

A delete trigger fires when a delete is performed on the table or view to which the trigger is assigned. During a delete operation, not only is the data placed in the table, but also in a hidden table named deleted. This hidden table can be queried and an audit trail of deleted data can be maintained in another table.

The following example creates a trigger on the Production.Products table named ProductDelete that rolls back the deleted transaction when a record of any product is deleted in the Production.Products table.

CREATE TRIGGER ProductDelete ON Production.Products AFTER DELETE AS ROLLBACK TRANSACTION PRINT ‘Products cannot be deleted’

An update trigger fires when an update is performed on the table or view to which the trigger is assigned. An update operation makes use of both the inserted and the deleted table.

The following example creates a trigger on the Production.Products table named ProductUpdate that rolls back the updated transaction when a record of any product is updated in the Production.Products table.

CREATE TRIGGER ProductUpdate ON Production.Products AFTER UPDATE AS ROLLBACK TRANSACTION PRINT ‘Products cannot be updated’

**Nested and Recursive Triggers**

Nested Triggers. In some scenarios, a trigger fired in one table could place records in another table. This in turn would fire another trigger that would acts on other tables. This type of trigger is referred to as nested triggers.

SQL Server 2012 can allow nested triggers that can nest up to 32 levels deep and is enabled by default. Nesting can be disabled by using the following steps.

sp\_configure 'nested triggers', 0

**An alternate method to disable nested triggers is as follows:**

* In Object Explorer, right-click a server, and then select Properties.
* On the Advanced page, set the Allow Triggers to Fire Others option to False.

**Recursive triggers**

A recursive trigger causes the same trigger to fire, whether directly or indirectly. Recursive triggers are disabled by default, but can be enabled with either of the following Transact-SQL statements.

ALTER DATABASE databasename SET RECURSIVE\_TRIGGERS ON

or

sp\_dboption databasename 'recursive triggers', True

**Syntax for DDL Triggers**

DDL triggers primarily correspond to Transact-SQL CREATE, ALTER, and DROP statements and system stored procedures that perform DDL-like operations. The following is the syntax for a DDL trigger.

CREATE TRIGGER trigger\_name

ON { ALL SERVER | DATABASE }

[ WITH <ddl\_trigger\_option> [ ,...n ] ]

{ FOR | AFTER } { event\_type | event\_group } [ ,...n ]

AS { sql\_statement [ ; ] [ ,...n ] | EXTERNAL NAME < method specifier > [ ; ] }

<ddl\_trigger\_option>::=

[ ENCRYPTION ]

[ EXECUTE AS Clause ]

<method\_specifier> ::=

assembly\_name.class\_name.method\_name

CREATE TRIGGER trigger\_name defines the name of the trigger.

ON { ALL SERVER | DATABASE } defines the scope of the DDL trigger. A DDL trigger can be defined at the server level or the database level.

[ WITH <ddl\_trigger\_option> [ ,...n ] ] enables you to define if the trigger definition is encrypted and/or the EXECUTE AS clause is used to change the security context of the trigger.

{ FOR | AFTER } { event\_type | event\_group } [ ,...n ] determines the type of event that the trigger will execute for.

AS { sql\_statement [ ; ] [ ,...n ] | EXTERNAL NAME < method specifier > [ ; ] } defines the Transact-SQL statement or .Net assembly that is executed as part of the triggering action.

**Syntax for DDL Triggers**

DDL triggers primarily correspond to Transact-SQL CREATE, ALTER, and DROP statements and system stored procedures that perform DDL-like operations. The following is the syntax for a DDL trigger.

CREATE TRIGGER trigger\_name

ON { ALL SERVER | DATABASE }

[ WITH <ddl\_trigger\_option> [ ,...n ] ]

{ FOR | AFTER } { event\_type | event\_group } [ ,...n ]

AS { sql\_statement [ ; ] [ ,...n ] | EXTERNAL NAME < method specifier > [ ; ] }

<ddl\_trigger\_option>::=

[ ENCRYPTION ]

[ EXECUTE AS Clause ]

<method\_specifier> ::=

assembly\_name.class\_name.method\_name

CREATE TRIGGERtrigger\_namedefines the name of the trigger.

ON { ALL SERVER | DATABASE } defines the scope of the DDL trigger. A DDL trigger can be defined at the server level or the database level.

[ WITH <ddl\_trigger\_option> [ ,...n ] ] enables you to define if the trigger definition is encrypted and/or the EXECUTE AS clause is used to change the security context of the trigger.

{ FOR | AFTER } { event\_type | event\_group } [ ,...n ] determines the type of event that the trigger will execute for.

AS { sql\_statement [ ; ] [ ,...n ] | EXTERNAL NAME < method specifier > [ ; ] } defines the Transact-SQL statement or .Net assembly that is executed as part of the triggering action.

**Structured Error Handling**

A common method that is used to manage structured error handling is the TRY...CATCH Transact-SQL syntax.

BEGIN TRY

{sql\_statement| statement block}

END TRY

BEGIN CATCH

{sql\_statement| statement block}

END CATCH

The TRY portion of the Transact-SQL statement contains the transaction. If the transaction fails, the CATCH portion of the Transact-SQL statement contains the structured error handling that manages the failed transaction. It is important that the CATCH part of the statement occurs immediately after the TRY statement.

The following example tries to insert data into the Production.Product table. If the transaction fails, the transaction is rooted back and a message is returned.

BEGIN TRY

INSERT INTO Production.Product (ProductID, Name) VALUES (1, 'Widgets')

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION

PRINT 'The following error: ' + ERROR\_NUMBER() + ' , occurred with the message:- ' + ERROR\_MESSAGE()

END CATCH

Error Handling Guidelines

Use the following guidelines when implementing structured error handling within stored procedures and triggers.

* Consider the common errors that can occur within the stored procedures and triggers that you create.
* Create CATCH block immediately after the TRY block.
* Rollback failed transactions within the CATCH statement.
* Use the ERROR\_NUMBER() and ERROR\_MESSAGE() system functions to provide information about the error that caused a transaction to fail.
* When returning messages back to the client application, ensure that the message provides useful information that can be used.
* Use the ERROR\_LINE() system function to return the line number in which the error occurred.