# **Trigger**

## **Introduction**

**A trigger is a database object that runs automatically when an event occurs.**

**These are logic’s like stored procedures that can be executed automatically before the Insert, Update or Delete happens in a table or after the Insert, Update, or Delete happens in a table.**

* In simple words, we can say that, if you want to execute some ***pre-processing or post-processing logic*** before or after the *Insert, Update, or Delete* in a table then you need to use Triggers in SQL Server.

### **In SQL Server, there are 3 types of triggers.**

1. **DML Trigger – Data Manipulation Language Triggers.**

* These are fired automatically in response to DML events (Insert, Update, delete)
* DML trigger can be classified into 2 types: After Tigger and Instead Of trigger.
* ***After Trigger***, fire after the triggering action. The Insert, update and delete statements, causes an after trigger to fire the respective statement complete.
* ***Instead Trigger***, fires instead the triggering action. The Insert, update and delete statements, causes an instead trigger to fire the respective statement complete.

1. **DDL Trigger -- Data Definition Language Triggers**

* The DDL triggers in SQL Server are fired in response to a variety of data definition language (DDL) events such as Create, Alter, Drop, Grant, Deny, and Revoke (Table, Function, Index, Stored Procedure, etc…).
* That means DDL triggers in SQL Server are working on a database.

1. **Logon Trigger**

* The Logon Triggers in SQL Server are the special kind of stored procedure or we can also say a special type of operation which fire or executed automatically in response to a LOGON event and moreover, we can define more than one Logon trigger on the server.
* The Logon triggers are fired only after the successful authentication but before the user session is actually established.
* If the authentication is failed then the logon triggers will not be fired.

1. **CLR Trigger – Common Language Runtime Triggers**

* We can create a database object inside SQL Server that is programmed in an assembly created in the Microsoft .NET Framework common language runtime (CLR).
* Database objects that can leverage the rich programming model provided by the CLR include DML triggers, DDL triggers, stored procedures, functions, aggregate functions, and types.

**Note: We can find newly created trigger. [Server > Databases > Database Name > Tables > Triggers].**

Whenever you fire any ***INSERT****,* ***UPDATE****, and* ***DELETE*** statement on a table, all the new records are actually going to the inserted table i.e. all the updated and new records are present in the inserted table. On the other hand, all the old values are present in the deleted table.

**Triggers are used for several purposes:**

1. **Produce additional checking during insert, update or delete operations on the affected table.**
2. **They allow us to encode complex default values that cannot be handled by default constraints.**
3. **Implement referential integrity across databases.**

* You can read more about this in this tip: SQL Server Referential Integrity across Databases Using Triggers.

1. **They allow us to control what actually happens when one performs an insert, update, or delete on a view that accesses multiple tables.**
2. **You can calculate aggregated columns in a table using triggers.**
3. **Log table modifications.**

* Some tables have sensitive data such as customer email, employee salary, etc., that you want to log all the changes.
* In this case, you can create the UPDATE trigger to insert the changes into a separate table.

1. **Enforce complex integrity of data.**

* In this scenario, you may define triggers to validate the data and reformat the data if necessary.
* For example, you can transform the data before insert or update using a BEFORE INSERT or BEFORE UPDATE trigger.

## **When to Use Triggers: Benefits, Considerations, and Examples**

Triggers can be useful for implementing business rules, enforcing data integrity, and automating certain tasks. Here are some benefits, considerations, and examples of when to use triggers in SQL Server:

### **Benefits of using triggers:**

1. **Data integrity enforcement**: Triggers can help ensure that data remains consistent and adheres to predefined rules.

* For example, you can use a trigger to validate or modify data before it is inserted, updated, or deleted in a table.

1. **Business rule implementation**: Triggers allow you to implement complex business rules that involve multiple tables or data dependencies.

* They can enforce specific conditions or actions based on the data being manipulated.

1. **Audit and logging**: Triggers can be used to track changes made to the database by capturing information about the modified data or generating audit logs.

* This can be helpful for compliance, troubleshooting, or historical analysis.

1. **Data synchronization**: Triggers can be used to propagate changes made to one table to other related tables, ensuring data consistency across multiple entities.
2. **Automation of tasks**: Triggers can automate certain tasks or workflows based on specific events.

* For example, you can use a trigger to send notifications or perform calculations whenever a certain condition is met.

### **Considerations when using triggers:**

1. **Performance impact**: Triggers introduce additional processing overhead, as they are executed synchronously with the triggering action.

* Poorly designed triggers or excessive use of triggers can negatively impact database performance.

1. **Complexity and maintainability**: Triggers can introduce additional complexity to the database schema and logic.

* It's important to design and document them carefully to ensure they are understandable and maintainable.

1. **Implicit behavior**: Triggers can fire automatically without explicit invocation, which may lead to unexpected behavior if not properly understood or accounted for in application logic.

### **Examples of trigger usage:**

1. **Enforcing Business Rules:** Use triggers to enforce complex business rules, such as validating data based on certain conditions, preventing data modifications that violate specific rules, or applying data transformations before inserting or updating data.
2. **Auditing and Logging:** Use triggers to capture and log changes made to a table, including the old and new values, timestamps, and user information.

* This can be helpful for compliance requirements or tracking data changes.

1. **Data Synchronization:** Use triggers to automatically update related data in other tables when changes occur in a primary table.

* For example, if a customer's address is updated, you can use a trigger to update the address in all related orders.

1. **Referential Integrity:** Use triggers to enforce referential integrity by automatically deleting or updating child records when changes occur in a parent table, ensuring that foreign key relationships are maintained.

# DML Trigger

**This type of trigger is the most known and used by developers.**

**DML stands for Data Manipulation Language and refers to the SQL instructions that changes data**.

* Those instructions are INSERT, UPDATE and DELETE.
* Basically, DML triggers can be defined as pieces of code written mostly in Transact SQL language whose execution is not performed manually by the user and instead is run automatically in response to DML events.

**DML Triggers are associated to a table or view and to any of the DML events (*INSERT, UPDATE and DELETE*).**

**Something to remark is that a trigger can only be associated with one single table or view, but can be associated to more than one DML event.**

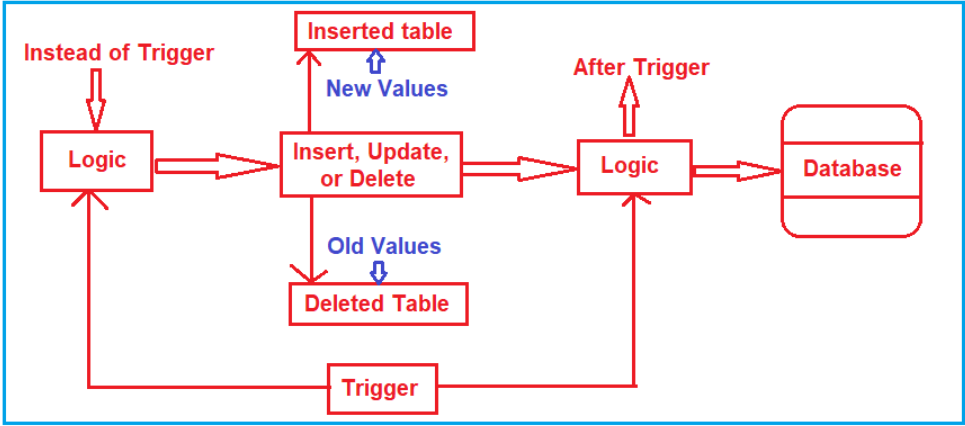
* For instance, you could have a trigger that is associated with a table that fires in response to INSERT and UPDATE events.

**There are two types of triggers. They are as follows:**

1. ***Instead of Triggers***: The Instead Of triggers are going to be executed instead of the corresponding DML operations.

* That means *instead of the DML operations such as Insert, Update, and Delete,* the Instead Of triggers are going to be executed.

1. ***After Triggers***: The After Triggers fires in SQL Server execute after the triggering action.

* ****That means once the DML statement (such as Insert, Update, and Delete) completes its execution, this trigger is going to be fired.

## **After Trigger**

* Sometime called as FOR triggers.
* These kinds of triggers fire after the execution of an action query that can be either DDL statements like Create, Alter and Drop or DML statements like Insert, Update and Delete.
* Whenever you fire any *INSERT, UPDATE, and DELETE* statement on a table, all the new records are actually going to the inserted table i.e. all the updated and new records are present in the ***inserted*** table. On the other hand, all the old values are present in the ***deleted*** table.

### **Syntax for Trigger**

**CREATE TRIGGER** [schema\_name.]trigger\_name

**ON** table\_name

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

**[NOT FOR REPLICATION]**

**AS**

**{**sql\_statements**}**

**Syntax:**

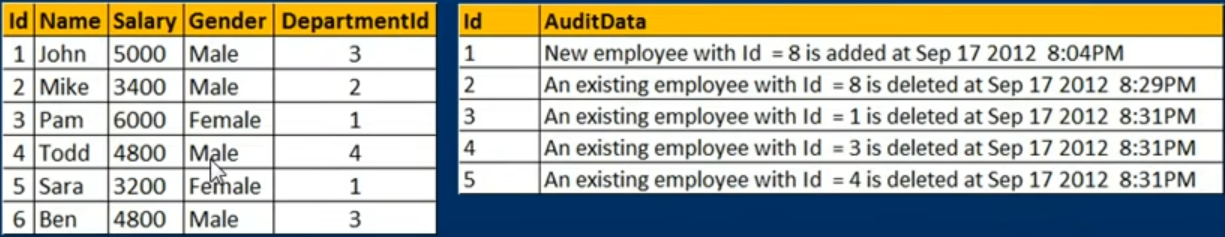
* The ***schema\_name*** is the name of the schema to which the new trigger belongs. The schema name is optional.
* The ***trigger\_name*** is the user-defined name for the new trigger.
* The ***table\_name*** is the table to which the trigger applies.
* The event is listed in the AFTER clause. The event could be INSERT, UPDATE, or **DELETE**. A single trigger can fire in response to one or more actions against the table.
* The **NOT FOR REPLICATION** option instructs SQL Server not to fire the trigger when data modification is made as part of a replication process.
* The ***sql\_statements*** is one or more Transact-SQL used to carry out actions once an event occurs.

**The content of the INSERTED and DELETED tables (Virtual Table) before and after each event:**

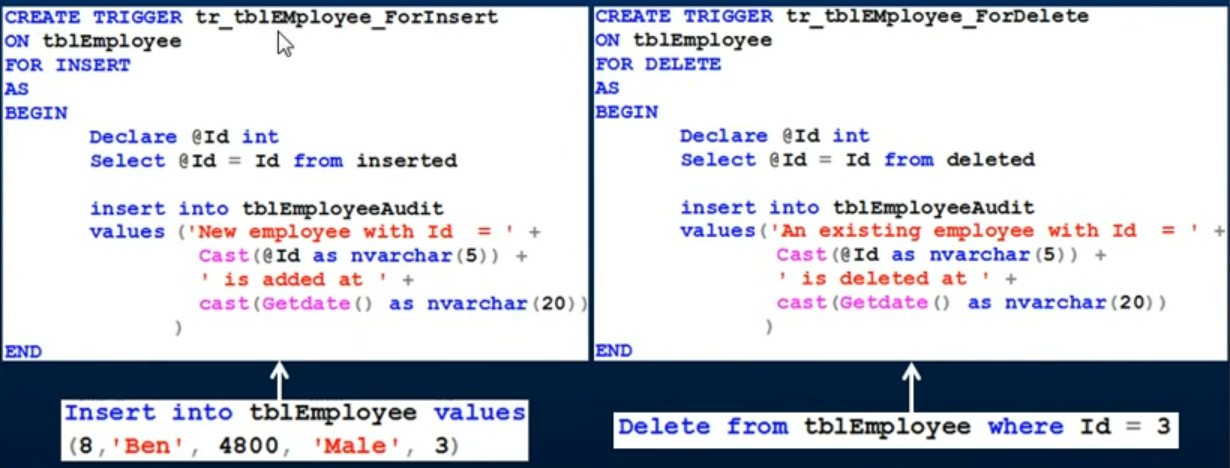
|  |  |  |
| --- | --- | --- |
| DML event | INSERTED table holds | DELETED table holds |
| INSERT | rows to be inserted | empty |
| UPDATE | new rows modified by the update | existing rows modified by the update |
| DELETE | empty | rows to be deleted |

### **Practical**

1. **Suppose, we add/remove a new employee in Employee tables, we want to some audit information to be captured automatically in Audit Table.**



1. **We want to capture employee id with date and time of data inserted or deleted.**

****

* Naming convention to use at starting ‘***tr’***.
* We create a trigger for a table and for a specific event (like insert, etc).
* For example, we want to create a trigger on this table ***tbl\_employee*** for insert event.
* Because as soon as a new row is inserted into ***tbl\_employee*** we want to log the audit information.

**I want to capture the information and we can also make use of store procedures to do this task.**

* However, the easiest way to achieve that by associate an ***After Update trigger*** with the ***tbl\_Employee*** as soon as the insert statement is fired.
* I want this after insert trigger to be fired which will capture the ID of the record that you have inserted into the table and then the date and time, and insert another row into this ***TBL employee audit table.***

-- Create After trigger to insert

alter trigger tr\_tbl\_EmployeeDetails\_ForInsert

on tblEmployeeDetails

for insert

as

BEGIN

-- check id from inserted table and store in id variable

Declare @Id int

select @Id=id from inserted

-- insert the record into audit table

insert into tbl\_employeeAuditTable values ( 'New Employee with id: ' +

CAST(@Id as nvarchar(5)) +

' is added at ' +

cast(getdate() as nvarchar(max))

)

END

-- insert data in employee table and check audit table

insert into tblEmployeeDetails (Name,salary,Gender,DepartmentId)values ('Riya',6000,'Female',1)

Select \* from tbl\_employeeAuditTable

go

--Prerequisite

create database Test

go

use test

Create TABLE tblEmployeeDetails ( Id int IDENTITY(1,1) primary key ,

[Name] nvarchar(20),

salary int,

Gender nvarchar(10),

DepartmentId int

)

-- drop table tblEmployeeDetails

-- ALTER TABLE tblEmployeeDetails ADD NewColumn INT IDENTITY(1,1)

--insert into tblEmployeeDetails (Name,salary,Gender,DepartmentId)values ('shubham',5000,'Male',3),

('shivam',null,'Male',2),

('shivani',4000,'Female',3),

('Nikhil',5100,'Male',4),

('Hemant',3500,'Male',1),

('Rahul',3500,'Male',3)

select \* from test..tblEmployeeDetails

Create table tbl\_employeeAuditTable ( id int primary key identity(1,1),

AuditData nvarchar(max)

)

Select \* from tbl\_employeeAuditTable

go

-- #2. Create After Delete trigger

alter trigger tr\_tbl\_EmployeeDetails\_ForDelete

on tblEmployeeDetails

for delete

as

BEGIN

-- check id from inserted table and store in id variable

Declare @Id int

select @Id=id from deleted

-- insert the record into audit table

insert into tbl\_employeeAuditTable values ( 'New Employee with id: ' +

CAST(@Id as nvarchar(5)) +

' is removed at ' +

cast(getdate() as nvarchar(max))

)

END

-- delete data in employee table and check audit table

delete from tblEmployeeDetails where id=6

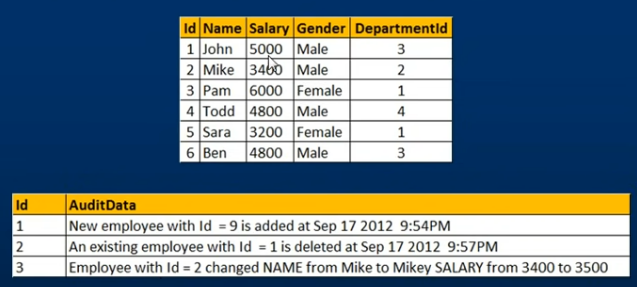
select \* from test..tblEmployeeDetails

Select \* from tbl\_employeeAuditTable

go

## **After Update Trigger**

**SQL Server provides two virtual tables that are available specifically for triggers called *INSERTED* and *DELETED* tables.**

**SQL Server uses these tables to capture the data of the modified row before and after the event occurs.**

* The After trigger for UPDATE event, makes use of both inserted and deleted tables. The inserted table contains the updated data and the deleted table contains the old data.

### Practical

--Prerequisite

create database Test

go

use test

Create TABLE tblEmployeeDetails ( Id int IDENTITY(1,1) primary key ,

[Name] nvarchar(20),

salary int,

Gender nvarchar(10),

DepartmentId int

)

-- drop table tblEmployeeDetails

-- ALTER TABLE tblEmployeeDetails ADD NewColumn INT IDENTITY(1,1)

--insert into tblEmployeeDetails (Name,salary,Gender,DepartmentId)values ('shubham',5000,'Male',3),

('shivam',null,'Male',2),

('shivani',4000,'Female',3),

('Nikhil',5100,'Male',4),

('Hemant',3500,'Male',1),

('Rahul',3500,'Male',3)

select \* from test..tblEmployeeDetails

Create table tbl\_employeeAuditTable ( id int primary key identity(1,1),

AuditData nvarchar(max)

)

Select \* from tbl\_employeeAuditTable

go

-- Create/Alter a update trigger

Create trigger tr\_tblEmployee\_ForUpdate

on tblEmployeeDetails

for update

as

Begin

-- declare varable to store old and new data

Declare @Id int

Declare @OldName nvarchar(20), @NewName nvarchar(20)

Declare @OldSalary nvarchar(20), @NewSalary nvarchar(20)

Declare @OldGender nvarchar(20), @NewGender nvarchar(20)

Declare @OldDeptId nvarchar(20), @NewDeptId nvarchar(20)

-- create a string for Audit Table dynamically

Declare @AuditString nvarchar(1000)

-- insert row from inserted virtual table to Temp table

Select \*

into #TempTable

from inserted

-- while loop is used in case if user update multiple record at a time

-- if select query returns no record, exists functiion return false and while loop get skiped.

While(Exists(Select Id from #TempTable))

Begin

set @AuditString = ''

-- set variable for inserted row form temp table

Select top 1 @Id=Id, @NewName = Name,

@NewGender = Gender, @NewSalary = Salary,

@NewDeptId = DepartmentId

from #TempTable

-- fetch record based on @id variable from deleted and set old data into variables

Select @OldName = Name,

@OldGender = Gender, @OldSalary = Salary,

@OldDeptid = DepartmentId

from deleted where id = @id

Set @AuditString = 'Employee with Id = ' + Cast(@id as nvarchar(4)) + ' changed '

if(@OldName <> @NewName)

set @AuditString = @AuditString + ' Name From ' + @OldName + ' to ' + @NewName

if(@OldSalary <> @NewSalary)

set @AuditString = @AuditString + ' Salary From ' + @OldSalary + ' to ' + @NewSalary

if(@OldGender <> @NewGender)

set @AuditString = @AuditString + ' Gender From ' + @OldGender + ' to ' + @NewGender

if(@OldDeptId <> @NewDeptId)

set @AuditString = @AuditString + ' DeptId From ' + @OldDeptId + ' to ' + @NewDeptId

-- insert the @AuditString into Audit Table

insert into tbl\_employeeAuditTable (AuditData) values (@AuditString)

-- delete the variable @id record from temp data, to prevent infinite loop and in case of multiple update scenario.

delete from #TempTable where Id = @Id

End

End

go

-- update and check the audit data

Update tblEmployeeDetails set Name='Ramesh', salary=5500, Gender='Male' where Id=5

Select \* from tbl\_employeeAuditTable

-- drop trigger tr\_tblEmployee\_ForUpdate

## **Instead of Trigger**

* An ***INSTEAD OF*** trigger is a trigger that allows you to skip an *INSERT*, *DELETE*, or *UPDATE* statement to a table or a view and execute other statements defined in the trigger instead.
* The actual insert, delete, or update operation does not occur at all.
* In other words, an INSTEAD OF trigger skips a DML statement and execute other statements.
* **Instead of Triggers usually used to update views correctly that are based on multiple tables.**

### **Syntax of Instead Trigger**

**CREATE TRIGGER [**schema\_name.] trigger\_name

**ON {**table\_name | view\_**name }**

**INSTEAD OF {[*INSERT*] [,] [*UPDATE*] [,] [*DELETE*]}**

**AS**

**{**sql\_statements**}**

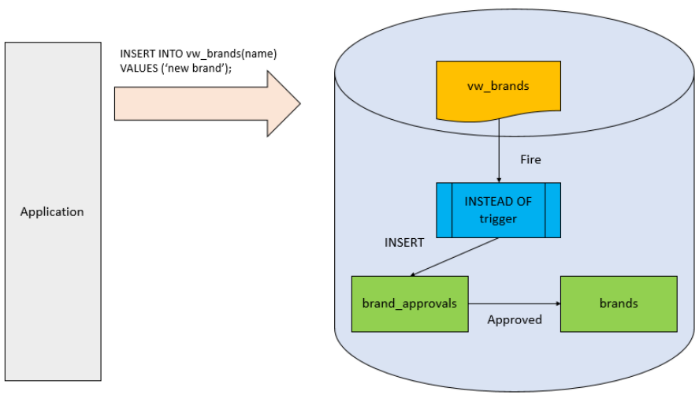
**In this syntax:**

* First, specify the name of the trigger and optionally the name of the schema to which the trigger belongs in the **CREATE TRIGGER** clause.
* Second, specify the name of the table or view which the trigger associated with.
* Third, specify an event such as ***INSERT, DELETE, or UPDATE*** which the trigger will fire in the **INSTEAD OF** clause. The trigger may be called to respond to one or multiple events.
* Fourth, place the trigger body after the **AS** keyword. A trigger’s body may consist of one or more Transact-SQL statements.

**Example**

Suppose, an application needs to insert new brands into the ***production.brands*** table. However, the new brands should be stored in another table called ***production.brand\_approvals*** for approval before inserting into the ***production.brands*** table.

|  |  |
| --- | --- |
| Trigger | INSERTED or DELETED? |
| Instead of Insert | **DELETED table is always empty and the INSERTED table contains the newly inserted data.** |
| Instead of Delete | **INSERTED table is always empty and the DELETED table contains the rows deleted** |
| Instead of Update | **DELETED table contains OLD data (before update), and inserted table contains NEW data(Updated data)** |

To accomplish this, you create a view called ***production.vw\_brands*** for the application to insert new brands. If brands are inserted into the view, an INSTEAD OF trigger will be fired to insert brands into the ***production.brand\_approvals*** table.

### **INSTEAD OF Insert**

* An ***INSTEAD OF*** trigger is a trigger that allows you to skip an [INSERT](https://www.sqlservertutorial.net/sql-server-basics/sql-server-insert/), [DELETE](https://www.sqlservertutorial.net/sql-server-basics/sql-server-delete/), or [UPDATE](https://www.sqlservertutorial.net/sql-server-basics/sql-server-update/) statement to a table or a view and execute other statements defined in the trigger instead.
* The actual insert, delete, or update operation does not occur at all.
* In other words, an INSTEAD OF trigger skips a DML statement and execute other statements.

#### **Syntax**

CREATE TRIGGER [*schema\_name*].TriggerName

ON [TableName | ViewName]

INSTEAD OF UPDATE

AS

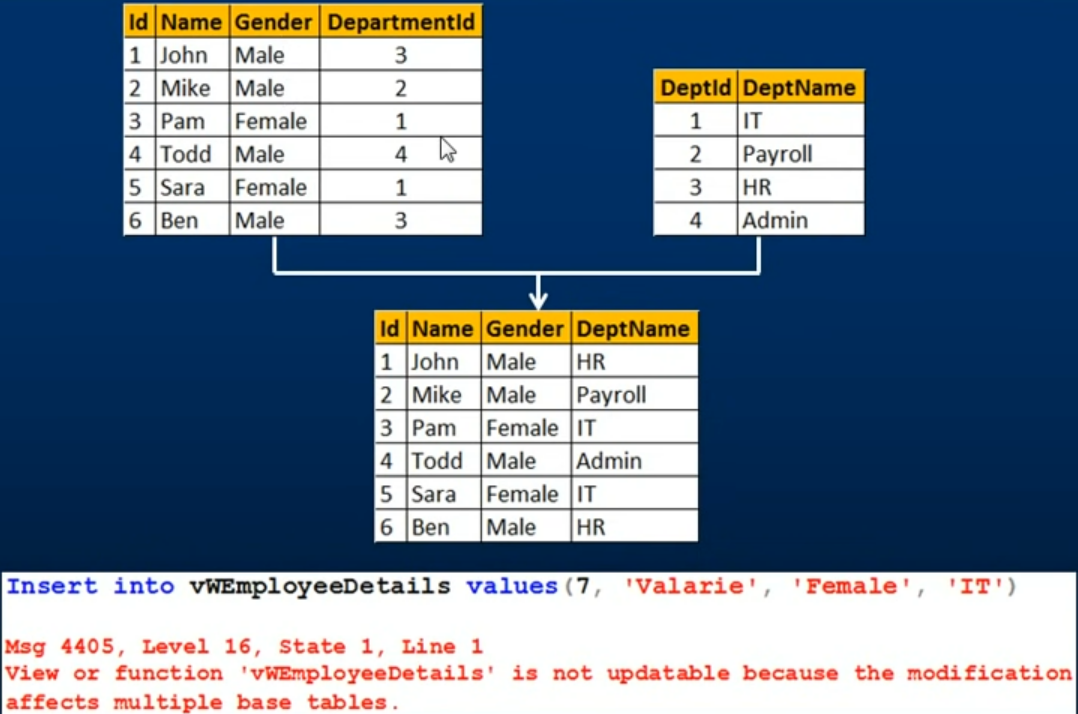
BEGIN

**-- Custom logic for the trigger**

END

**In this syntax:**

* First, specify the name of the trigger and optionally the name of the schema to which the trigger belongs in the CREATE TRIGGER clause.
* Second, specify the name of the table or view which the trigger associated with.
* Third, specify an event such as INSERT, DELETE, or UPDATE which the trigger will fire in the INSTEAD OF clause. The trigger may be called to respond to one or multiple events.
* Fourth, place the trigger body after the AS keyword. A trigger’s body may consist of one or more Transact-SQL statements.

**We have two tables here, *Tbl\_Employees* and *tbl\_Department.***

* Now we create a view based on these two tables so we want Id, Name, Gender from tbl\_employee table and ***DepartName*** from ***Tbl\_department*** table.
* So, to create a view we need to join these two table, So, this view is based on multiple base tables.
* As we know **View is a virtual table meaning it is nothing more than stored SQL Query, it doesn’t really contain any data.**
* The view actually gets its data from its underlying based tables.
* Now, when we try to insert a row into this View, behind the scenes insert that row into these base tables.
* SQL server has confusion which column should go into which table or base table should be this row should be inserted because we already have a row with ID, so should another row should be inserted.
* That why, if insert statement affecting multiple base tables, SQL server will throw an error stating so.
* This view is based on multiple tables and your insert statement into this view will affect both these tables.
* If we look at the error message view of function view employee details is not updatable because the modification affects multiple base tables.

**Instead of triggers are used to update/insert in VIEW correctly that are based on multiple base tables.**

--Prerequisite

create database Test

go

use test

go

-- create Employee Table

Create TABLE tblEmployeeDetails ( Id int primary key ,

[Name] nvarchar(20),

salary int,

Gender nvarchar(10),

DepartmentId int

)

--insert into tblEmployeeDetails (Id,Name,salary,Gender,DepartmentId)values (1,'shubham',5000,'Male',3),

(2,'shivam',1000,'Male',2),

(3,'shivani',4000,'Female',3),

(4,'Nikhil',5100,'Male',4),

(5,'Hemant',3500,'Male',1),

(6, 'Rahul',3500,'Male',3)

truncate table tblEmployeeDetails

-- create department table

Create TABLE tblDepartment( DeptId int IDENTITY(1,1) primary key ,

DeptName nvarchar(50)

)

insert into tblDepartment(DeptName) values ('IT'),('Payroll'),('HR'),('Admin')

-- create a view

Create view vw\_EmployeeDetails

AS

Select Id,Name, Gender, de.DeptName

from tblEmployeeDetails em inner join tblDepartment de on em.DepartmentId = de.DeptId

go

-- select view

select \* from vw\_EmployeeDetails

-- insert a row in view

Insert into vw\_EmployeeDetails (Id,Name, Gender, DeptName)values ((select max(id) + 1 from vw\_EmployeeDetails),'Vari','Female', 'IT')

Insert into vw\_EmployeeDetails (Id,Name, Gender, DeptName)values (50,'Vari','Female', 'I')

-- Error : View or function 'vw\_EmployeeDetails' is not updatable because the modification affects multiple base tables.

go

-- create instead trigger for insert

Create trigger tr\_vwEmployeeDetails\_InsteadofInsert

on vw\_EmployeeDetails

Instead of Insert

AS

Begin

-- create a variable which hold the deptid

Declare @DeptId int

--Check if there is a valid DepartmentId

--for the given DepartmentName

Select @DeptId = DeptId

from tblDepartment

join inserted

on inserted.DeptName = tblDepartment.DeptName

--If DepartmentId is null throw an error

--and stop processing

if (@DeptId is null)

Begin

Raiserror('Invalid Department Name. Statement terminated', 16, 1)

return

End

--Finally insert into tblEmployee table

Insert into tblEmployeeDetails(Id, Name, Gender, DepartmentId)

Select Id, Name, Gender, @DeptId

from inserted

end

### **INSTEAD OF Update**

**An INSTEAD OF UPDATE trigger is fired instead of the default update operation on a table or view. It allows you to define custom logic to be executed instead of the actual update operation.**

#### **Scenario**

**Consider an "Employees" table in a human resources database that stores employee information, including employee ID, name, department, and salary.**

* The table has a unique constraint on the employee ID column, ensuring each employee has a distinct identifier.

**In this scenario, you can use an INSTEAD OF UPDATE trigger to enforce a business rule that restricts salary updates for certain employees based on their role or department.**

* The trigger will allow updates to other employee information but prevent changes to the salary for specific cases.

**By using the INSTEAD OF UPDATE trigger, you can define custom logic to handle the update operation and enforce the** salary **restriction.**

* The trigger can check the role or department of the employee being updated and either allow or block the salary change accordingly.

#### **Syntax**

CREATE TRIGGER TriggerName

ON [TableName | ViewName]

INSTEAD OF UPDATE

AS

BEGIN

**-- Custom logic for the trigger**

END

#### **Update Function**

**The *UPDATE*() function is used within an INSTEAD OF trigger to determine which columns of a table have been updated as part of an update operation.**

* It returns a bit value (0 or 1) for each column indicating whether it was included in the update statement or not.

#### **Practical**

/\*

An INSTEAD OF UPDATE trigger is fired instead of the default update operation on a table or view.

It allows you to define custom logic to be executed instead of the actual update operation.

\*/

--Prerequisite

-- create database

IF NOT EXISTS (SELECT name FROM sys.databases WHERE name = 'KudvenkatTutorial')

BEGIN

CREATE DATABASE KudvenkatTutorial;

END

-- select database

Use KudvenkatTutorial

-- Create table

CREATE TABLE tblEmployee

(

Id int Primary Key, Name nvarchar(30), Gender nvarchar(10), DepartmentId int

)

CREATE TABLE tblDepartment

(

DeptId int Primary Key, DeptName nvarchar(20)

)

-- add data to tables

--Insert data into tblDepartment table

Insert into tblDepartment values (1,'IT'), (2,'Payroll'), (3,'HR'), (4,'Admin')

--Insert data into tblEmployee table

Insert into tblEmployee values (1,'John', 'Male', 3),

(2,'Mike', 'Male', 2),(3,'Pam', 'Female', 1),

(4,'Todd', 'Male', 4),(5,'Sara', 'Female', 1),(6,'Ben', 'Male', 3)

go

--Script to create the view:

Create view vWEmployeeDetails

as

Select Id, Name, Gender, DeptName from tblEmployee join tblDepartment

on tblEmployee.DepartmentId = tblDepartment.DeptId

go

Select \* from vWEmployeeDetails

-- update view data

Update vWEmployeeDetails set Name = 'Johny', DeptName = 'IT' where Id = 1

/\*

In Part 45, we tried to insert a row into the view, and we got an error stating -

'View or function vWEmployeeDetails is not updatable because the modification affects multiple base tables.'

Now, while updating the view, in such a way that, it affects,

both the underlying tables, and see, if we get the same error.

The following UPDATE statement changes Name column from tblEmployee and DeptName column from tblDepartment.

So, when we execute this query, we get the same error.

\*/

/\*

-- Conclusion

If a view is based on multiple tables, and if you update the view, the UPDATE may not always work as expected.

To correctly update the underlying base tables, thru a view, INSTEAD OF UPDATE TRIGGER can be used.

\*/

go

Create Trigger tr\_vWEmployeeDetails\_InsteadOfUpdate

on vWEmployeeDetails

instead of update

as

Begin

-- if EmployeeId is updated

if(Update(Id))

Begin

Raiserror('Id cannot be changed', 16, 1)

Return

End

-- If DeptName is updated

if(Update(DeptName))

Begin

Declare @DeptId int

Select @DeptId = DeptId

from tblDepartment

join inserted

on inserted.DeptName = tblDepartment.DeptName

if(@DeptId is NULL )

Begin

Raiserror('Invalid Department Name', 16, 1)

Return

End

Update tblEmployee set DepartmentId = @DeptId

from inserted

join tblEmployee

on tblEmployee.Id = inserted.id

End

-- If gender is updated

if(Update(Gender))

Begin

Update tblEmployee set Gender = inserted.Gender

from inserted

join tblEmployee

on tblEmployee.Id = inserted.id

End

-- If Name is updated

if(Update(Name))

Begin

Update tblEmployee set Name = inserted.Name

from inserted

join tblEmployee

on tblEmployee.Id = inserted.id

End

End

Go

--Now, let's try to update JOHN's Department to IT.

Update vWEmployeeDetails

set DeptName = 'IT'

where Id = 1

/\* The UPDATE query works as expected.

The INSTEAD OF UPDATE trigger, correctly updates, JOHN's DepartmentId to 1, in tblEmployee table.

\*/

--Now, let's try to update Name, Gender and DeptName. The UPDATE query, works as expected, without raising the error - 'View or function vWEmployeeDetails is not updatable because the modification affects multiple base tables.'

Update vWEmployeeDetails

set Name = 'Johny', Gender = 'Female', DeptName = 'IT'

where Id = 1

### **INSTEAD OF Delete**

**The INSTEAD OF DELETE trigger is used to replace the default behaviour of a delete operation on a table or view.**

* It allows you to define custom logic to be executed instead of the actual delete operation.
* **INSERTED table is always empty and the DELETED table contains the rows deleted**

#### **Syntax**

CREATE TRIGGER TriggerName

ON [TableName | ViewName]

INSTEAD OF Delete

AS

BEGIN

**-- Custom logic for the trigger**

END

#### **Use Cases for INSTEAD OF DELETE Triggers**

**The *INSTEAD OF DELETE* trigger is typically used in scenarios where you want to customize the behaviour of a delete operation on a table or a view.**

1. **Soft Deletes**: Instead of permanently deleting records from a table, you can use an INSTEAD OF DELETE trigger to move the deleted records to a separate "deleted" table.

* This allows you to keep a historical record of deleted data for auditing or recovery purposes.

1. **Cascading Deletes**: If you have a table with foreign key relationships to other tables, you can use an INSTEAD OF DELETE trigger to perform cascading deletes manually.

* This gives you greater control over the deletion process, allowing you to delete related records in other tables or handle the deletion based on certain conditions.

1. **Complex Delete Logic**: In certain scenarios, you might have complex business rules or conditions that need to be applied before deleting records.

* An INSTEAD OF DELETE trigger enables you to implement custom logic to enforce those rules and determine whether or not the delete operation should proceed.

1. **Logging or Auditing**: An INSTEAD OF DELETE trigger can be used to capture information about the delete operation, such as the user who initiated the deletion, the date and time of the deletion, or any additional metadata.

* This can be useful for tracking changes, maintaining an audit trail, or generating reports.

1. **Data Transformation:** If you need to transform or restructure data during the delete operation, an INSTEAD OF DELETE trigger can be used to apply the necessary transformations before performing the delete action.

#### **Practical**

Select \* from vWEmployeeDetails

go

--Now, let's try to delete a row from the view, and we get the same error.

Delete from vWEmployeeDetails where Id = 1

/\*

Error

Msg 4405, Level 16, State 1, Line 64

View or function 'vWEmployeeDetails' is not updatable because the modification affects multiple base tables.

\*/

Go

--Script to create INSTEAD OF DELETE trigger:

Create Trigger tr\_vWEmployeeDetails\_InsteadOfDelete

on vWEmployeeDetails

instead of delete

as

Begin

Delete tblEmployee

from tblEmployee

join deleted

on tblEmployee.Id = deleted.Id

--Subquery

--Delete from tblEmployee

--where Id in (Select Id from deleted)

End

go

/\*

Notice that, the trigger tr\_vWEmployeeDetails\_InsteadOfDelete, makes use of DELETED table.

DELETED table contains all the rows, that we tried to DELETE from the view.

So, we are joining the DELETED table with tblEmployee, to delete the rows.

You can also use sub-queries to do the same. In most cases JOINs are faster than SUB-QUERIEs.

However, in cases, where you only need a subset of records from a table that you are joining with, sub-queries can be faster.

\*/

# **Derived Tables**

**A derived table is a temporary table that is created as a result of a subquery within a larger query.**

* It is not stored as a separate table in the database but exists only for the duration of the query execution.
* Derived tables are often used when you need to perform complex calculations or filtering on intermediate results before further processing.

**Here's an example to illustrate the usage of a derived table:**

**Suppose you have two tables: "Orders" and "Customers".**

* The "***Orders***" table contains information about customer orders, including the order ID, customer ID, and order amount.
* The "***Customers***" table contains information about the customers, including the customer ID and customer name.

**SELECT**

**C.customer\_name,**

**DT.total\_order\_amount**

**FROM**

**(**

**-- derived table**

**SELECT**

**customer\_id,**

**SUM(order\_amount) AS total\_order\_amount**

**FROM**

**Orders**

**GROUP BY**

**customer\_id**

**HAVING**

**SUM(order\_amount) >= 100**

**) AS DT**

**INNER JOIN Customers AS C ON C.customer\_id = DT.customer\_id;**

**In this example, the derived table (DT) is created by performing a subquery on the "Orders" table.**

* The subquery calculates the total order amount for each customer using the SUM function and groups the results by the customer ID.
* The HAVING clause filters out any customers whose total order amount is less than $100.
* The outer query then joins the derived table with the "Customers" table using the customer ID to retrieve the customer name and the total order amount.

**The derived table allows you to perform the necessary calculations and filtering on the intermediate results before presenting the final result set.**

* It helps in organizing complex queries and makes the code more readable and manageable.

## **USECASE**

* Perform intermediate calculations within a query.
* Apply filtering conditions to subset data.
* Calculate aggregates (e.g., sums, counts, averages) on subsets of data.
* Perform data transformations before joining with other tables.
* Simplify complex join conditions.
* Break down complex queries into smaller, more manageable steps.
* Improve code readability and maintainability

## **View vs. Derived Table vs. CTE**

### **Views**

* **Saved queries that behave like virtual tables.**
* **Persist in the database and can be accessed by multiple users.**
* **Have a defined structure and can include multiple columns and rows.**
* **Can be modified, updated, or deleted separately from the query that created them.**
* **Can be reused across multiple queries.**
* **Provide a way to abstract complex queries and simplify the overall query structure.**

### **Derived Tables**

* **Temporary tables created as a result of a subquery within a larger query.**
* **Exist only for the duration of that query and are specific to it.**
* **Used for intermediate calculations, filtering, or transformations within a single query.**
* **Do not persist in the database and cannot be referenced or used outside the query.**
* **Do not have a predefined structure and their structure is defined by the subquery.**

### **CTEs (Common Table Expressions)**

* **Named temporary result sets within a query.**
* **Defined using the WITH keyword and can be referenced multiple times in the same query.**
* **Provide a way to break down complex queries into smaller, more readable steps.**
* **Can be used for recursive queries.**
* **Can be considered as a way to create temporary views within a query.**

# **CTE (Common Table Expressions)**

**A Common Table Expression (CTE) is a named temporary result set that you can use within a query.**

* It allows you to define subqueries and reference them multiple times within the same query.
* CTEs are defined using the WITH keyword followed by the name of the CTE and the query that defines it.

## **Syntax:**

**WITH CTE\_Name (column1, column2, ..., columnN)**

**AS (**

**SELECT ...**

**FROM ...**

**WHERE ...**

**)**

1. **WITH Clause**: The query begins with the WITH clause, which introduces the CTE and specifies its name.

* This is followed by a list of column names (optional) that defines the structure of the CTE.

1. **CTE\_Name:** The name given to the CTE, which is used to reference it later within the same query. The name should be unique within the query.
2. **Subquery:** The subquery within the parentheses defines the result set for the CTE.

* It can be any valid SQL query, including joins, aggregations, filtering conditions, or even another CTE.

1. **Usage in the Query**: Once a CTE is defined, it can be referenced like a table in the subsequent parts of the query.

* For example, you can include it in the ***SELECT, FROM, JOIN, or WHERE clauses***.

## **Benefits**

1. **Improved Code Readability**: CTEs allow you to break down complex queries into smaller, self-contained steps.

* Each CTE represents a logical unit of the query, making the overall code more readable and easier to understand.
* By giving meaningful names to CTEs, you can provide context and clarity to the different parts of the query.

1. **Code Reusability**: CTEs can be referenced multiple times within the same query.

* This eliminates the need to repeat complex subquery logic, reducing the chances of errors and improving query efficiency.
* By defining a CTE once, you can reuse it across multiple parts of the query, simplifying the code and enhancing maintainability.

1. **Recursive Queries**: CTEs are particularly useful for performing recursive operations on hierarchical or graph-based data structures.

* They allow a query to refer to its own CTE in subsequent iterations, enabling the processing of hierarchical relationships, such as organizational structures or network data.
* Recursive CTEs make it easier to handle complex tree-like structures and traverse them efficiently.

1. **Simplified Debugging**: CTEs enable you to isolate and test subqueries separately.

* This can be beneficial when troubleshooting complex queries, as you can focus on specific parts of the query and identify any issues more easily.
* By breaking down the query into smaller logical units, it becomes simpler to understand and pinpoint errors or unexpected results.

1. **Optimization Opportunities**: CTEs provide optimization opportunities for the query optimizer.

* Since CTEs act as temporary named result sets, the optimizer can make better decisions on how to efficiently execute the query.
* It can optimize access paths, join operations, and filter conditions based on the structure and characteristics of the CTE.

## **Disadvantages**

1. **Limited Scope**: CTEs are only visible within the query in which they are defined.

* They cannot be referenced or used outside of the query, limiting their scope and reuse potential.

1. **Execution Plan Complexity**: Complex CTEs may result in more intricate execution plans, which can impact query performance.

* Care should be taken to ensure efficient execution plans.

1. **Memory Usage**: CTEs reside in memory while the query is executing, which may affect memory consumption, particularly for large or recursive CTEs.
2. **Lack of Persistence**: CTEs are temporary and do not persist like views or tables.

* They are recalculated each time the query is executed, which may impact performance for queries with frequent execution.

## Use Case

**Complex Data Transformations**:

If you have a query that involves complex calculations or data transformations, using CTEs can break down the steps and make the code more readable.

* For example, if you need to calculate intermediate results before joining or aggregating data, CTEs can simplify the query structure.

**WITH CTE\_TransformedData AS (**

**SELECT**

**column1,**

**column2,**

**-- perform complex calculations or transformations here**

**FROM**

**source\_table**

**)**

**SELECT**

**column1,**

**SUM(column2) AS total**

**FROM**

**CTE\_TransformedData**

**GROUP BY**

**column1;**

**Recursive Operations**:

If you are working with hierarchical or recursive data structures, CTEs are particularly useful.

* They allow you to traverse the hierarchy and perform iterative operations. For example, when working with organizational charts or tree-like structures.

**WITH RECURSIVE CTE\_Hierarchy AS (**

**SELECT**

**employee\_id,**

**employee\_name,**

**manager\_id,**

**1 AS level**

**FROM**

**employees**

**WHERE**

**employee\_id = 1**

**UNION ALL**

**SELECT**

**e.employee\_id,**

**e.employee\_name,**

**e.manager\_id,**

**level + 1**

**FROM**

**employees e**

**JOIN**

**CTE\_Hierarchy c ON c.employee\_id = e.manager\_id**

**)**

**SELECT**

**employee\_id,**

**employee\_name,**

**level**

**FROM**

**CTE\_Hierarchy;**

### **Do not use CTEs when:**

**Simple Queries**:

For straightforward queries that do not involve complex transformations or recursive operations, using CTEs may add unnecessary complexity.

* It's often more straightforward to write the query without CTEs.

SELECT

column1,

column2

FROM

source\_table

WHERE

column3 = 'value';

**Single-Use Subqueries:**

If you have a simple subquery that is used only once in the main query, it may not be necessary to use a CTE.

* Instead, you can directly incorporate the subquery within the main query.

SELECT

column1,

(

SELECT

MAX(column2)

FROM

sub\_table

WHERE

sub\_table.column1 = main\_table.column1

) AS max\_value

FROM

main\_table;

## **Practical**

# **Optimization Techniques**

**There are several optimization techniques that can be applied to SQL queries to improve their performance.**

**Here are some commonly used techniques:**

1. **Indexing**: Properly indexing the tables involved in the query can significantly speed up query execution.

* Indexes provide a quick lookup mechanism for retrieving data and can greatly reduce the number of disk I/O operations.

1. **Query Rewriting**: Rewriting the query itself to use more efficient constructs or rearranging the order of joins and filters can often lead to better performance.

* Understanding the query execution plan and identifying potential bottlenecks is crucial for effective query rewriting.

1. **Filtering and Joining**: Applying filters and joins in an optimal order can improve query performance.

* Filtering early in the query execution process reduces the number of rows processed in subsequent steps, improving overall performance.

1. **Subqueries and CTEs (Common Table Expressions):** Using subqueries or CTEs instead of complex joins or temporary tables can sometimes improve performance.

* They allow for better query structuring and can help the query optimizer make better decisions.

1. **Denormalization:** In some cases, denormalizing the database schema can improve query performance.

* By duplicating data and reducing the need for complex joins, queries can execute faster.
* However, denormalization should be carefully considered, as it can introduce data integrity and maintenance challenges.

1. **Query Optimization Tools**: Many database management systems (DBMS) provide query optimization tools and hints that allow you to influence the query execution plan.

* These tools provide insights into how the query optimizer evaluates and executes queries, enabling you to make informed decisions for optimization.

1. **Partitioning:** Partitioning involves splitting large tables into smaller, more manageable pieces based on a specific criterion, such as ranges of values.

* Partitioning can improve query performance by reducing the amount of data accessed during query execution.

1. **Query Caching**: Caching query results can significantly improve performance for repetitive queries.

* Caching mechanisms store the results of a query in memory, allowing subsequent identical queries to be served from the cache rather than executing the entire query.

1. **Data Archiving and Purging**: Removing unnecessary data and archiving older data that is not frequently accessed can reduce the size of the tables, leading to improved query performance.

* Archiving data that is seldom accessed can be moved to separate tables or even different databases.

1. **Hardware Considerations:** Upgrading hardware components such as CPUs, memory, and storage can have a significant impact on query performance.

* Increasing memory allocation for the database server, utilizing faster storage devices, or adding additional CPU cores can speed up query execution.