TRIGGER 17

Trigger'lar (*tetikleyiciler*), veri ya da sistemle ilgili değişimlerde otomatik olarak tetiklenen Stored Procedure'lerdir. Trigger'ların Stored Procedure'lerden farkı; dışarıdan parametre almaması, dışarıya parametre göndermemesi ve bir kullanıcı tarafından değil, bir olay tarafından tetiklenmesidir.

Trigger'ları Anlamak

SQL Server'da verilerin yönetiminde neredeyse her şeyi geliştirici belirler ve yönetir. Bu şekilde verileri değiştirmek, silmek ve yeni veriler eklemek gibi işlemler gerçekleştirilir. Ancak bazen belirli işlemlerin otomatik olarak gerçekleştirilmesi gerekir. Örneğin; bir veri eklendiğinde, aynı ya da farklı bir tabloda, başka bir sütunun değerini değiştirmek gerekebilir. Bu tür durumlarda SQL Server'ın geliştirici tarafından çalıştırılan nesneler yerine, olay oluştuğu anda tetiklenerek otomatik olarak çalışacak nesnelere ihtiyaç duyulur. Bu tanım, nesne yönelimli programlama tecrübesi olan T-SQL geliştiricileri tarafından çabuk kavranabilecektir. Çünkü nesne yönelimli programlamada event (*olay*) kavramının veritabanı programlamadaki karşılığı trigger'lardır. Veritabanında bir işlem tarafından tetiklenmeler gerçekleştirmek için trigger'lar kullanılır.

Trigger'lar sadece **`INSERT`**, **`UPDATE`**, **`DELETE`** işlemlerinden sonra devreye girebilecek şekilde programlanabilirler. Çoklu tablo ilişkisi (**`Inner Join`**) bulunan view'lere veri ekleme, sadece trigger'lar ile mümkün olabilir. Aynı zamanda, **`INSERT`**, **`UPDATE`**, **`DELETE`** sorguları çalıştırıldığında, bu işlemlerin yerine yapılmak istenen farklı işlemler de trigger'lar ile gerçekleştirilebilir.

Trigger anlatımında sürekli işlemlerden sonra gerçekleşecek şekilde programlanabileceğinden bahsettik. Çünkü SQL Server, işlemlerden önce çalışması için kullanılan **BEFORE** Trigger özelliğini desteklemez. SQL Server'da olaydan önce devreye giren trigger'lar olarak **INSTEAD OF** kullanılabilir. **INSTEAD OF** trigger'lar, gerçek tablolar değişiklikten etkilenmeden önce devreye girerler.

NOT

Oracle eğitimlerimde **BEFORE** Trigger'ları anlatmıştım. Bu trigger'lar bir işlem gerçekleştirilmeden önce çalışırlar. SQL Server bu sorunu **INSERTED** ve **DELETED** adında iki sözde (*pseudo*) tablo ile çözmüştür. Bu tabloları ilerleyen bölümlerde inceleyeceğiz. Oracle'ın **BEFORE** Trigger'ı ile SQL Server'ın **INSTEAD OF** ve **INSERTED**, **DELETED** tablo modeli açısından sonuç anlamında pek fark yoktur. Ancak kullanılan yöntem ve mimari farklıdır.

SQL Server da trigger'lar, T-SQL ile oluşturulabildiği gibi, Management Studio ile de oluşturulabilir. SQL Server, **DML** (***D**ata **M**anipulation **L**anguage*) komutları olan **INSERT**, **UPDATE**, **DELETE**'e destek verdiği gibi, **DDL** (***D**ata **D**efination **L**anguage*) komutları olan **CREATE**, **ALTER**, **DROP** sorguları için de trigger oluşturmayı destekler. DDL trigger'lar sadece transaction'dan sonra devreye girebilir, **INSTEAD OF** olamazlar.

Trigger'lar Nasıl Çalışır?

Trigger'lar bir transaction olarak çalışırlar. Hata ile karşılaşıldığında **ROLLBACK** ile yapılan işlemler geri alınabilir. Trigger yapısının en önemli unsurlarından biri sözde tablolardır. Sözde tablolar, **INSERTED** ve **DELETED** olmak üzere iki adettir. Bu tablolar RAM üzerinde mantıksal olarak bulunurlar. Gerçek veri tablosuna veri eklendiğinde, eklenen kayıt **INSERTED** tablosuna da eklenir. Tablodan bir kayıt silindiğinde ise silinen kayıt **DELETED** tablosunda yer alır. Trigger'lar güncelleme işlemi için **UPDATED** isimli bir sözde tabloya sahip değildir. Güncellenen veriler ilk olarak silinerek **DELETE** işlemi ve sonrasında tekrar eklenerek **INSERT** işlemi gerçekleştirilir. Bir kayıt güncellendiğinde orijinal kayıt **DELETED** tablosunda, değişen kayıt ise **INSERTED** tablosunda saklanır.

ON

Trigger'ın hangi nesne için oluşturulduğunu belirtmek için kullanılır.

WITH ENCRYPTION

View ve Stored Procedure bölümlerinde anlatılan özellikler aynen geçerlidir. Bu özellik sadece View ve Stored Procedure için geçerlidir. Hangi veriler üzerinde

trigger işlemi yapıldığının bilinmemesi gereken durumlarda, yani trigger'ın yaptığı işlemlerin bilinememesi için, içeriği şifreleme amacı ile kullanılır.

WITH APPEND

SQL Server 6.5 ve öncesi versiyonlar için uyumluluk açısından desteklenmektedir. Bu eski versiyonlarda, aynı tablo üzerinde aynı işlemi gerçekleştiren birden fazla trigger oluşturulamaz. Bir tabloda veri ekleme ve güncelleme işlemi için oluşturulan bir trigger varsa, aynı tablo üzerinde bir başka güncelleme işlemi için yeni bir trigger oluşturulamaz.

Yeni versiyonlarda bu durum bir sorun değildir. Ancak geriye dönük çalıştırılması gereken veritabanlarında **WITH APPEND** özelliği, aynı tabloda aynı işlemi yapan birden fazla trigger oluşturulmasını sağlar.

NOT FOR REPLICATION

Bu özellik ile replikasyon-ilişkili görev tabloyu değiştirdiğinde, trigger başlatılmayacaktır.

AS

Trigger'in içerik, yani script kısmına geçildiğini belirtir. Bu sözcükten sonraki sorgular trigger'ın içeriğidir.

FOR | AFTER İFADESİNE VE INSTEAD OF KOŞULU

Trigger'ı, **INSERT**, **UPDATE** ve **DELETE** gibi sorguların hangisi ya da hangileri tarafından ve ne zaman başlatılacağını belirtmek için kullanılır.

TRIGGER TÜRLERİ VE INSERTED, DELETED TABLOLARI

Trigger'lar farklı ihtiyaçlara yönelik farklı türlere sahiptir. Bir trigger'ın, veri ekleme, güncelleme ya da silme işleminden sonra tetiklenmesi gerektiği durumlarda farklı seçenekler tanımlanması gerekir. Tanımlanan bu seçenekler ile trigger tam olarak ne yapacağını bilebilir.

Trigger'lar, veri ekleme, silme ve güncelleme işlemlerini gerçekleştirmek için **INSERTED** ve **DELETED** sözde tablolarına ihtiyaç duyar. İlerleyen anlatımlarda bu tablolara da değinerek çeşitli örnekler yapacağız.

INSERT Trigger

FOR INSERT ile kullanılır. Tabloya yeni bir veri eklemek istendiğinde tetiklenen trigger türüdür. Eklenen her yeni satır, trigger var olduğu sürece var olan **INSERTED** tablosuna kaydedilir. **INSERTED** tablosu trigger'lar ile bütünleşik bir kavramdır. Bir trigger'dan önce de yoktur, sonra da. Sadece trigger varken vardır.

DELETE Trigger

FOR DELETE ile kullanılır. Tablodan bir veri silme işlemi gerçekleştirilirken kullanılır. Silinen her kaydın bir kopyası **DELETED** tablosunda tutulur. Trigger ile kullanılır.

UPDATE Trigger

FOR UPDATE ile kullanılır. Tablodaki güncelleme işlemleri için tetikleme gerçekleştirmede kullanılır. Güncelleme işlemlerine özel bir **UPDATED** tablosu yoktur. Bunun yerine, güncellenen bir kayıt, tablodan silinmiş ve daha sonra tekrar eklenmiş olarak düşünülür. **INSERT** ve **DELETE** işlemlerini gerçekleştirdiği için doğal olarak **INSERTED** ve **DELETED** tablolarından erişilir. **INSERTED** ve **DELETED** tabloların satır sayıları eşittir.

Trigger Oluşturmak

SQL Server'da iki farklı türde trigger oluşturulabilir. **FOR** ve **AFTER** ile transaction'dan sonra tetiklenecek, **INSTEAD OF** ile de tablolar değişiklikten etkilenmeden önce tetiklenecek trigger'lar oluşturulabilir. Bu DML trigger'ları **INSERT**, **UPDATE**, **DELETE** işlemleri için oluşturulabilir.

Trigger'lar tüm veritabanı nesneleri gibi **CREATE** ifadesiyle oluşturulur. **CREATE TRIGGER** ifadesini çalıştırabilmek için şu veritabanı izinlerinden birine sahip olunmalıdır.

- **sysadmin** server rolü
- **db_owner** sistem rolü
- **db_dlladmin** sistem rolü

Kullanıcının sahibi olduğu view ya da tablo üzerinde herhangi bir ek izne sahip olmasına gerek yoktur.

Trigger'lar otomatik tetiklenen bir nesne olması nedeniyle içerisinde kullanılacak ifadelerde bazı kısıtlamalar olması veri güvenliği ve tutarlılığı açısından gereklidir.

Trigger'lar şu ifadeleri içeremez;

- **CREATE DATABASE**
- **ALTER DATABASE**
- **DROP DATABASE**
- **LOAD DATABASE**
- **RESTORE DATABASE**
- **LOAD LOG**
- **RESTORE LOG**
- **DISK INIT**
- **DISK RESIZE**
- **RECONFIGURE**

Söz Dizimi (DML Trigger)

```
CREATE TRIGGER trigger_ismi
ON tablo_ismi
[WITH seçenekler]
{FOR | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
AS
-- Trigger SQL gövdesi
```

Söz Dizimi (DDL Trigger)

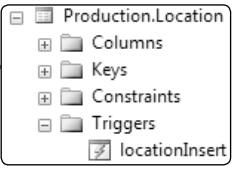
```
CREATE TRIGGER trigger_ismi
ON { DATABASE | ALL SERVER }
{ veritabani_seviyeli_olaylar | server_seviyeli_olaylar }
AS
-- Trigger SQL gövdesi
```

INSERT Trigger

Bir tabloya yeni kayıt ekledikten sonra devreye girecek işlemler için kullanılır. Trigger oluşturulduktan sonra, yeni eklenen her kaydı **Inserted** tablosunda tutmaya başlar. Bu kayıtlar gerçek tablonun yapısal bir kopyasıdır. SQL Server, satır bazlı trigger destek vermez. Her eklenen kayıt için değil, kayıt seti için çalışır.

Production.Location tablosuna yeni bir lokasyon bilgisi eklemek istendiğinde, ekleme işleminden sonra çalışacak basit bir trigger oluşturalım.

```
CREATE TRIGGER locationInsert
ON Production.Location
AFTER INSERT
AS
BEGIN
SET NOCOUNT ON;
 SELECT 'Yeni lokasyon bilgisi eklendi.';
SET NOCOUNT OFF;
END;
```



Oluşturulan trigger, **Management Studio** içerisinde, ilgili tablonun **Triggers** menüsünde görüntülenebilir.

Trigger'ın tetiklenmesi için bir kayıt ekleyelim.

```
INSERT INTO Production.Location(Name, CostRate, Availability, ModifiedDate)
VALUES('Yeni Lokasyon', 0.00, 0.00, GETDATE());
```

INSERT sorgusu ile birlikte kayıt eklenecektir. Ekleme işleminden hemen sonra da, ekranda aşağıdaki gibi bir bilgi mesajı görüntülenecektir.

	(No column name)
1	Yeni lokasyon bilgisi eklendi.

Yeni eklenen her personel için bir bildirim zorunluluğu oluşturmak istenebilir. Bunu, trigger içerisinde **RAISERROR** kullanarak gerçekleştirebiliriz. Bu örnekte, bir personel yerine ürün ya da sipariş ekleme gibi bir senaryoda söz konusu olabilirdi.

```
CREATE TRIGGER trg_PersonelHatirlatici
ON Personeller
AFTER INSERT
AS
 RAISERROR ('Eklenen Personeli Bildir', 16, 10);
```

Bu örneğimizde daha önceki örneklerde oluşturduğumuz **Personeller** tablosunu kullandık. Trigger ismi **trg_PersonelHatirlatici** olarak belirlendi.

```
Msg 50000, Level 16, State 10, Procedure trg_PersonelHatirlatici, Line 5
Eklenen Personeli Bildir

(1 row(s) affected)
```

Yukarıdaki kullanımda hata seviyesi olarak 16 verildiği için kırmızı yazı ile hata bildirimi yapılır. Ancak bu seviyeyi 10 yaparak bir uyarı bildirimi seviyesine düşürebiliriz.

```
Eklenen Personeli Bildir

(1 row(s) affected)
```

Bu örnekteki işlem, her eklenen personelin mail ile yöneticiye bildirilmesi gibi bir işi de gerçekleştirmek istiyor olabilirdi.

Bir çok durumda, yapılan işlemlerin anlık olarak otomatik loglama ihtiyacı duyulur. Bu ihtiyacı karşılamak için trigger kullanılabilir.

`Production.Product` tablosunda yapılan her veri ekleme işlemini otomatik olarak **`ProductLog`** tablosuna kaydedelim.

`ProductLog` tablosunu oluşturalım.

```
CREATE TABLE ProductLog
(
 ProductID  INT,
 Name   VARCHAR(50),
 ProductNumber NVARCHAR(25),
 ListPrice  DATETIME
);
```

`Production.Product` tablosunda anlık loglama içerisinde, bulunmasını istediğimiz sütunları barındıracak **`ProductLog`** tablosunu oluşturduk. Bu tabloda belirtilen sütunların birebir karşılığını, **`INSERTED`** isimli sözde tablodan alacağız.

```
CREATE TRIGGER trg_ProductLog
ON Production.Product
AFTER INSERT
AS
BEGIN
 INSERT INTO ProductLog SELECT ProductID, Name,
   ProductNumber, ListPrice
 FROM inserted;
END;
```

Oluşturulan trigger **Management Studio**'da, **`Production.Product`** tablosu içerisindeki **`Triggers`** bölümünde görülebilir.

Trigger'ı test etmek için veri ekleyelim.

```
INSERT INTO Production.Product
 (Name, ProductNumber, MakeFlag, FinishedGoodsFlag,
SafetyStockLevel,ReorderPoint, StandardCost, ListPrice,
 DaysToManufacture, SellStartDate,rowguid, ModifiedDate)
VALUES('Test Ürün','SK-3335', 1, 0, 500, 700, 0, 0, 3,
  GETDATE(), NEWID(), GETDATE());
```

Normalde tek veri ekleme sorgusu çalıştığında tek bir "*1 row(s) effected*" mesajı döner. Ancak bu sorgu ile birlikte, iki kez aynı metnin **Messages** ekranında görüntülendiği görülür. Bunun nedeni; ilk metin **`INSERT`** sorgusunu belirtirken, ikinci sorgu trigger tarafından tetiklenerek, **`ProductLog`** tablosuna eklenen veri için otomatik olarak oluşturulan ve kullanılan **`INSERT`** sorgusunun mesajıdır.

```
SELECT ProductID, Name, ProductNumber FROM Production.Product
WHERE ProductNumber = 'SK-3335';
```

	ProductID	Name	ProductNumber
1	1021	Test Ürün1	SK-3335

```
SELECT * FROM ProductLog;
```

	ProductID	Name	ProductNumber	ListPrice
1	1021	Test Ürün1	SK-3335	1900-01-01 00:00:00.000

Trigger'ı tetiklemek için sadece T-SQL ile değil, **Management Studio**'da kullanılabilir. Veri düzenleme ekranında yeni bir satır eklendiğinde trigger tetiklenecek ve yeni eklenen kaydı **`ProductLog`** tablosuna loglayacaktır.

Trigger'lar, veritabanı programlamaya yeni başlayanlara biraz karmaşık gelebilir. Bu nedenle anlamak ve ne tür çalışmalarda kullanılacağı kavramada zorlanılması normaldir. Trigger'lar, SQL Server'da ileri seviye bir kavramdır.

Farklı bir örnek daha yaparak **`INSERTED`** tablosunu ilişkisel olarak **`JOIN`** ile kullanalım.

Production.Product tablosuna son eklenen kaydı, **INSERTED** tablosundan okuyarak, otomatik olarak göstereceğiz.

```
CREATE TRIGGER trg_GetProduct
ON Production.Product
AFTER INSERT
AS
BEGIN
 SELECT PL.ProductID, PL.Name, PL.ProductNumber
 FROM Production.Product AS PL
 INNER JOIN INSERTED AS I
 ON PL.ProductID = I.ProductID;
END;
```

Trigger'ı tetiklemek için veri ekleyelim.

```
INSERT INTO Production.Product
 (Name, ProductNumber, MakeFlag, FinishedGoodsFlag,
SafetyStockLevel,  ReorderPoint, StandardCost,ListPrice,DaysToManufa
cture,SellStartDate,
 rowguid, ModifiedDate)
VALUES('Test Product','SK-3334', 1, 0, 500, 700, 0.00, 0.00, 3,
 GETDATE(), NEWID(), GETDATE());
```

Veri eklendiğinde, son eklenen kayıt otomatik olarak listelenecektir.

	ProductID	Name	ProductNumber
1	1033	Test Product	SK-3334

DELETE Trigger

Bir tablodan kayıt silindiğinde otomatik olarak gerçekleşmesi istenen işlemler için kullanılır. **DELETE** Trigger'ı tarafından **DELETED** isimli sözde tablo oluşturulur. **DELETED** tablosu sadece trigger içerisinde kullanılabilir. **DELETED** tablosu transaction log dosyasından türetilir. Bu nedenle silme tekniği önemlidir. **TRUNCATE** ile silme işlemini anlatırken, bu yöntemle silinen verilerin veri page'lerinden anında koparılarak silindiğini belirtmiştik. Yani, veri ile ilgili bir log tutulmaz. Bu nedenle **TRUNCATE** ile silinen veriler **DELETED** tablosu tarafından tutulmadığı için **TRUNCATE** ile silinen verilerde **DELETE** Trigger'ı beklenen işlemi gerçekleştiremez.

DELETED tablosu gerçek veri tablosundan bile silinmiş olan verileri tutar. Bunun nedeni, transaction sırasında geri alma işlemi gerçekleştirme olasılığında verilerin **ROLLBACK** ile geri alınabilmesini sağlamaktır. Hatırlarsanız, geri alma işlemleri için transaction log dosyası kullanılıyordu.

Kullanicilar isimli tabloda silinen her verinin, **INSERTED** tablosundaki kayıtları kullanılarak bir mesaj gösterelim.

```
CREATE TRIGGER trg_KullaniciSil
ON Kullanicilar
AFTER DELETE
AS
BEGIN
 SELECT deleted.KullaniciAd + ' kullanıcı adına ve ' + deleted.Email +
 ' email adresine sahip kullanıcı silindi.' FROM deleted;
END;
```

Kullanicilar tablosundaki kayıt.

	KullaniciID	KullaniciAd	Sifre	Email	Telefon
1	1	CihanOzhan	cihan.sifre	cihan.ozhan@hotmail.com	02223456789

Kullanicilar tablosundan bir kayıt silelim.

```
DELETE FROM Kullanicilar WHERE KullaniciID = 1;
```

	(No column name)
1	CihanOzhan kullanici adina ve cihan.ozhan@hotmail.com email adresine sahip kullanici silindi.

DELETE işleminden sonra verinin nasıl **SELECT** edilebildiği konusunu anlamak, **DELETE** trigger'larını anlamak için yeterlidir. Trigger içerisinde Kullanicilar tablosunu sorgulamak yerine, **DELETED** tablosunu sorguladık. Çünkü Kullanicilar tablosunda (gerçek veri tablosu) ilgili kayıt silindi. Ancak **DELETED** tablosu, silinen kayıtları transaction log dosyasını kullanarak tutar. Bu nedenle silinen kayda ulaşmak için **DELETED** tablosunu kullanabiliriz. Tabi ki **INSERTED** ile görüntülenen verinin gerçek tabloda aslının olmadığını unutmamak gerekir.

UPDATE Trigger

Tablo üstünde kayıtlarda güncelleme olduğunda tepki vermek üzere kodlanan trigger'lardır. **INSERT** ve **DELETE** trigger'lar gibi kendine has bir sözde tabloya

sahip değildir. **INSERTED** ve **DELETED** tablolarının her ikisini de kullanır. Ana tablodaki güncellenen kayıtların kopyasını **INSERTED** tablosu, güncellenmeden önceki hallerini ise **DELETED** tablosu tutar.

Gelişmiş veritabanı tasarımlarında, neredeyse tüm tablolar için güncellenme tarihi bilgisini tutacak bir sütun bulunur. Örneğin; bir ürün tablosunda, ürünün herhangi bir bilgisinin güncellenmesi durumunda, o anki sistem tarih ve zamanı ürünün güncellenme tarihi sütununa yazılır. Güncelleme işlemini yönetmek ve sorumluluk takibi açısından güncelleme yapan kullanıcının **ID** değeri de bir log tablosuna yazılır.

Production.Product tablosunda herhangi bir sütunun değeri değiştirildiğinde, **ModifiedDate** sütunu değerini o anki sistem tarih ve zaman bilgisiyle değiştirelim. Güncelleme işleminde, **ModifiedDate** sütunu **SET** edilmese bile, otomatik olarak o anki güncel bilgilerle değiştirilsin.

```
CREATE TRIGGER trg_UrunGuncellemeTarihiniGuncelle
ON Production.Product
AFTER UPDATE
AS
BEGIN
 UPDATE Production.Product
 SET ModifiedDate = GETDATE()
 WHERE ProductID = (SELECT ProductID FROM inserted)
END;
```

ProductID değeri **999** olan ürünü listeleyelim.

```
SELECT ProductID, Name, ProductNumber, ModifiedDate FROM Production.Product
WHERE ProductID = 999;
```

	ProductID	Name	ProductNumber	ModifiedDate
1	999	Road-750 Red, 52	BK-R19B-52	2013-02-12 00:27:00.983

Şimdi, bu ürünü güncelleyelim.

```
UPDATE Production.Product
SET Name = 'Road-750 Red, 52'
WHERE ProductID = 999;
```

Güncellemeden sonraki sütun değerlerini listeleyelim.

```
SELECT ProductID, Name, ProductNumber, ModifiedDate FROM Production.Product
WHERE ProductID = 999;
```

	ProductID	Name	ProductNumber	ModifiedDate
1	999	Road-750 Yellow, 52	BK-R19B-52	2013-02-13 10:23:28.923

Güncelleme işlemi sadece Name sütunu üzerinde yapılmasına rağmen **`ModifiedDate`** sütunu otomatik olarak trigger tarafından güncellendi.

Ürünler tablosundaki kayıtlarda bir güncelleme yapıldığında, bu güncelleme bilgilerini başka bir tabloda log olarak tutalım. Bu log tablosunda güncelleme işleminden önceki ve sonraki veriler tutulsun.

Güncellenen ürünlerin özet bilgilerinin loglanacağı tabloyu oluşturalım.

```
CREATE TABLE UrunGuncellemeLog
(
 ProductID  INT,
 Name   VARCHAR(50),
 ProductNumber NVARCHAR(25),
 ListPrice  MONEY,
 ModifiedDate DATETIME
);
```

Otomatik loglama işlemini gerçekleştirecek **`UPDATE`** trigger'ı oluşturalım.

```
CREATE TRIGGER trg_UrunGuncelleLog
ON Production.Product
AFTER UPDATE
AS
BEGIN
 DECLARE @ProductID INT, @Name VARCHAR,
    @ProductNumber NVARCHAR, @ListPrice MONEY,
    @ModifiedDate DATETIME;

 SELECT @ProductID = i.ProductID, @Name = i.Name,
   @ProductNumber = i.ProductNumber, @ListPrice = i.ListPrice,
   @ModifiedDate = i.ModifiedDate FROM inserted AS i;
```

```
INSERT INTO UrunGuncellemeLog
 VALUES(@ProductID, @Name, @ProductNumber, @ListPrice, @
ModifiedDate)
END;
```

Bir ürün güncelleyelim.

```
UPDATE Production.Product
SET Name = 'Road-750 Red, 52'
WHERE ProductID = 999;
```

Güncelleme işleminden sonra **UrunGuncellemeLog** tablosu aşağıdakine benzer değerlerde olacaktır.

	ProductID	Name	ProductNumber	ListPrice	ModifiedDate
1	999	R	B	539,99	2013-02-11 23:56:37.073
2	999	R	B	539,99	2013-02-12 00:27:00.983

Tek bir güncelleme yapılmasına rağmen **UrunGuncellemeLog** tablosuna iki kayıt eklendi. Bunun nedeni; **UPDATE** trigger'ın **INSERTED** ve **DELETED** tablolarının her ikisini de kullanıyor olmasıdır. İlk sıradaki kayıt verinin güncellenmeden önceki halini (**DELETED**), ikinci kayıt ise, güncelleme sonrasındaki (**INSERTED**) yeni halini belirtir ve log olarak kaydeder.

Birden Fazla İşlem İçin Trigger Oluşturmak

Trigger'lar tekil işlemler için kullanılabildiği gibi, **INSERT**, **UPDATE** ve **DELETE** işlemlerinin tümünü tek bir trigger ile de takip edebilir.

Bunun diğer trigger'lardan tek farkı **AFTER** tanımlama kısmıdır.

```
CREATE TRIGGER trigger_ismi
ON tablo_ismi
AFTER INSERT, UPDATE, DELETE
```

AFTER ifadesinden sonra aralarına virgül konularak gerekli işlemlerin bildirimi yapılabilir.

INSTEAD OF Trigger

INSTEAD OF Trigger'lar veri değişimi başlamadan hemen önce çalışırlar. Tabloda bir değişiklik yapılmadan, hatta constraint'ler bile devreye girmeden çalışırlar. Sözde tablolar (**INSERTED**, **DELETED**) bu trigger türü tarafından da desteklenir.

İşlem gerçekleşmeden hemen önce devreye girdiği için, bu trigger'lar genel olarak view'lere veri ekleme işlemlerini yönetmek için kullanılır. Tablolar üzerinde de, yapılmak istenen işlemi durdurmak ya da farklı bir işleme yönlendirerek otomatik olarak belirlenen bu işlemin gerçekleşmesini sağlamak için kullanılabilir.

View bölümünde çoklu tablolar ile **JOIN** kullanılarak birleştirilen view'lere trigger'lar ile veri eklenebileceğini ancak zahmetli bir iş olduğunu belirtmiştik. O konuda bahsedilen trigger türü **INSTEAD OF Trigger**'lardır.

Bir table üzerinde, her bir işlem türü (**INSERT**, **UPDATE**, **DELETE**) için sadece tek bir **INSTEAD OF Trigger** oluşturulabilir. Yani, bir tabloda **INSERT** işlemi için **INSTEAD OF Trigger** oluşturulduysa, aynı tabloda **INSERT** işlemi için ikinci bir **INSTEAD OF Trigger** oluşturulamaz.

INSTEAD OF Trigger örnekleri için ilgili tablo ve verileri oluşturalım.

Müşteriler tablosunu oluşturalım.

```
CREATE TABLE Musteriler
(
 MusteriID INT NOT NULL PRIMARY KEY,
 Ad  VARCHAR(40) NOT NULL
);
```

Siparişler tablosunu oluşturalım.

```
CREATE TABLE Siparisler
(
 SiparisID  INT IDENTITY NOT NULL PRIMARY KEY,
 MusteriID  INT NOT NULL REFERENCES Musteriler(MusteriID),
 SiparisTarih DATETIME NOT NULL
);
```

Ürünler tablosunu oluşturalım.

```
CREATE TABLE Urunler
(
 UrunID INT IDENTITY NOT NULL PRIMARY KEY,
 Ad  VARCHAR(50) NOT NULL,
 BirimFiyat MONEY NOT NULL
);
```

`SiparisUrunleri` tablosunu oluşturalım.

```
CREATE TABLE SiparisUrunleri
(
 SiparisID INT NOT NULL REFERENCES Siparisler(SiparisID),
 UrunID INT NOT NULL REFERENCES Urunler(UrunID),
 BirimFiyat MONEY NOT NULL,
 Miktar INT NOT NULL
 CONSTRAINT PKSiparisUrun PRIMARY KEY CLUSTERED(SiparisID, UrunID)
);
```

Tablolara örnek veri ekleyelim.

```
INSERT INTO Musteriler VALUES(1,'Bilişim Yayıncılığı');
INSERT INTO Musteriler VALUES(2,'Çocuk Kitapları Yayıncılığı');

INSERT INTO Siparisler VALUES(1, GETDATE());
INSERT INTO Siparisler VALUES(2, GETDATE());

INSERT INTO Urunler VALUES('İleri Seviye SQL Server T-SQL', 50);
INSERT INTO Urunler VALUES('Keloğlan Masalları', 20);

INSERT INTO SiparisUrunleri VALUES(1, 1, 50, 3);
INSERT INTO SiparisUrunleri VALUES(2, 2, 20, 2);
```

Tablo ve verileri listeleyerek inceleyelim.

```
SELECT * FROM Siparisler;
```

	SiparisID	MusteriID	SiparisTarih
1	1	1	2013-02-12 11:37:10.397
2	2	2	2013-02-12 11:37:10.397

```
SELECT * FROM Urunler;
```

	UrunID	Ad	BirimFiyat
1	1	Ileri Seviye SQL Server T-SQL	50,00
2	2	Keloglan Masallari	20,00

```
SELECT * FROM Musteriler;
```

	MusteriID	Ad
1	1	Bilisim Yayinciligi
2	2	Çocuk Kitaplari Yayinciligi

```
SELECT * FROM SiparisUrunleri;
```

	SiparisID	UrunID	BirimFiyat	Miktar
1	1	1	50,00	3
2	2	2	20,00	2

Bu tablo ve verileri kullanarak **INNER JOIN** ile birleştirilen bir view oluşturacağız. Bu kitabın view bölümünde çoklu veri (**INNER JOIN**) için kullanılan view'leri anlatırken, view üzerinden veri ekleme işleminde bazı kısıtlamaları olduğunu belirtmiştik. Join ile birleştirilmiş bir view'in veri tablolarına **INSERT**, **UPDATE**, **DELETE** yapabilmek için **INSTEAD OF Trigger** kullanılır.

Yukarıda hazırladığımız örnek tablo ve verileri kullanarak bir view oluşturalım. **INSTEAD OF Trigger** örneğimizde bu oluşturulan view'i kullanarak **INSERT**, **UPDATE**, **DELETE** yapacağız.

```
CREATE VIEW vw_MusteriSiparisleri
AS
SELECT S.SiparisID,
  SU.UrunID,
  U.Ad,
  SU.BirimFiyat,
  SU.Miktar,
  S.SiparisTarih
FROM Siparisler AS S
JOIN SiparisUrunleri AS SU
```

```
 ON S.SiparisID = SU.SiparisID
JOIN Urunler AS U
 ON SU.UrunID = U.UrunID;
```

vw_MusteriSiparisleri view'i, siparişlerin önemli bilgilerini listeleyecektir.

```
SELECT * FROM vw_MusteriSiparisleri;
```

	SiparisID	UrunID	Ad	BirimFiyat	Miktar	SiparisTarih
1	1	1	Ileri Seviye SQL Server T-SQL	50,00	3	2013-02-13 10:34:26.173
2	2	2	Keloglan Masallari	20,00	2	2013-02-13 10:34:26.173

INSTEAD OF INSERT Trigger

Tablo ya da view'e fiziksel olarak veri ekleme işlemi yapılmadan önce veriyi inceleyerek, farklı bir işlem tercih edilmesi gereken durumlarda kullanılabilir.

INSTEAD OF Trigger'lar genel olarak view üzerinden veri ekleme işlemi için kullanılır. Bunun nedeni, view'in karmaşık sorgulara sahip olmasıdır. SQL Server, her ne kadar karmaşık sorguları çözümleyebilecek yeteneklere sahip olsa da, bu tür kullanıcı tercihi gereken ve veri bütünlüğü açısından risk oluşturabilecek durumlarda, kendisine yol göstermesi için T-SQL geliştiricisinin müdahale etmesini bekler.

vw_MusteriSiparisleri view'ini kullanarak ilişkilendirilmiş tablolara veri eklemek için bir **INSTEAD OF INSERT** trigger oluşturmalıyız.

```
CREATE TRIGGER trg_MusteriSiparisEkle
ON vw_MusteriSiparisleri
INSTEAD OF INSERT
AS
BEGIN
SET NOCOUNT ON;
 IF(SELECT COUNT(*) FROM inserted) > 0
 BEGIN
  INSERT INTO dbo.SiparisUrunleri
   SELECT i.SiparisID,
     i.UrunID,
     i.BirimFiyat,
     i.Miktar
```

```
   FROM inserted AS i
   JOIN Siparisler AS S
    ON i.SiparisID = S.SiparisID

 IF @@ROWCOUNT = 0
  RAISERROR('Eşleşme yok. Ekleme yapılamadı.', 10, 1)
 END;
SET NOCOUNT OFF;
END;
```

Trigger'ın tetiklenebilmesi için bir veri ekleme gerçekleştireceğiz.

```
INSERT INTO vw_MusteriSiparisleri(
 SiparisID, SiparisTarih, UrunID, Miktar, BirimFiyat)
VALUES(1, '2013-02-02', 2, 10, 20);
```

View içeriğini listeleyelim.

	SiparisID	UrunID	Ad	BirimFiyat	Miktar	SiparisTarih
1	1	1	Ileri Seviye SQL Server T-SQL	50,00	3	2013-02-13 10:34:26.173
2	1	2	Keloglan Masallari	20,00	10	2013-02-13 10:34:26.173
3	2	2	Keloglan Masallari	20,00	2	2013-02-13 10:34:26.173

Veri ekleme işlemini bir tablo değil, view üzerinde gerçekleştirdik. Herhangi bir tetikleme ve ek işlem yapmamış olsak da, oluşturduğumuz trigger bu veri ekleme işleminin gerçekleşmesini sağladı.

Tüm çok tablolu view'ler için **`INSTEAD OF Trigger`** kullanılmak zorundadır demek yanlış olur. SQL Server'ın **`Primary Key`** sütunlar üzerinden sorunsuz olarak erişebileceği view yapılarında trigger'lara gerek yoktur. Ancak gerek olmasa bile, veri bütünlüğü ve işin doğru yoldan yapılma gerekliliği açısından trigger kullanılması önerilir. **`Join`**'li view'lerde genel olarak doğrudan veri ekleme işlemi hata ile sonuçlanır. Bu durumda **`INSTEAD OF Trigger`** kullanılır.

INSTEAD OF UPDATE Trigger

`INSTEAD OF Trigger`'lar, güncelleme işleminden önce gerçekleştirilmesi gereken kontroller ve işlemlerdeki değişiklikler için kullanılabilir.

`Production.Product` tablosunda güncelleme işlemini yönetecek bir trigger oluşturalım.

```
CREATE TRIGGER trg_InsteadOfTrigger
ON Production.Product
INSTEAD OF UPDATE
AS
BEGIN
 PRINT 'Güncelleme işlemi gerçekleştirilmek istendi.';
END;
```

Bu trigger, güncelleme işleminden önce çalışacak ve güncellemenin gerçekleşmesini engelleyecektir.

Production.Product tablosunda **Name** sütununu değiştiren bir güncelleme işlemi yapalım.

İlk olarak güncellenecek kaydı görüntüleyelim.

```
SELECT ProductID, Name, ProductNumber FROM Production.Product
WHERE ProductID = 1;
```

	ProductID	Name	ProductNumber
1	1	Adjustable Race	AR-5381

Kaydın **Name** sütununu değiştirelim.

```
UPDATE Production.Product
SET Name = 'Adjustable Race1'
WHERE ProductID = 1;
```

```
Güncelleme islemi gerçeklestirilmek istendi.

(1 row(s) affected)
```

Güncelleme sorgusu hatasız çalışmasına rağmen kayıt güncellenmedi ve trigger içerisinde belirtilen uyarı mesajı görüntülendi.

INSTEAD OF DELETE Trigger

Veri silme işlemlerinden önce de bazı işlem ya da kontroller yapılmak istenebilir. Örneğin; bir veri silme işlemi gerçekleştirilirken, silme işlemi yerine güncelleme işlemi yapmak gerekebilir.

Bu tür işlemleri gerçekleştirebilmek için **INSTEAD OF DELETE** trigger kullanılır.

vw_MusteriSiparisleri view'ini kullanarak bir ürünün silinmek istendiği senaryosunu ele alalım. Kullanıcı view üzerinden bir ürünü silmek istediğinde, ürünün silinmesini değil, sadece yayından kaldırılmasını sağlayabiliriz. Genellikle veri bütünlüğünü sağlamak için birçok tabloda **AktifMi/isActive** gibi sütunlar bulundurulur. Bu sütunların veri tipi BIT olarak ayarlanır ve 0 değerini alırsa **False**, **1** değerini alırsa **True** olarak değerlendirilir. Yayında olması için bu değerin **True**, yani **1** olması beklenir.

Şimdi, **vw_MusteriSiparisleri** view'i üzerinden bir veri silme sırasında, veriyi silmek yerine ürünün **AktifMi** sütununu **False** yaparak yayından kaldırılmasını sağlayalım.

Senaryodaki işlemi gerçekleştirebilmek için ürünler tablosuna **AktifMi** sütunu ekleyelim.

```
ALTER TABLE Urunler
ADD AktifMi BIT;
```

Mevcut verilerdeki oluşan yeni sütunun değerini belirlemek için bir güncelleme yapalım.

```
UPDATE Urunler
SET AktifMi = 1;
```

Mevcut view içerisinde işlem yapacağımız için view'e **AktifMi** sütununu ekleyelim.

```
ALTER VIEW vw_MusteriSiparisleri
AS
SELECT S.SiparisID,
  SU.UrunID,
  U.Ad,
  SU.BirimFiyat,
  SU.Miktar,
  S.SiparisTarih,
  U.AktifMi
FROM Siparisler AS S
JOIN SiparisUrunleri AS SU
```

```
 ON S.SiparisID = SU.SiparisID
JOIN Urunler AS U
 ON SU.UrunID = U.UrunID;
```

View artık üzerinde veri silme işlemi gerçekleştirmeye hazır. View kullanarak veri silmeye çalışırken, silme işleminden önce devreye girecek ve **DELETE** yerine **UPDATE** sorgusu çalıştırarak ürünü silmeden, sadece **AktifMi** sütunu değerini False yapacak bir trigger oluşturalım.

```
CREATE TRIGGER trg_UrunuYayindanKaldir
ON vw_MusteriSiparisleri
INSTEAD OF DELETE
AS
BEGIN
SET NOCOUNT ON;
 IF(SELECT COUNT(*) FROM deleted) > 0
 BEGIN
  DECLARE @ID INT
  SELECT @ID = UrunID FROM deleted
  UPDATE vw_MusteriSiparisleri
  SET AktifMi = 0
  WHERE UrunID = @ID;
 IF @@ROWCOUNT = 0
  RAISERROR('Eşleşme yok. Pasifleştirme işlemi yapılamadı.', 10, 1)
 END;
SET NOCOUNT OFF;
END;
```

Trigger'ı tetikleyecek işlemi yapmadan önce, **Urunler** tablosundaki veriye bir göz atalım.

	UrunID	Ad	BirimFiyat	AktifMi
1	1	Ileri Seviye SQL Server T-SQL	50,00	1
2	2	Kelogları Masalları	20,00	1

View'i kullanarak ürün silelim.

```
DELETE FROM vw_MusteriSiparisleri WHERE UrunID = 1;
```

Silme işleminden sonra **Urunler** tablosundaki veriye tekrar göz atalım.

```
SELECT * FROM Urunler;
```

	UrunID	Ad	BirimFiyat	AktifMi
1	1	Ileri Seviye SQL Server T-SQL	50,00	0
2	2	Keloglan Masallari	20,00	1

DELETE sorgusuna rağmen verinin silinmediğini, sadece **AktifMi** sütununun değerinin değiştiğini görüyoruz.

Veri silme işlemlerinde doğrudan çok fazla kullanılmasa da, bu tür **DELETE** yerine farklı bir işlem yapmak için tercih edilen bir trigger türüdür. Birçok veritabanı için kritik ve arşiv niteliğinde kayıtlar vardır. Makale yayını yapan bir web sitesinin veritabanında, kullanıcının kendi yazdığı makaleleri yayından kaldırma hakkı olması doğaldır. Ancak silmek ile yayından kaldırmak farklı işlemlerdir. Veritabanından silinen bir kayıt üzerinde farklı işlemler ve nesneler tanımlanmış olabilir. Hatta istatistiksel anlamda bazı özel yazılımlar geliştirilmiş de olabilir. Verilerin tablolardan silinmesi, bu istatistiklerdeki verinin gerçek istatistikleri yansıtmamasına sebep olur. Bu durumda, makaleyi tamamen silmek yerine yayından kaldırmak daha doğru olacaktır. Yayından kaldırma işlemini tabi ki basit bir **UPDATE** ile de gerçekleştirebilirdik. Ancak **UPDATE** kullanmak, veritabanında ilgili tablo üzerine herhangi bir kullanıcı ya da geliştiricinin **DELETE** sorgusunu çalıştırmasını engellemez. Oluşturduğumuz bu trigger ile **DELETE** kullanılsa bile, verinin silinmediği garanti edilmiş olur.

INSTEAD OF DELETE trigger'ı için farklı bir örnek daha yapalım.

Bir tablo ya da view üzerinde aynı komutu (INSERT, UPDATE, DELETE) kullanan tek bir INSTEAD OF Trigger oluşturulabileceğini unutmayın. Yukarıdaki örneği yaptıysanız, şimdi yapacağımız örneği oluşturabilmek için DROP TRIGGER ile yukarıdaki trigger'ı silmeniz gerekir.

Bir siparişi silerken, o sipariş ile ilişkili diğer kayıtları da silecek yeni bir trigger oluşturalım.

```
CREATE TRIGGER trg_MusteriSiparisSil
ON vw_MusteriSiparisleri
INSTEAD OF DELETE
AS
```

```
BEGIN
SET NOCOUNT ON;
 IF(SELECT COUNT(*) FROM deleted) > 0
 BEGIN
  DELETE SU
   FROM SiparisUrunleri AS SU
   JOIN deleted AS d
    ON d.SiparisID = SU.SiparisID
    AND d.UrunID = SU.UrunID
  DELETE S
   FROM Siparisler AS S
   JOIN deleted AS d
    ON S.SiparisID = d.SiparisID
   LEFT JOIN SiparisUrunleri AS SU
    ON SU.SiparisID = d.SiparisID
    AND SU.UrunID = d.SiparisID
 END;
SET NOCOUNT OFF;
END;
```

Yukarıdaki trigger örneği ile view üzerinden bir sipariş silindiğinde, o siparişle ilişkili olan **`SiparisUrunleri`** listesindeki sipariş kaydı da silinecektir.

Trigger'ı tetiklemeden önce ilişkili tablolardaki kayıtları listeleyelim.

```
SELECT * FROM Siparisler;
GO
SELECT * FROM SiparisUrunleri;
```

	SiparisID	MusteriID	SiparisTarih
1	1	1	2013-02-12 15:42:34.280
2	2	2	2013-02-12 15:42:34.280

	SiparisID	UrunID	BirimFiyat	Miktar
1	1	1	50,00	3
2	2	2	20,00	2

Şimdi yapacağımız silme işlemini gerçekleştirmeden önce, daha önce oluşturduğumuz **`Urunler`** ve **`Siparisler`** tablolarını **`DROP`** ile silerek tekrar oluşturun ve view'in hata üretmemesi için **`Urunler`** tablosuna **`AktifMi`** sütunu tekrar ekleyerek, daha önceki işlemleri gözden geçirin.

Trigger'ı tetikleyecek veri silme işlemini yapalım.

```
DELETE vw_MusteriSiparisleri WHERE SiparisID = 1 AND UrunID = 1;
```

Silme işleminden sonra, ilgili tablodaki verileri tekrar listeleyelim.

```
SELECT * FROM Siparisler;
GO
SELECT * FROM SiparisUrunleri;
```

	SiparisID	MusteriID	SiparisTarih
1	2	2	2013-02-12 15:42:34.280

	SiparisID	UrunID	BirimFiyat	Miktar
1	2	2	20,00	2

IF UPDATE() VE COLUMNS_UPDATED()

UPDATE trigger'da, bazen ilgilenilen sütunların zaten değişmiş olup olmadığının bilinmesi gerekebilir. Belirlenen bu zaten değişmiş olan sütunlardan, trigger'ın haberinin olması, trigger'da uygulanacak işlemlerin azaltılmasını sağlar. Yani trigger'a, kendi içerisinde bulunan sütunların güncellik durumunu haber vererek trigger'ın daha performanslı ve gerekli işlemleri uygulaması sağlanabilir.

UPDATE() FONKSİYONU

UPDATE() fonksiyonu, trigger çalışma alanı içerisinde çalışan bir fonksiyondur. Belirli bir sütunun güncellenip güncellenmediğini öğrenmek için kullanılabilir. Güncelleme bilgisini **Boolean** veri tipiyle **True/False** olarak bildirir.

Örneğin; **Production.Product** içerisinde **ProductNumber** sütununun güncellenmeye karşı kontrol edilmesi gerekebilir. Bir ürünün ürün numarasının değişmesi veritabanı bütünlüğünü etkileyebilir. **ProductNumber** sütununun değiştirilmesini engellemek için trigger içerisinde **UPDATE()** fonksiyonunu kullanalım.

```
CREATE TRIGGER Production.ProductNumberControl
ON Production.Product
AFTER UPDATE
```

```
AS
BEGIN
 IF UPDATE(ProductNumber)
 BEGIN
  PRINT 'ProductNumber değeri değiştirilemez.';
  ROLLBACK
 END;
END;
```

Yukarıdaki trigger oluşturulduktan sonra, **Management Studio** ya da **SQL** ile güncellenmeye karşı korumalı hale gelmiştir.

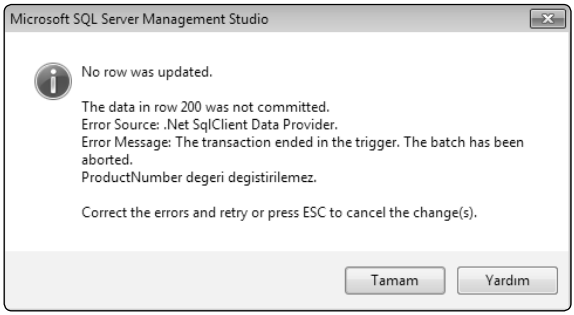
SQL ile güncellemeyi deneyelim.

```
UPDATE Production.Product
SET ProductNumber = 'SK-9283'
WHERE ProductID = 527;
```

```
(1 row(s) affected)
ProductNumber degeri degistirilemez.
Msg 3609, Level 16, State 1, Line 2
The transaction ended in the trigger. The batch has been aborted.
```

Güncelleme işlemi başarısız oldu ve **`PRINT`** ile belirtilen mesaj ekranda görüntülendi.

Management Studio ile güncellemeyi deneyelim.



Her iki şekilde de artık bu sütun değeri değişikliklere karşı korumalıdır.

`Production.ProductNumberControl` trigger'ı, **`Production.Product`** tablosunda yapılan güncelleme işlemlerini takibe eder. Güncelleme işlemleri içerisinde

ProductNumber sütunu varsa, bu sütunu korumaya alarak güncelleme işlemini reddeder.

Eğer ilgili tabloda farklı bir sütun üzerinde güncelleme gerçekleştirilirse, **UPDATE** sorgusu içerisinde **ProductNumber** sütunu olmadığı sürece güncelleme işlemi sorunsuz çalışacaktır.

COLUMNS_UPDATED() Fonksiyonu

Trigger ile güncelleme işlemlerinde kullanılır. **UPDATE()** fonksiyonu ile benzer amaca sahiptir. **UPDATE()** fonksiyonu tekil sütun değişikliklerini takip ederken, **COLUMNS_UPDATED()** fonksiyonu çoklu sütun değişikliklerini takip etmek için kullanılır. Bu fonksiyon, binary olarak değişen sütunların listesini metinsel veri tipi olan **VARCHAR** olarak döndürür.

Söz Dizimi:

```
CREATE TRIGGER trigger_ismi
ON tablo_ismi
FOR UPDATE[, INSERT, DELETE]
AS
IF COLUMNS_UPDATED() & maskeleme_degeri > 0
BEGIN
  -- Sütun değişikliklerine göre çalışacak sorgu bloğu.
END;
```

İç İçe Trigger (Nested Trigger)

Birbirini tetikleyerek çalışan trigger'lara iç içe trigger denir. Buradaki iç içe kavramı yanlış anlaşılmamalıdır. Bir trigger içerisinde başka bir trigger oluşturarak bunu tetiklemek anlamına gelmez. Zaten trigger'ların kullanıcı tarafından tetiklenmediğini, belirli olaylara karşı otomatik olarak tetiklendiğini biliyoruz.

Örneğin; bir işlem sonucunda tetiklenen trigger'ın farklı tablo üzerinde gerçekleştirdiği veri ekleme ya da güncelleme, silme gibi herhangi bir işlemin sonucunda daha önceden tanımlanmış bir trigger'ın tetiklenmesi durumuna iç içe trigger denir. Yani, bir trigger'ın tetiklenmesiyle gerçekleşen işlemin üzerinde bulunduğu view ya da tablo'da, o trigger tarafından yapılan işlem için oluşturulmuş bir trigger'ın otomatik olarak tetiklenmesidir.

Tetiklenen her trigger bir seviyedir. SQL Server, bu şekilde 32 seviyeye kadar iç içe trigger'ı destekler. Yani 32 trigger birbirini tetikleyebilir.

Trigger'ların seviyesini öğrenebilmek için **@@NESTLEVEL** ya da aşağıdaki söz dizimi kullanılabilir.

Söz Dizimi:

```
SELECT trigger_nestlevel(object_ID('trigger_ismi'));
```

İç içe trigger çok kullanılmayan ve açıkçası oldukça riskli bir kullanımdır. Trigger'lar otomatik tetiklenirler ve iç içe trigger'lardan herhangi bir trigger işleminde hata meydana gelirse, tüm trigger'ların yaptığı işlemler için **ROLLBACK** uygulanarak işlemler geri alınır.

SQL Server varsayılan olarak iç içe trigger'ı destekler. Ancak bu özelliğin kapatılması ya da tekrar açılması mümkündür.

Aşağıdaki yöntem ile iç içe trigger özelliği kapatılabilir.

```
sp_configure 'nested triggers', 0
```

İç içe trigger özelliğinin tekrar açılabilmesi için 0 değeri yerine 1 kullanmak yeterlidir.

Recursive Trigger

SQL Server'ın varsayılan ayarlarında trigger'lar kendine atıflı olarak çalışmaz. Yani bir tablodaki herhangi bir sütun üzerinde yapılan değişiklik nedeniyle tetiklenen trigger, aynı tablo üzerinde başka bir sütunda değişikliğe neden oluyorsa, ikinci değişiklik için trigger tetiklenmez. Bu durumun tersine olarak tekrar tekrar çalışması yönetimi zorlaştırır.

Ancak, özel durumlar nedeniyle bazen esneklik gerekebilir. Bu özelliğinde tersi şekilde, tekrarlamaya izin vermek için aşağıdaki ayarlama yöntemi kullanılabilir.

```
ALTER DATABASE veritabani_ismi
SET RECURSIVE_TRIGGERS ON
```

DDL Trigger'lar

DDL (*Data Definition Language*), olarak tanınan **CREATE**, **ALTER**, **DROP** komutlarını içerir. Veritabanında nesne oluşturmak, değiştirmek ve silmek için kullanılan bu komutlar için de trigger'lar tanımlanabilir.

DDL trigger'lar, sadece **AFTER** trigger olarak tanımlanabilirler. **INSTEAD OF** türden bir DDL trigger oluşturulamaz.

Daha önce birçok örnek yaptığımız DML trigger'ları view ve tablolar üzerinde oluşturulduğu için, sadece veri değişimlerine karşı duyarlıdır. Amaçları, veri değişimindeki olayları yönetmektir. DDL trigger'lar ise veritabanı ve sunucu seviyeli olarak tanımlanabilirler. Örneğin; veritabanında bir tablo, prosedür ya da başka bir trigger'ın oluşturulması gibi olaylarla ilgilenir ve bu olaylarla ilgili durumlarda tetiklenir.

DDL trigger'lar için de **ROLLBACK** ile işlemleri geri alma durumu söz konusu olabilir. DDL trigger'lar sadece **eventdata()** fonksiyonu ile ortam hakkında bilgi alabilirler. Trigger içerisinde **eventdata()** fonksiyonu çağrılarak, trigger'ı tetikleyen olay ile ilgili ayrıntılı bilgiler içeren XML veri elde edilebilir.

DDL Trigger Söz Dizimi:

```
CREATE TRIGGER trigger_ismi
ON {DATABASE | ALL SERVER}
FOR { veritabani_seviyeli_olaylar | sunucu_seviyeli_olaylar }
AS
-- Trigger sorgu gövdesi
```

Veritabanı Seviyeli DDL Trigger'lar

Veritabanı seviyeli bir DDL trigger oluşturmak için **ON** komutundan sonra **DATABASE** kullanılmalıdır. DDL trigger'ı oluştururken bir veritabanı ismi verilmez. Hangi veritabanı için çalıştırılırsa, o veritabanı için geçerli bir trigger olacaktır.

AdventureWorks veritabanı seçili iken yeni bir DDL trigger oluşturalım. Bu trigger, veritabanında tablo, prosedür, view oluşturmayı yasaklasın.

```
CREATE TRIGGER KritikNesnelerGuvenligi
ON DATABASE
FOR CREATE_TABLE, CREATE_PROCEDURE, CREATE_VIEW
AS
   PRINT 'Bu veritabanında tablo, prosedür ve view oluşturmak yasaktır!'
   ROLLBACK;
```

Oluşturulan bu trigger ile, artık **AdventureWorks** veritabanı üzerinde tablo, prosedür ya da view oluşturulamayacaktır.

Trigger'ı test etmek için bir tablo oluşturmayı deneyelim.

```
CREATE TABLE test ( ID INT );
```

```
Bu veritabaninda tablo, prosedür ve view olusturmak yasaktir!
Msg 3609, Level 16, State 2, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

Tablo oluşturulamadı ve 16. seviyeden kritik bir hata olarak kırmızı hata mesajı fırlatıldı.

Veritabanı seviyeli DDL trigger'lar nesneler ile gerçekleştirilen işlemler için log tutma amacı ile de kullanılabilir.

EventsLog adında bir tablo oluşturalım.

```
CREATE TABLE EventsLog
(
 EventType VARCHAR(50),
 ObjectName VARCHAR(256),
 ObjectType VARCHAR(25),
 SQLCommand VARCHAR(MAX),
 UserName VARCHAR(256)
);
```

Oluşturacağımız DDL trigger'ı, **AdventureWorks** veritabanı üzerinde oluşturulan tablo, view, procedure ve trigger nesnelerinin bilgilerini **EventsLog** tablosuna log olarak kaydetsin.

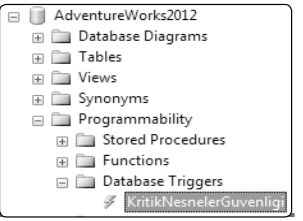
```
CREATE TRIGGER EventLogCreateBackup
ON DATABASE
FOR CREATE_PROCEDURE, CREATE_TABLE, CREATE_VIEW, CREATE_TRIGGER
AS
BEGIN
SET NOCOUNT ON;

DECLARE @Data XML;
SET @Data = EVENTDATA();

INSERT INTO dbo.EventsLog(EventType, ObjectName, ObjectType,
      SQLCommand, UserName)
VALUES(@Data.value('(/EVENT_INSTANCE/EventType)[1]', 'VARCHAR(50)'),
       @Data.value('(/EVENT_INSTANCE/ObjectName)[1]',
'VARCHAR(256)'),
       @Data.value('(/EVENT_INSTANCE/ObjectType)[1]',
'VARCHAR(25)'),
       @Data.value('(/EVENT_INSTANCE/TSQLCommand)[1]',
'VARCHAR(MAX)'),
       @Data.value('(/EVENT_INSTANCE/LoginName)[1]', 'VARCHAR(256)')
);
SET NOCOUNT OFF;
END;
```

`EventsLog` tablosu ve **`EventLogCreateBackup`** isimli trigger'ı test edebilmek için, önceki örnekte tablo ve view oluşturmayı engelleyen trigger'ın kaldırılması gerekir.

Trigger, **Management Studio** ile yandaki gibi kaldırılabilir. Sağda görünen trigger ismine sağ tıklayıp **Delete** butonuna tıklandığında trigger silinecektir.



Yeni bir tablo ve view oluşturalım.

```
CREATE TABLE test1
(ID INT);

CREATE VIEW view1
AS
SELECT * FROM test1;
```

EventsLog tablosunu kontrol edelim.

```
SELECT * FROM EventsLog;
```

	EventType	ObjectName	ObjectType	SQLCommand	UserName
1	CREATE_TABLE	test1	TABLE	CREATE TABLE test1 (ID INT);	dijibil-pc\dijibil
2	CREATE_VIEW	view1	VIEW	CREATE VIEW view1 AS SELECT * FROM test1	dijibil-pc\dijibil

Log tutulması istenen nesneler türünden nesne oluşumlarında, EventsLog tablosu log tutmaya devam edecektir. Bu işlem genel olarak **SQL Server DBA**'lerin veritabanının yönetimi ve güvenlik ayarlarını gözden geçirmesi için kullanılır.

Sunucu Seviyeli DDL Trigger'lar

Sunucu seviyeli DDL trigger'lar, sunucunun bütününe özgü işlemlere duyarlıdır. Sunucu seviyeli trigger oluşturmak için **ON** komutundan sonra **ALL SERVER** kullanılmalıdır.

Yeni bir veritabanı oluşturma işlemini yakalayacak trigger oluşturalım.

```
CREATE TRIGGER SunucuBazliDegisiklikler
ON ALL SERVER
FOR CREATE_DATABASE
AS
PRINT 'Veritabanı oluşturuldu.';
SELECT EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]',
'NVARCHAR(MAX)');
```

	(No column name)
1	CREATE DATABASE dbtest

```
Veritabanı oluşturuldu.

(1 row(s) affected)
```

Benzer şekilde, yeni bir login oluşturma işlemi de yakalanabilir. Yukarıdaki trigger kullanımının **FOR** kısmını aşağıdaki gibi değiştirerek login oluşturma işlemleri yakalanabilir.

```
...
FOR DDL_LOGIN_EVENTS
...
```

Yeni bir login oluşturma işlemiyse aşağıdaki gibi yapılabilir.

```
CREATE LOGIN yakalanacak_login WITH PASSWORD = 'pwd123';
```

Sunucu bazlı trigger'lar **master.sys.server_triggers** sistem kataloğunda yer alır. Oluşturulan trigger'ı listeleyelim.

```
SELECT * FROM master.sys.server_triggers;
```

	name	object_id	parent_class	parent_class_desc	parent_id	type	type_desc	create_date	modify_date	is_ms_shipped	is_disabled
1	SunucuBazliDegisiklikler	279672044	100	SERVER	0	TR	SQL_TRIGGER	2013-02-12 23:03:56.277	2013-02-12 23:03:56.277	0	0

Trigger Yönetimi

Trigger nesneleri, diğer tüm nesneler gibi değiştirilebilir, silinebilir. Hatta trigger'ların, aktiflik ve pasifliği değiştirilerek kullanımdan kaldırılabilir ve tekrar kullanıma sunulabilir.

Trigger'ı Değiştirmek

Trigger'ların içeriği ve etkilendiği olaylar değiştirilebilir. **UPDATE** için hazırlanan bir trigger, **INSERT** ve **DELETE** işlemlerinde de tetiklenmesi için düzenlenebilir. **sp_rename** sistem prosedürü kullanılarak trigger'ın sadece ismi değiştirilebilir.

SunucuBazliDegisiklikler trigger'ını değiştirerek login oluşturma olaylarını da takip edecek şekilde düzenleyelim.

```
ALTER TRIGGER SunucuBazliDegisiklikler
ON ALL SERVER
FOR CREATE_DATABASE, DDL_LOGIN_EVENTS
AS
PRINT 'Veritabanı ya da Login oluşturma olayı yakalandı.';
SELECT EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]',
 'NVARCHAR(MAX)');
```

Artık trigger'ımız login oluşturma işlemlerini de tetikleyebilir.

```
CREATE LOGIN testLogin WITH PASSWORD = '123';
```

	(No column name)
1	CREATE LOGIN testLogin WITH PASSWORD = '******'

Trigger'ları Kapatmak ve Açmak

Trigger'ların tetiklenmesi geçici ya da kalıcı olarak engellenebilir. Bu işlem iki şekilde gerçekleşebilir. Bir trigger'ın adını belirterek, tablo ya da view üzerinde sadece bu trigger'ın tetiklenmeye kapatılması sağlanabileceği gibi, doğrudan tablo ya da view adı belirterek, bir trigger ismi belirtmeden tablo ya da view'deki tüm trigger'ların kapatılması sağlanabilir.

Aynı yöntemler ile kapatılan trigger ya da trigger'lar açılabilir.

Bir tablodaki tek bir trigger'ı kapatalım.

```
ALTER TABLE Production.Product
DISABLE TRIGGER ProductNumberControl;
```

Bir tablodaki tüm trigger'ları kapatalım.

```
ALTER TABLE Production.Product
DISABLE TRIGGER ALL;
```

Kapatılan tek bir trigger'ı tekrar açalım.

```
ALTER TABLE Production.Product
ENABLE TRIGGER ProductNumberControl;
```

Bir tablodaki tüm trigger'ları tekrar açalım.

```
ALTER TABLE Production.Product
DISABLE TRIGGER ALL;
```

Trigger'ları Silmek

Tablo ya da view üstündeki trigger'ları silmek için tablo ya da view'in sahibi olmak ya da **`sysadmin`**, **`db_owner`** rollerinden birine sahip olmak gerekir.

```
DROP TRIGGER trigger_ismi
```

Bir tablo ya da view üzerinde tanımlı DML trigger'ı, üzerinde bulunduğu tablo ya da view silindiğinde otomatik olarak silinir.

Veritabanı Seviyeli DDL Trigger'ları Silmek

DML trigger'lardan farklı olarak, bir DDL trigger silmek için söz diziminin sonu **ON DATABASE** ile bitmek zorundadır.

Söz Dizimi:

```
DROP TRIGGER trigger_ismi
ON DATABASE
```

EventLogCreateBackup isimli veritabanı seviyeli DDL trigger'ı silelim.

```
DROP TRIGGER EventLogCreateBackup
ON DATABASE;
```

Sunucu Seviyeli DDL Trigger'ları Silmek

Sunucu seviyeli DDL trigger'ları silmek için söz dizimi **ON ALL SERVER** ifadesi alır.

Söz Dizimi:

```
DROP TRIGGER trigger_ismi
ON ALL SERVER;
```

SunucuBazliDegisiklikler isimli sunucu bazlı DDL trigger'ı silelim.

```
DROP TRIGGER SunucuBazliDegisiklikler
ON ALL SERVER;
```