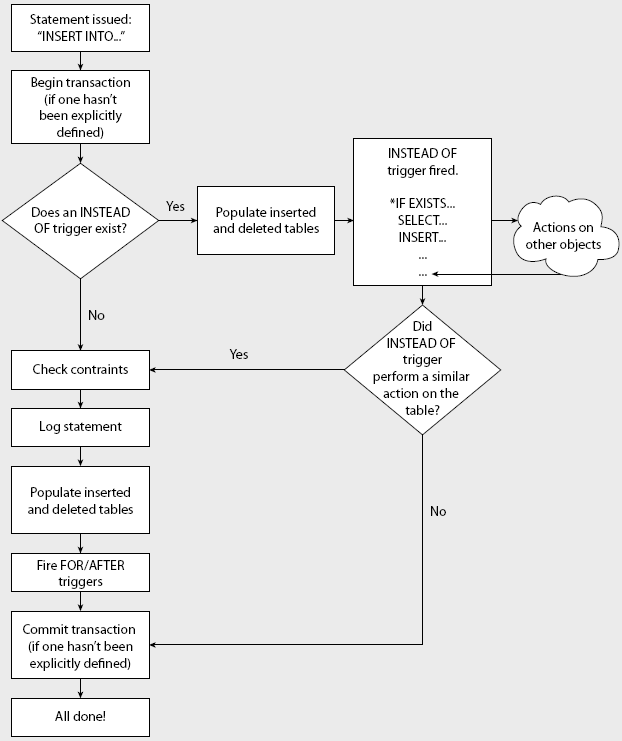
**Modul 153**

***-Block 04 (Trigger)-***



# **1) Übersicht**

Trigger sind Funktionen die das Datenbankmanagement automatisch, beim Eintreffen von definierten Ereignissen), ausführt.

Ein Trigger besteht aus SQL-Anweisungen und gehört immer zu genau einer Tabelle. Pro Tabelle kann es aber mehrere Trigger geben.

Erinnern Sie sich an die automatische Löschweitergabe (on delete cascade) oder an die referenzielle Integrität? Wir haben diese Sachverhalte einfach mit SQL-DDL formuliert und hatten nichts weiter damit zu tun. Das DBMS erstellt uns im Hintergrund Trigger (Funktionen), welche diese Arbeiten für uns erledigen.

Oftmalls unterstützt ein Datenbanksystem Aktionen wie Löschweitergabe oder Aktualisierungsweitergabe nicht automatisch. Dann kann man, sofern das DBMS Trigger unterstützt, diese Arbeiten selbst vornehmen. Dies war zum Besipiel bei SQL-Server, vor der Version 2000 oder ist bei MySQL Version 5 der Fall.

Tirigger sind aber auch viel flexibler als die automatischen Lösch-und Aktualisierungsfunktionen. Mit Trigger könnte man z.B. eine Beziehung so erstellen, dass das Löschen eines Kunden nur erlaubt ist, wenn der Kunde in den letzten zwei Jahren weniger als 10 Artikel bestellt hat. Wird ein Kunde gelöscht, sollen aber trotzdem auch alle Bestelldetails gelöscht werden.

Trigger werden in den folgenden Fällen angewendet:

* Zur Wahrung der Datenintegrität, die über die einfache Aktualisierung von Referencen hinausgeht.
* Zur Aktualisierung von Summen
* Zur Aktualisierung von Spalten, die rechenergebnisse beinhalten.
* Zur Pflege von Revisionsdatensätzen
* Zur durchführung von referenziellen Operationen wie kaskadierendes Löschen, etc.
* Zur Auslösung externer Aktionen (E-Mail auslösen bei Unterschreitung des Mindestbestandes.

Ein Trigger gehört immer zu einer Tabelle und reagiert auf eines oder mehrere der folgenden Ereignisse

* Insert (Ein neuer Datensatz wird hinzugefügt)
* Update (Ein bestehender Datensatz wird geändert)
* Delete (Ein bestehender Datensatz wird gelöscht)

Somit sieht die Kopf-Definition eines Triggers wie folgt aus:

CREATE TRIGGER PersonMutation ON Personen

FOR INSERT, UPDATE

AS

Begin

..................

(SQL- BEFEHLE ODER TRANSACT-SQL-BEFEHLE)

..................

END

Pro Tabelle und Ereigniss kann man mehrere Trigger definieren. Die Ablauf Reihenfolge ist aber nicht voraussagbar. Wenn wir zum Beispiel drei Insert Trigger für die Tabelle Personen definieren, werden alle drei Trigger bei einem Einfüge-Vorgang ausgelöst. Man kann aber keine Aussagen darüber machen, welcher der drei Trigger zuerst oder zuletzt abgearbeitet wird. Bei zeitkritischen Aktionen sollte somit immer nur ein Trigger verwendet werden.

Nehmen wir an, eine Tabelle besitzt einen Trigger, welcher auf Delete-Ereignisse reagiert. Ein SQL-Statement der Form “delete from Tabelle where ID between 1 and 50” löscht nun 50 Datensätze dieser Tabelle. Der Trigger wird aber nur einmal aufgerufen. Mit der Systemfunktion @@rowcount können wir Abfragen wieviele Zeilen von der Aktion betroffen sind.

**Aufgabe 1:** Gemeinsam erstellen wir ein Beispiel eines kleinen Triggers. Erstellen sie zuerst die Datenbank und den Trigger via der Ausführung des folgenden Skriptes:

CREATE DATABASE TriggerTest

Go

USE DATABASE TriggerTest

CREATE TABLE test\_trigger

(

col1 int,

col2 char(6)

)

GO

insert into test\_trigger values(1, 'First');

insert into test\_trigger values(2, 'Second');

insert into test\_trigger values(3, 'Third');

insert into test\_trigger values(4, 'Fourth');

insert into test\_trigger values(5, 'Fifth');

GO

CREATE TRIGGER delete\_test

ON test\_trigger

FOR Delete

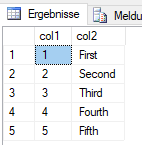
AS

PRINT 'Sie haben Zeilen gelöscht'

GO

SELECT \* FROM test\_trigger

Sie bekommen die folgende Meldung als Reaktion auf die Select-Anfrage zurück:



Testen sie nun unseren Mini-Trigger. Was ist die Reaktion des DBMS auf jeden der folgenden beiden Befehle?

DELETE test\_trigger where col1 =0

Reaktion DBMS:

DELETE test\_trigger where col1 =1

Reaktion DBMS:

Erklären sie die Funktion des erstellten Triggers:**Aufgabe 2:** Wir möchten den Trigger durch einen neuen Trigger ersetzen, der etwas mehr Logik besitzt:

Löschen sie den zuvor erstellten Trigger mit folgendem Befehl:

DROP TRIGGER delete\_test

Erstellen sie den neuen Trigger «delete\_test»:

CREATE TRIGGER delete\_test

ON test\_trigger

FOR Delete

AS

BEGIN

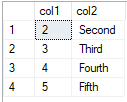
IF @@rowcount =0

RETURN

ELSE

PRINT 'Sie haben Zeilen gelöscht'

END

Unsere Tabelle hat aktuell die folgenden Inhalte:

SELECT \* FROM test\_trigger

Testen sie nun den neuen Trigger. Was ist die Reaktion des DBMS auf jeden der folgenden beiden Befehle?

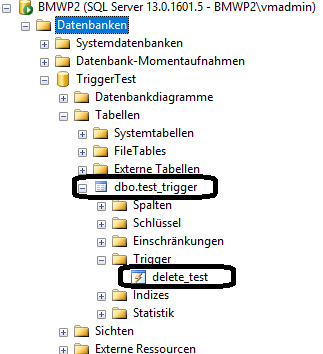
**DELETE test\_trigger where col1 =0**

Reaktion DBMS:

**DELETE test\_trigger where col1 >2**

Reaktion DBMS:

Erklären sie die Funktion des erstellten Triggers:



Löschen Sie danach die zuvor erstellten beiden Elemente (Trigger und Tabelle).

Wie sie im nebenstehenden Bild sehen, gehört der Trigger zur Tabelle test\_trigger.

Es genügt also nur die Tabelle zu löschen:

**DROP TRIGGER delete\_test**

**2) Die zwei “Schattentabellen” inserted und deleted**

Jede Tabelle besitzt zwei Schattentabellen, welche die Bezeichnungen inserted und deleted tragen. Die Schattentabellen sind von der Struktur her identisch aufgebaut wie die “Original“-Tabelle, nur sind sie normalerweise leer.

Gehen wir einmal von folgender Tabellenstruktur aus:

|  |  |  |
| --- | --- | --- |
| Personen | | |
| PNr | Name | Vorname |
| 1 | Felder | Karin |
| 2 | Näf | Roger |
| 3 | Schneider | Toni |

**Einfügevorgang:**

Mit der folgenden insert-Anweisung fügen wir eine neue Zeile in die Tabelle Personen ein:

insert into Personen values (5, “Decker”, “Harald” )

Was läuft nun genau ab?

Bevor die Daten in die Tabelle Personen effektiv eingefügt werden, werden Sie in die Inserted-Schattentabelle der Tabelle Personen geschrieben.

Nun wird der (oder mehrere) Trigger ausgeführt, die auf ON INSERT reagieren.

Innerhalb dieses Triggers können wir nun auf den Inhalt der inserted-Tabelle zugreifen.. Der Trigger kann nun die Einfügeoperation zulassen oder verbieten. Wenn er die Operation zulässt werden die Daten von der Schattentabelle INSERTED in die Tabelle Personen übertragen und in der Schattentabelle gelöscht.

Wenn der Trigger die Operation nicht zulässt (Rollback) , wird nur der Inhalt der Schattentabelle (INSERTED) gelöscht.

Wenn Sie , z.B. mit der folgenden Anweisung, eine neue Zeile in die Tabelle Personen einfügen, wird der Trigger, der auf INSERT hört, ausgelöst.

*Insert into Personen values (5, ‘Decker’, ‘Harald’)*

Innerhalb dieses Triggers stehen uns die Änderungsinformationen in Form einer inserted-Tabelle zur Verfügung. Inserted = {5, “Decker”, “Harald”} .

**Löschvorgang:**

Mit der folgenden delete-Anweisung löschen wir eine Zeile in der Tabelle Personen:

delete Personen where PNr = 2

Was läuft nun genau ab?

Bevor die Daten in der Tabelle Personen effektiv gelöscht werden, wird eine Kopie der Daten in die deleted-Schattentabelle der Tabelle Personen geschrieben.

Nun wird der (oder mehrere) Trigger ausgeführt, die auf ON DELETE reagieren.

Innerhalb dieses Triggers können wir nun auf den Inhalt der deleted-Tabelle zugreifen. Der Trigger kann nun die Löschoperation zulassen oder verbieten. Wenn er die Operation zulässt werden die Daten in der Schattentabelle DELETED und in der Tabelle Personen gelöscht.

Wenn der Trigger die Operation nicht zulässt (Rollback), wird nur der Inhalt der Schattentabelle (DELETED) gelöscht.

Wenn Sie, z.B. mit der folgenden Anweisung, eine bestehende Zeile aus der Tabelle Personen löschen, wird der Trigger, der auf DELETE hört, ausgelöst.

*delete Personen where PNr = 2*

Innerhalb dieses Triggers stehen uns die zu löschenden informationen in Form einer deleted-Tabelle zur Verfügung. deleted = {2, “Näf”, “Roger”} .

**Änderungsvorgang:**

Mit der folgenden update-Anweisung ändern wir eine Zeile in der Tabelle Personen:

update Personen set Name = “Feldermann“ where PNr = 1

Was läuft nun genau ab?

Bevor die Daten in der Tabelle Personen effektiv geändert werden, wird der alte Zustand der betroffenen Daten (Zeilen) in die Schattentabelle “DELETED“ kopiert. Der neue Zustand der Daten wird in die Schattentabelle INSERTED kopiert.

z.B. deleted = {1, “Felder”, “Karin”} ; inserted = {1, “Feldermann”, “Karin”}

Nun wird der (oder mehrere) Trigger ausgeführt, die auf ON UPDATE reagieren.

Innerhalb dieses Triggers können wir nun auf den Inhalt der deleted-Tabelle und auf den Inhalt der inserted-Tabelle zugreifen. Der Trigger kann auch hier wieder die Updateoperation zulassen oder verbieten.

Wenn Sie, z.B. mit der folgenden Anweisung, einen bestehenden Datensatz änderen, wird der Trigger, der auf UPDATE hört, ausgelöst.

*update Personen set Name = Feldermann where PNr = 1*

Innerhalb dieses Triggers stehen uns die Änderungsinformationen in Form einer inserted und deleted-Tabelle zur Verfügung. Deleted = Zeile vor der Änderung; Inserted = Zeile nach der Änderung. deleted = {1, “Felder”, “Karin”} ; inserted = {1, “Bissig-Felder”, “Karin”}

**Die grundsätzliche Idee:**

Bevor eine Aktion definitiv ausgeführt wird, werden die zu löschenden, die einzfügenden oder zu ändernden Informationen in die inserted und deleted Schattentabellen geschrieben. Die Originaltabelle wird zu diesem Zeitpunkt noch nicht geändert. Nun wird, falls existent ein entsprechender Trigger aufgerufen. Wenn der Trigger keinen Abbruch (Rollback) befiehlt, können die Änderungen ausgeführt werden und die inserted und deleted Schattentabellen werden wieder geleert.

**3) Referenzielle Vorgänge mit Hilfe von Triggern implementieren:**

**Aufgabe 3:** Priorität der Referenzielle Integrität vs Trigger prüfen.

Erstellen sie das folgende Skript und führen Sie es aus.

CREATE TABLE Personen

(

PersNr int NOT NULL PRIMARY KEY,

Namen varchar(30) NULL

)

GO

CREATE TABLE Kinder

(

KindNr int NOT NULL,

Name varchar(30) NULL,

fk\_Eltern int NULL,

CONSTRAINT pk\_Kinder PRIMARY KEY(KindNr),

CONSTRAINT bez\_Kind\_Eltern FOREIGN KEY (fk\_Eltern) REFERENCES Personen (PersNr)

)

GO

insert into Personen values (1, 'Buettel')

insert into Personen values (2, 'Rappin')

insert into Personen values (3, 'Haller')

insert into Kinder values (100, 'Kay', 1);

insert into Kinder values (101, 'Cedric', 1);

insert into Kinder values (102, 'Eric', 2);

CREATE TRIGGER LOESCHPERSON ON Personen

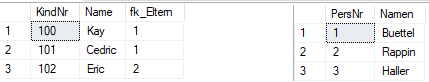
FOR DELETE

AS

BEGIN

delete from Kinder where fk\_Eltern IN (Select PersNr from deleted)

END

GO

SELECT \* FROM Kinder

SELECT \* FROM Personen

Erklären Sie, was der Trigger LOESCHPERSON machen wird.

Testen Sie den Trigger mit den folgenden beiden Anweisung.

**delete from Personen where PersNr =3; --Test 1**

Was gibt das DBMS zurück? Was fällt ihnen auf?

**delete from Personen where PersNr =2; --Test 2**

Was fällt Ihnen auf?

Deaktivieren Sie die referenzielle Integritätskontrolle durch dasDBMS mit der folgenden SQL-Anweisung und testen sie den Trigger erneut.

**ALTER TABLE Kinder NOCHECK CONSTRAINT bez\_Kind\_Eltern**

Was fällt Ihnen auf?

**Aufgabe 4:**

Löschen Sie den vorher erstellten Trigger LOESCHPERSON (mit dem Befehl Drop Trigger Loeschperson) und erstellen und testen Sie den folgenden Trigger:

Erklären sie, was der Trigger macht

CREATE TRIGGER LOESCHPERSON ON PERSONEN

FOR DELETE

AS

BEGIN

update Kinder set fk\_Eltern = NULL where Kinder.fk\_Eltern IN (Select PersNr from deleted)

END

Löschen Sie alle zuvor erstellten Trigger und erstellen und testen Sie den folgenden Trigger:

CREATE TRIGGER VERAENDEREMITARBEITERNUMMER ON Personen

FOR UPDATE

AS

BEGIN

if update(PersNr)

BEGIN

update Kinder set Kinder.fk\_Eltern = (Select PersNr from inserted)

from Kinder

where Kinder.fk\_Eltern = (Select PersNr from deleted)

END

END

Testen Sie den Trigger mit dem folgenden Aufruf:

update Personen Set PersNr =66 where PersNr =1;

Erklären sie, was der Trigger macht.

Löschen Sie alle zuvor erstellten Trigger und erstellen und testen Sie den folgenden Trigger:

CREATE TRIGGER FKCHECK ON KINDER

FOR UPDATE, INSERT

AS

BEGIN

If not exists(Select \* from Personen where PersNr IN (Select fk\_Eltern from inserted))

BEGIN

raiserror ( ’Der Fremdschlüssel verweist auf einen PK, welcher nicht existiert’,11,10)

rollback transaction

END

END

Testen Sie den Trigger mit dem folgenden Aufruf:

insert into Kinder values (444, 'Gretali', 55);

Erklären sie, was der Trigger macht.

**Aufgabe 5:** Überfliegen Sie das verlinkte PDF-Dokument zum Thema “Referenzielle Vorgänge mit Hilfe von Triggernimplementieren“ und versuchen Sie zu ver­stehen, wie man mittels Trigger die Kontrolle über die Referenzielle Integrität implementieren kann.

Beim obenstehenden Beispiel Personen – Kind wurde die Beziehung mittels FOREIGN KEY –Klausel definiert.

Nun möchten wir die Beziehung mittels Trigger erstellen. Überlegen Sie, wieviele Trigger es braucht um die referenzielle Integrität zu waren. Das heisst, jede Aktion, welche die Ref.Integrität gefährdet soll verboten werden.

Beschreiben Sie genau zu welcher Tabelle jeder Trigger gehört, bei welchen Ereignissen er aufgerufen werden soll und was er prinzipiell ausführen soll.

|  |  |  |  |
| --- | --- | --- | --- |
| **TriggerNr** | **gehört zu Tabelle** | **auslösendes Ereignis** | **Beschreibung** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Aufgabe 6:** Überlegen Sie, welche Änderungen nötig sind, um das Beispiel aus Aufgabe 3 so zu erweitern, dass nebst der referenziellen Integrität, auch noch die Löschweitergabe funktionieren soll. Beschreiben Sie genau zu welcher Tabelle jeder Trigger gehört, bei welchen Ereignissen er aufgerufen werden soll und was er ausführen soll. Sollte der Trigger keine Veränderung gegenüber der Aufgabe 3 aufweisen, schreiben Sie einfach bei Bemerkungen „unverändert“

|  |  |  |  |
| --- | --- | --- | --- |
| TriggerNr | gehört zu Tabelle | auslösendes Ereignis | Beschreibung |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

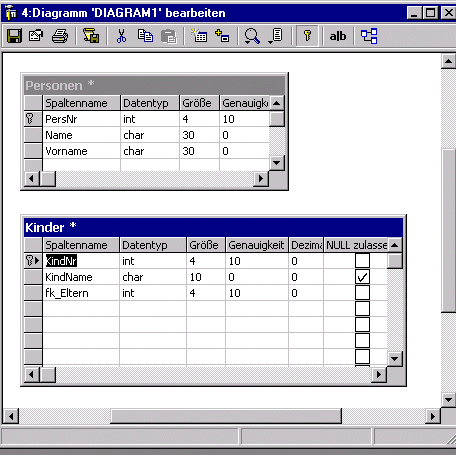
**Aufgabe 7:**  Implementieren Sie die Beziehung Pesonen🡨🡪Kinder inkl. referenzieller Integrität, Löschweitergabe und Aktualisierungsweitergabe mit Trigger.

**Aufgabe 8:** Erstellen Sie eine Datenbank nach dem folgenden ER\_Modell:



Realisieren Sie die Aktualisierungsweitergabe, verbieten Sie aber die Löschweitergabe (mittels Trigger).

**Anhang: Personen-Kind-Beispiel mit Trigger (SQL-Server)**



Es wird kein Foreign-Key definiert.

Das heisst: Das RDBMS nimmt uns per Default keine Hilfe bezüglich Ref. Integ. ab.

Im Schema ist keine Beziehung ersichtlich.

/\*TABELLEN ERSTELLEN\*/

CREATE TABLE Personen (

PersNr int NOT NULL PRIMARY KEY,

Vorname varchar(30) NULL,

)

CREATE TABLE Kinder (

KindNr int NOT NULL PRIMARY KEY,

Name varchar(30) NULL,

fk\_Eltern int NULL

)

CREATE TRIGGER LOESCHMITARBEITER ON PERSONEN

FOR DELETE

AS

BEGIN

update Kinder set fk\_Eltern = NULL where Kinder. fk\_Eltern IN (Select PersNr from deleted)

END

CREATE TRIGGER FREMDSCHLUESSELPRUEFUNG ON KINDER

FOR UPDATE, INSERT

AS

BEGIN

If not exists(Select \* from Personen where PersNr IN (Select fk\_Eltern from inserted))

BEGIN

rollback transaction

raiserror ( "Der Fremdschlüssel verweist auf einen PK, welcher nicht existiert",11,10)

END

END

Ist noch nicht gewährleistet: Man muss noch folgende Situationen per Trigger verbieten:

* Neues Kind einfügen, welches einen ungültigen Fremdschlüssel auf die Personentabelle besitzt.
* Den Fremdschlüssel eines Kindes auf einen ungültigen Wert setzen.

CREATE TRIGGER VERAENDEREMITARBEITERNUMMER ON Personen

FOR UPDATE

AS

BEGIN

if update(PersNr)

begin

update Kinder set Kinder.fk\_Eltern = (Select PersNr from inserted)

from Kinder, deleted, inserted

where Kinder.fk\_Eltern = (Select PersNr from deleted)

end

END

/\*ON DELETE CASCADE 🡺 PER TRIGGER Kinder Löschen\*/

CREATE TRIGGER LOESCHMITARBEITER ON Personen

FOR DELETE

AS

BEGIN

delete from Kinder where fk\_Eltern IN (Select PersNr from deleted)

END

/\*ON DELETE SET NULL 🡺 PER TRIGGER Fremdschlüssel der betroffenen Kinder auf NULL setzen\*/

/\*ON UPDATE CASCADE 🡺 PER TRIGGER Fremdschlüssel der Kinder auf neue PersonenNr setzen\*/

/\*DIE REFERENZIELLE INTEGRITAET\*/

**Variante mit Fremdschlüsseln**([https://www.techonthenet.com/sql\_server/foreign\_keys/foreign\_delete.php](https://docs.microsoft.com/en-us/sql/t-sql/functions/columns-updated-transact-sql) )

## What is a foreign key with Cascade DELETE in SQL Server?

A foreign key with cascade delete means that if a record in the parent table is deleted, then the corresponding records in the child table will automatically be deleted. This is called a cascade delete in SQL Server.

A foreign key with cascade delete can be created using either a CREATE TABLE statement or an ALTER TABLE statement.

## Create Foreign key with cascade delete - Using CREATE TABLE statement

### Syntax

The syntax for creating a foreign key with cascade delete using a CREATE TABLE statement in SQL Server (Transact-SQL) is:

CREATE TABLE child\_table

(

column1 datatype [ NULL | NOT NULL ],

column2 datatype [ NULL | NOT NULL ],

...

CONSTRAINT fk\_name

FOREIGN KEY (child\_col1, child\_col2, ... child\_col\_n)

REFERENCES parent\_table (parent\_col1, parent\_col2, ... parent\_col\_n)

ON DELETE CASCADE

[ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]

);

child\_table

The name of the child table that you wish to create.

column1, column2

The columns that you wish to create in the table. Each column must have a datatype. The column should either be defined as NULL or NOT NULL and if this value is left blank, the database assumes NULL as the default.

fk\_name

The name of the foreign key constraint that you wish to create.

child\_col1, child\_col2, ... child\_col\_n

The columns in child\_table that will reference a primary key in the parent\_table.

parent\_table  
 The name of the parent table whose primary key will be used in the child\_table.

parent\_col1, parent\_col2, ... parent\_col3

The columns that make up the primary key in the parent\_table. The foreign key will enforce a link between this data and the child\_col1, child\_col2, ... child\_col\_n columns in the child\_table.

ON DELETE CASCADE

It specifies that the child data is deleted when the parent data is deleted.

ON UPDATE

Optional. It specifies what to do with the child data when the parent data is updated. You have the options of NO ACTION, CASCADE, SET NULL, or SET DEFAULT.

NO ACTION

It is used in conjunction with ON DELETE or ON UPDATE. It means that no action is performed with the child data when the parent data is deleted or updated.

CASCADE

It is used in conjunction with ON DELETE or ON UPDATE. It means that the child data is either deleted or updated when the parent data is deleted or updated.

SET NULL

It is used in conjunction with ON DELETE or ON UPDATE. It means that the child data is set to NULL when the parent data is deleted or updated.

SET DEFAULT

It is used in conjunction with ON DELETE or ON UPDATE. It means that the child data is set to their default values when the parent data is deleted or updated.

### Example

Let's look at an example of how to create a foreign key with cascade delete in SQL Server (Transact-SQL) using the CREATE TABLE statement.

For example:

CREATE TABLE products

( product\_id INT PRIMARY KEY,

product\_name VARCHAR(50) NOT NULL,

category VARCHAR(25)

);

CREATE TABLE inventory

( inventory\_id INT PRIMARY KEY,

product\_id INT NOT NULL,

quantity INT,

min\_level INT,

max\_level INT,

CONSTRAINT fk\_inv\_product\_id

FOREIGN KEY (product\_id)

REFERENCES products (product\_id)

ON DELETE CASCADE

);

In this foreign key example, we've created our parent table as the products table. The products table has a primary key that consists of the product\_id field.

Next, we've created a second table called inventory that will be the child table in this foreign key with cascade delete example. We have used the CREATE TABLE statement to create a foreign key on the inventory table called fk\_inv\_product\_id. The foreign key establishes a relationship between the product\_id column in the inventory table and the product\_id column in the products table.

For this foreign key, we have specified the ON DELETE CASCADE clause which tells SQL Server to delete the corresponding records in the child table when the data in the parent table is deleted. So in this example, if a product\_id value is deleted from the products table, the corresponding records in the inventory table that use this product\_id will also be deleted.

## Create a foreign key with cascade delete - Using ALTER TABLE statement

### Syntax

The syntax for creating a foreign key with cascade delete using an ALTER TABLE statement in SQL Server (Transact-SQL) is:

ALTER TABLE child\_table

ADD CONSTRAINT fk\_name

FOREIGN KEY (child\_col1, child\_col2, ... child\_col\_n)

REFERENCES parent\_table (parent\_col1, parent\_col2, ... parent\_col\_n)

ON DELETE CASCADE;

child\_table

The name of the child table that you wish to modify.

fk\_name

The name of the foreign key constraint that you wish to create.

child\_col1, child\_col2, ... child\_col\_n

The columns in child\_table that will reference a primary key in the parent\_table.

parent\_table

The name of the parent table whose primary key will be used in the child\_table.

parent\_col1, parent\_col2, ... parent\_col3

The columns that make up the primary key in the parent\_table. The foreign key will enforce a link between this data and the child\_col1, child\_col2, ... child\_col\_n columns in the child\_table.

ON DELETE CASCADE

It specifies that the child data is deleted when the parent data is deleted.

### Example

Let's look at an example of how to create a foreign key with cascade delete in SQL Server (Transact-SQL) using the ALTER TABLE statement.

For example:

ALTER TABLE inventory

ADD CONSTRAINT fk\_inv\_product\_id

FOREIGN KEY (product\_id)

REFERENCES products (product\_id)

ON DELETE CASCADE;

In this foreign key example, we've created a foreign key on the inventory table called fk\_inv\_product\_id that references the products table based on the product\_id field.

For this foreign key, we have specified the ON DELETE CASCADE clause which tells SQL Server to delete the corresponding records in the child table when the data in the parent table is deleted. So in this example, if a product\_id value is deleted from the products table, the corresponding records in the inventory table that use this product\_id will also be deleted.

**Hinweise**

1. Trigger können beim SQL-Server von Microsoft auch auf gespeicherte Abfragen erstellt werden.
2. Ebenso können Trigger auf Logon-Ereignisse erstellt werden.
3. Auflisten aller Trigger einer Datenbank

SELECT

sysobjects.name AS trigger\_name

,USER\_NAME(sysobjects.uid) AS trigger\_owner

,s.name AS table\_schema

,OBJECT\_NAME(parent\_obj) AS table\_name

,OBJECTPROPERTY( id, 'ExecIsUpdateTrigger') AS isupdate

,OBJECTPROPERTY( id, 'ExecIsDeleteTrigger') AS isdelete

,OBJECTPROPERTY( id, 'ExecIsInsertTrigger') AS isinsert

,OBJECTPROPERTY( id, 'ExecIsAfterTrigger') AS isafter

,OBJECTPROPERTY( id, 'ExecIsInsteadOfTrigger') AS isinsteadof

,OBJECTPROPERTY( id, 'ExecIsTriggerDisabled') AS [disabled]

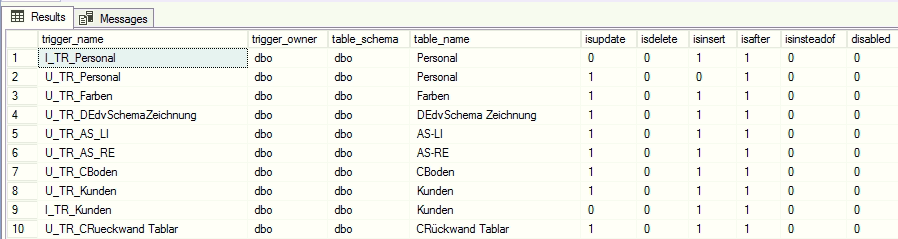
FROM sysobjects

INNER JOIN sysusers ON sysobjects.uid = sysusers.uid

INNER JOIN sys.tables t ON sysobjects.parent\_obj = t.object\_id

INNER JOIN sys.schemas s ON t.schema\_id = s.schema\_id

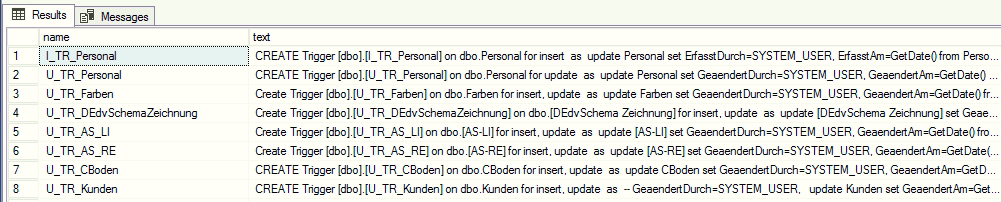
WHERE sysobjects.type = 'TR'



SELECT o.name, c.text

FROM sys.objects AS o

INNER JOIN sys.syscomments AS c ON o.object\_id = c.id WHERE o.[type] = 'TR'



**Tabelle mit vielen Datensätzen erzeugen:**

SELECT TOP 2000000

RowNum = IDENTITY(INT,1,1),

SomeID = ABS(CHECKSUM(NEWID()))%250000+1,

SomeCode = CHAR(ABS(CHECKSUM(NEWID()))%26+65) + CHAR(ABS(CHECKSUM(NEWID()))%26+65)

INTO TestData

FROM Master.dbo.SysColumns t1, Master.dbo.SysColumns t2;

**Videos**

[https://www.linkedin.com/learning/sql-server-2014-developing-databases/understanding-triggers?u=2976210](https://www.techonthenet.com/sql_server/foreign_keys/foreign_delete.php?u=2976210)

[https://www.linkedin.com/learning/sql-server-triggers-stored-procedures-and-functions/using-after-triggers?u=2976210](https://docs.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?u=2976210)

**Links**

[https://docs.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-2017](https://docs.microsoft.com/en-us/sql/t-sql/functions/update-trigger-functions-transact-sql?view=sql-server-2017)

[https://de.wikipedia.org/wiki/Datenbanktrigger](https://www.linkedin.com/learning/sql-server-2014-developing-databases/understanding-triggers)

<https://www.techonthenet.com/sql_server/foreign_keys/foreign_delete.php>

**Anhang: So finden Sie aktualisierte Spalten in SQL Server Trigger**

**Problem**Sie wollen wissen, welche Spalten in einer SQL Server-Tabelle nach der Ausführung einer UPDATE-Anweisung geändert wurden, unabhängig davon, wo die UPDATE-Anweisung ausgeführt wurde. Sie können einen Trigger verwenden, um zu identifizieren, wann ein UPDATE durchgeführt wurde, aber wie können Sie identifizieren, welche Spalten aktualisiert wurden.

Zusätzlich stehen für die eingefügten und gelöschten Tabellen mit DML-Triggern zwei Funktionen zur Verfügung, die Sie im Triggercode aufrufen können. Diese Funktionen sind UPDATE () und COLUMNS\_UPDATED (), die wir unten behandeln werden.

**SQL Server UPDATE() Function für Trigger**

Diese Funktion steht für Trigger zur Verfügung, die als Reaktion auf INSERT- oder UPDATE-Ereignisse ausgelöst werden. Sie erhält einen Spaltennamen als Parameter. Dies muss natürlich eine Spalte sein, die in der zugrunde liegenden Tabelle oder Sicht vorhanden ist. Die Funktion gibt TRUE oder 1 zurück, wenn die angegebene Spalte aktualisiert wurde, andernfalls wird FALSE oder 0 zurückgegeben.

Sehen wir uns ein Beispiel an. Zuerst müssen Sie die Testtabelle erstellen, indem Sie das Skript im nächsten Codeabschnitt ausführen.

CREATE TABLE TestColumns

(

Column\_1 INT PRIMARY KEY,

Column\_2 INT,

Column\_3 INT,

Column\_4 INT )

INSERT INTO dbo.TestColumns ( Column\_1, Column\_2, Column\_3, Column\_4 ) VALUES ( 0, 0, 0, 0 )

Jetzt ist es an der Zeit, einen AFTER UPDATE-Trigger zu erstellen, der eine Nachricht mit dem Namen der Spalte anzeigt, die wir aktualisieren.

CREATE TRIGGER TR\_TestColumns ON dbo.TestColumns

AFTER UPDATE

AS

IF UPDATE(Column\_1)

BEGIN

;THROW 51000, 'You can''t update the primary key', 1;

END

IF UPDATE(Column\_2)

BEGIN

PRINT 'Column\_2 was updated'

END

IF UPDATE(Column\_3)

BEGIN

PRINT 'Column\_3 was updated'

END

IF UPDATE(Column\_4)

BEGIN

PRINT 'Column\_4 was updated'

END

Wie Sie im obigen Code sehen können, gibt es einen IF-Block für jede Spalte. Beachten Sie, dass ich auch einen IF-Block für die Spalte Column\_1 hinzugefügt habe, bei der es sich um den Primärschlüssel der Tabelle handelt. Was wird Ihrer Meinung nach passieren, wenn wir versuchen, diese Spalte zu aktualisieren? Warten wir es ab.

Führen Sie zum Testen der UPDATE () -Funktion die folgenden UPDATE-Anweisungen aus.

UPDATE dbo.TestColumns

SET Column\_1 = 2

WHERE Column\_1 = 0

UPDATE dbo.TestColumns SET Column\_2 = 2

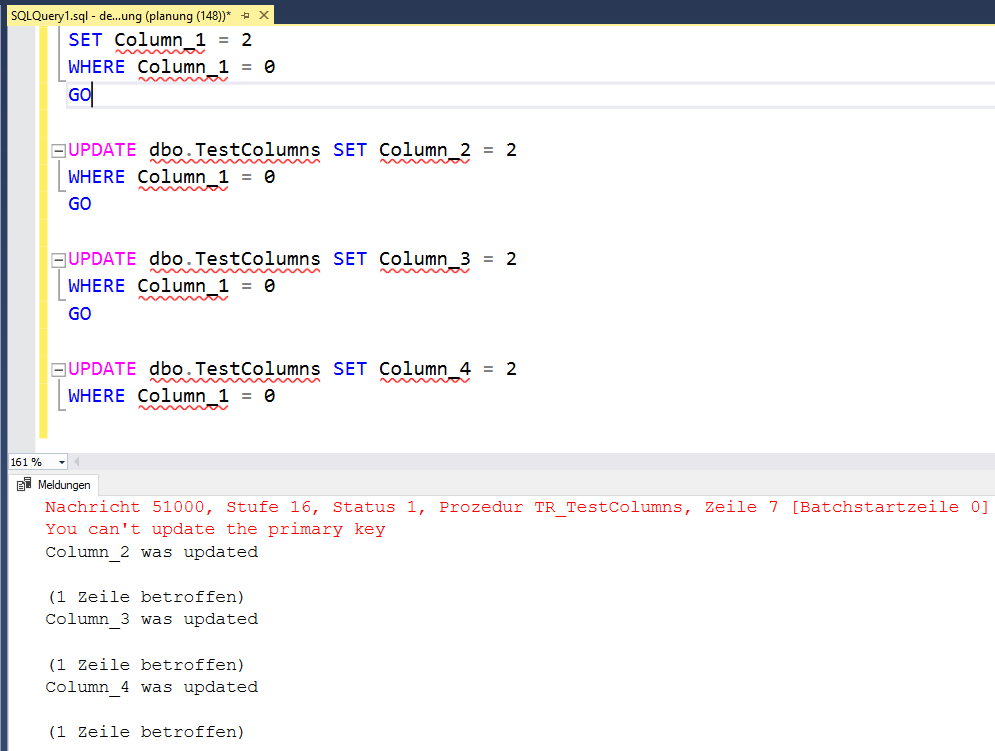
WHERE Column\_1 = 0

UPDATE dbo.TestColumns SET Column\_3 = 2

WHERE Column\_1 = 0

UPDATE dbo.TestColumns SET Column\_4 = 2

WHERE Column\_1 = 0



**SQL Server *COLUMNS\_UPDATED()* Funktion für Trigger**

[https://docs.microsoft.com/en-us/sql/t-sql/functions/columns-updated-transact-sql?view=sql-server-2017](https://www.linkedin.com/learning/sql-server-triggers-stored-procedures-and-functions/using-after-triggers?view=sql-server-2017)

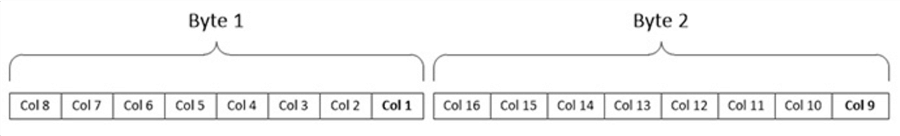
[https://docs.microsoft.com/en-us/sql/t-sql/functions/update-trigger-functions-transact-sql?view=sql-server-2017](https://de.wikipedia.org/wiki/Datenbanktrigger?view=sql-server-2017)

Diese Funktion wird verwendet, um die eingefügten oder aktualisierten Spalten einer Tabelle oder Ansicht zu kennen. Es gibt einen VARBINARY-Stream zurück, mit dem Sie mithilfe einer Bitmaske mehrere Spalten testen können. Der Unterschied zwischen den Funktionen UPDATE() und COLUMNS\_UPDATED() besteht darin, dass Sie mit der Funktion UPDATE() die Funktion COLUMNS\_UPDATED() für eine einzelne Spalte testen können.

Die *Columns\_UPDATED()*-Funktion gibt ein oder mehrere Bytes zurück, abhängig von der Anzahl der Spalten der Basistabelle oder Ansicht, um eine Bitmaske zu entsprechen, wobei jedes Bit eine Spalte darstellt. und wird auf 1 gesetzt, wenn die Spalte aktualisiert oder eingefügt wird, und andernfalls 0. Wie Sie vielleicht wissen, hat jedes Byte 8 Bits, so dass die Länge des VARBINARY-Streams in Bytes die kleinste ganze Zahl grösser als oder gleich der Anzahl der Spalten ist, die durch 8 geteilt sind. Wenn Sie beispielsweise eine Tabelle mit 25 Spalten haben und die Länge des VARBINARY-Streams wissen möchten, der von der Columns\_UPDATED()-Funktion zurückgegeben wird, teilen wir 25 über 8, was 3.125 ist. Schliesslich ist die kleinste ganze Zahl grösser oder gleich 3.125 4.

Die Bytes des VARBINARY-Streams werden von links nach rechts sortiert, wenn Sie also eine Tabelle mit 15 Spalten haben, enthält das erste Byte von links den Bitmaskenwert für die Spalten 1 bis 8 (die ersten acht Spalten). In der Zwischenzeit enthält das zweite Byte von links den Bitmaskenwert für die Spalten 9 bis 15.

Darüber hinaus wird jedes Byte des VARBINARY-Streams von rechts nach links sortiert, sodass das erste Bit von rechts vom ersten Byte beginnend von links der maskierte Wert für die erste Spalte ist. Ich weiss, dass es sehr verwirrend klingt, also habe ich die Grafik unten erstellt, um zu veranschaulichen, wie jede Spalte in einer Tabelle mit dem jeweiligen Bit auf der Bitmaske der COLUMNS\_UPDATED()-Funktion für einen Zwei-Byte-Stream übereinstimmt.



Nun, da wir wissen, wie man die Ausgabedaten der Columns\_UPDATED()-Funktion liest, sehen wir uns ein paar Beispiele an.

Ändern wir den Trigger, den wir frühzeitig erstellt haben, um die COLUMNS\_UPDATED() anstelle der UPDATE()-Funktion zu verwenden.

ALTER TRIGGER TR\_TestColumns ON dbo.TestColumns

AFTER UPDATE

AS

IF COLUMNS\_UPDATED() = 0x01

BEGIN

;THROW 51000, 'You can''t update the primary key', 1;

END

IF COLUMNS\_UPDATED() = 0x02

BEGIN

PRINT 'Column\_2 was updated'

END

IF COLUMNS\_UPDATED() = 0x04

BEGIN

PRINT 'Column\_3 was updated'

END

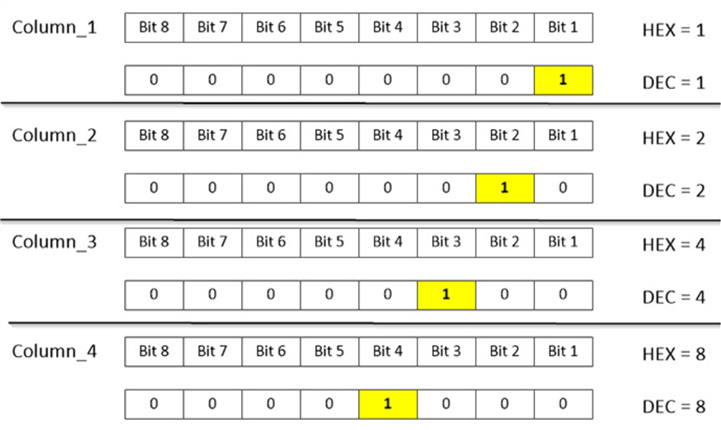
IF COLUMNS\_UPDATED() = 0x08

BEGIN

PRINT 'Column\_4 was updated'

END

Im vorherigen Code können Sie sehen, dass ich die Funktion COLUMNS\_UPDATED() vergleiche, um zu sehen, ob ihr Ergebnis gleich den hexadezimalen Zahlen 0x1, 0x02, 0x04 und 0x08 ist. Diese Zahl stellt die Spalten 1 bis 4 in der Bitmaske dar. Ein sehr häufiger Fehler, wenn Benutzer diese Funktion verwenden, ist, dass sie die hexadezimale Zahl, die das Spaltenbit auf der Bitmaske einschaltet, mit der Spaltennummer in hexadezimal verwechseln. Ich habe die nächste Grafik erstellt, um dies verständlicher zu machen.



Sehen wir uns nun an, wie sich unsere UPDATE-Anweisungen verhalten, wenn wir UPDATE() mit der Funktion COLUMNS\_UPDATED() geändert haben.

UPDATE dbo.TestColumns SET Column\_1 = 2

WHERE Column\_1 = 0

GO

UPDATE dbo.TestColumns SET Column\_2 = 2

WHERE Column\_1 = 0

GO

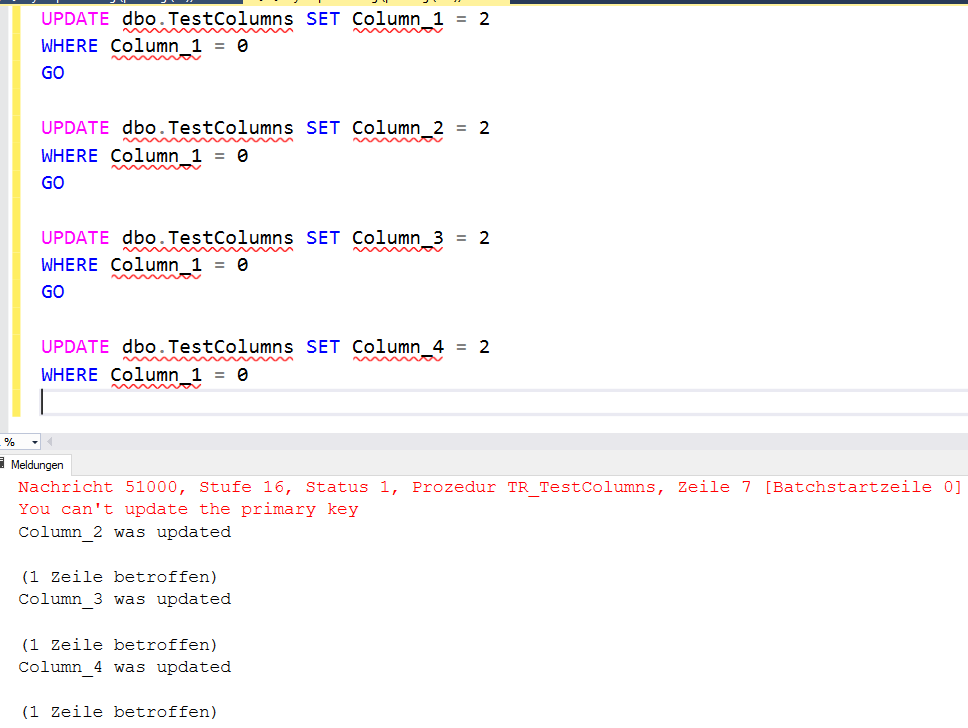
UPDATE dbo.TestColumns SET Column\_3 = 2

WHERE Column\_1 = 0

GO

UPDATE dbo.TestColumns SET Column\_4 = 2

WHERE Column\_1 = 0



Wie Sie auf dem Bild oben sehen können, sind die Ergebnisse, die durch das Ausführen dieser UPDATE-Anweisungen mit der Columns\_UPDATED()-Funktion im Code des Triggers erzielt wurden, die gleichen wie die Update()-Funktion, die das erwartete Verhalten ist.

Mittlerweile haben wir die Funktion COLUMNS\_UPDATED() so verwendet, als wäre es die UPDATE()-Funktion. Im nächsten Beispiel werden wir nach Aktualisierungen für mehr als eine Spalte suchen. Dazu werden wir den Trigger ändern, den wir zuvor erstellt haben. Der nächste Code druckt eine Meldung, wenn die Spalten 2 und 4 in den aktualisierten Spalten enthalten sind und andernfalls fehlschlagen. Sie werden sehen, dass wir den **& bitwise Operator** ([Bitwise Operators (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/bitwise-operators-transact-sql" \t "_blank)) einführen müssen, der ein logisches UND zum Testen auf bestimmte Spaltensätze in der Bitmaske ist.

ALTER TRIGGER TR\_TestColumns ON dbo.TestColumns

AFTER UPDATE

AS

IF COLUMNS\_UPDATED() & CAST(0x01 AS int) = 0x01

BEGIN

;THROW 51000, 'You can''t update the primary key', 1;

END

IF COLUMNS\_UPDATED() & CAST(0x0A AS int) = 0x0A

BEGIN

PRINT 'Column\_4 and Column\_2 were updated'

END ELSE

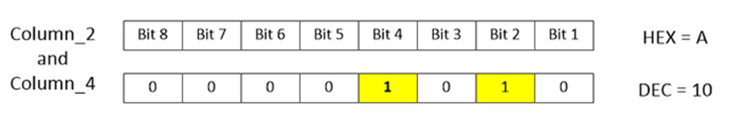
BEGIN

;THROW 51000, 'You tried to update a forbidden column', 1;

END

Im vorherigen Code mit der Verwendung des **& Operators** habe ich ein logisches UND zwischen der COLUMNS\_UPDATED()-Funktion und einer Umwandlung in ganzzahligen Ausdruck über eine hexadezimale Zahl gemacht. Dies liegt daran, dass der bitweise Operator, wenn der linke Operand vom varbinären Datentyp ist, den richtigen Operanden ganzzahligen Typ benötigt. Es gibt keine Notwendigkeit, hexadezimale Zahlen in diesem Fall zu verwenden, aber ich entschied mich, sie anstelle von Dezimalzahlen zu verwenden, weil ich denke, dass es einfacher und intuitiver ist, wenn ich mit Bitmasken arbeite.

Auf dem nächsten Bild befindet sich eine grafische Darstellung eines Bytes mit einer Maske für die Spalten 2 und 4. Diese Maske stellt die binäre Zahl 1010 dar, die in Dezimalnotation 10 und in der hexadezimalen Notation A ist.



Das nächste Testskript versucht, unsere Testtabelle zu aktualisieren. Zuerst versuchen wir, nur Column\_2 zu aktualisieren, die gemäss unserer Trigger-Definition fehlschlagen wird. Dann versuchen wir, Column\_2, Column\_3 und Column\_4 zu aktualisieren. Da unser Trigger erfordert, dass Column\_2 und Column\_4 aktualisiert werden, können wir davon ausgehen, dass diese Abfrage erfolgreich ausgeführt wird, da unser Trigger beim Aktualisieren von Column\_3 keine Einschränkung enthält. Schliesslich versucht die letzte Abfrage, Column\_2 und Column\_4 zu aktualisieren, was unser Trigger natürlich nicht leugnet.

PRINT 'UPDATE Column\_2: '

UPDATE dbo.TestColumns

SET Column\_2 = 2

WHERE Column\_1 = 0

GO

PRINT 'UPDATE Column\_2, Column\_3 and Column\_4: '

UPDATE dbo.TestColumns

SET Column\_2 = 2, Column\_3 = 2, Column\_4 = 2

WHERE Column\_1 = 0

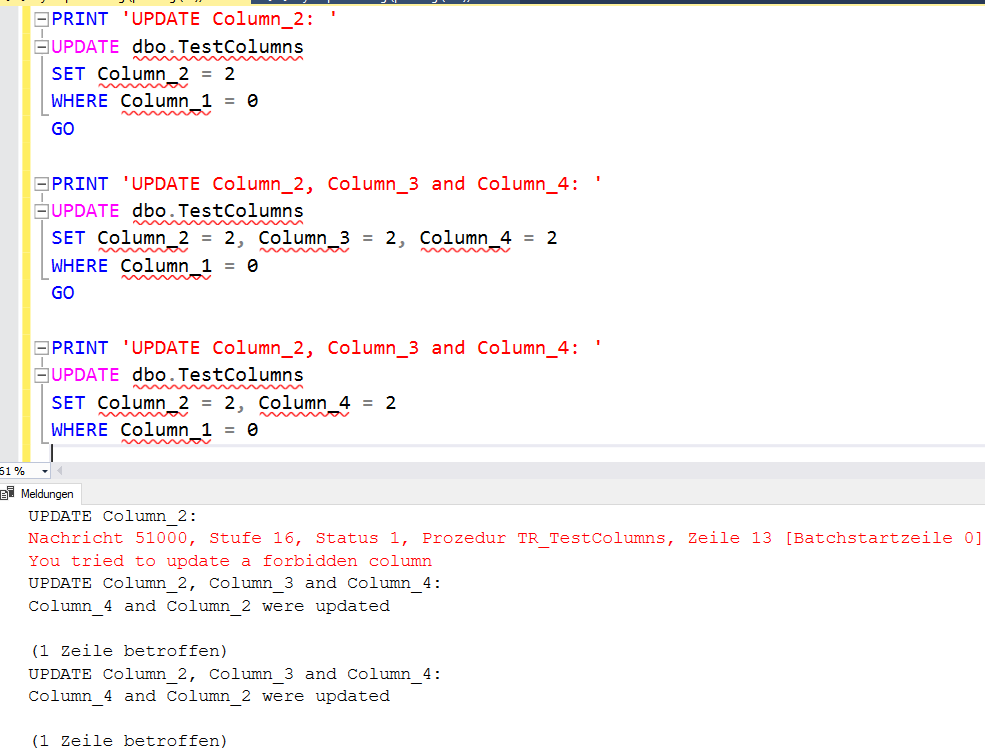
GO

PRINT 'UPDATE Column\_2, Column\_3 and Column\_4: '

UPDATE dbo.TestColumns

SET Column\_2 = 2, Column\_4 = 2

WHERE Column\_1 = 0



Als letztes Beispiel sehen wir ein Beispiel für den Umgang mit einer Tabelle mit mehr als acht Spalten. Dazu müssen wir unsere Beispieltabelle mit 14 Spalten wie folgt löschen und neu erstellen.

DROP TABLE TestColumns

CREATE TABLE TestColumns (

Column\_1 INT PRIMARY KEY,

Column\_2 INT NULL ,

Column\_3 INT NULL ,

Column\_4 INT NULL ,

Column\_5 INT NULL ,

Column\_6 INT NULL ,

Column\_7 INT NULL ,

Column\_8 INT NULL ,

Column\_9 INT NULL ,

Column\_10 INT NULL ,

Column\_11 INT NULL ,

Column\_12 INT NULL ,

Column\_13 INT NULL ,

Column\_14 INT NULL

)

INSERT INTO dbo.TestColumns ( Column\_1, Column\_2, Column\_3, Column\_4, Column\_5, Column\_6, Column\_7,

Column\_8, Column\_9, Column\_10, Column\_11, Column\_12, Column\_13, Column\_14 )

VALUES ( 0, 0, 0, 0, 0, 0, 0,

0, 0, 0, 0, 0, 0, 0 )

Der nächste Trigger druckt eine Meldung, wenn die Spalten Column\_4 und Column\_7 aktualisiert werden und wenn die Spalten Column\_9 und Column\_12 aktualisiert werden.

CREATE TRIGGER TR\_TestColumns ON dbo.TestColumns

AFTER UPDATE

AS

IF SUBSTRING(COLUMNS\_UPDATED(),1,1) & CAST( 0x48 AS INT) = 0x48

BEGIN

PRINT 'Column\_4 and Column\_7 were updated' ;

END

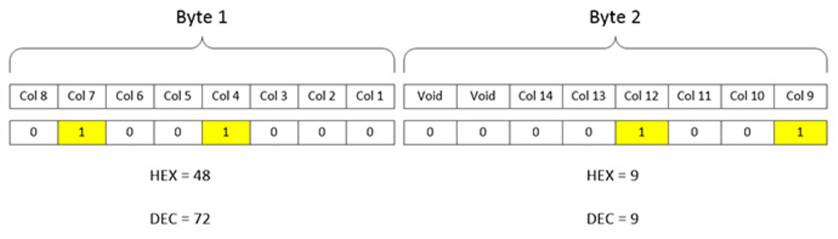
IF SUBSTRING(COLUMNS\_UPDATED(),2,1) & CAST( 0x09 AS INT) = 0x09

BEGIN

PRINT 'Column\_9 and Column\_12 were updated' ;

END

Im nächsten Bild sehen Sie die Bitmaske und ihren numerischen Wert sowohl in hexadezimalen als auch in dezimalen Formaten.



Führen wir die nächsten beiden Updateanweisungen aus und sehen Sie, was passiert.

UPDATE dbo.TestColumns

SET Column\_4 = 2, Column\_7 = 2

WHERE Column\_1 = 0

UPDATE dbo.TestColumns

SET Column\_9 = 2, Column\_12 = 2

WHERE Column\_1 = 0

