ICA | INFORMATICA COMMUNICATIE ACADEMIE

Course
Database Implementation

Information System Engineering

HAN

Demo After Trigger

```
CREATE TABLE dbo.Person (
    PersonID int   CONSTRAINT PK_Person PRIMARY KEY IDENTITY,
    LastName varchar(50) NOT NULL,
    Gender char(1) NOT NULL
        CONSTRAINT CK_Gender CHECK (Gender IN ('M', 'F'))
)
go

CREATE TRIGGER dbo.TriggerOne
ON dbo.Person           -- attached to the table Person
AFTER INSERT, UPDATE    -- fires after these events
AS
BEGIN
    PRINT 'In the After Trigger'
END
go

INSERT dbo.Person (LastName, Gender)
    VALUES ('Ebob', 'M')    -- this event fires the trigger
```

What is a Trigger?

Special stored procedure attached to table events

Can't be directly executed

Part of the statement that fires it

- Extends the duration of a statement, which can lead to locking and blocking problems for high-transaction systems
- Be aware of the potential performance impact
- **IF** the statement is executed, the trigger is **ALWAYS** executed!

Drawbacks of triggers:

- Non standard
- Complex coding
- Enforcing integrity rules is **reactive**: **prevention** is in fact better than reaction

Trigger types SQL Server

DML triggers

- Respond to data changes
- Attached to a table
- Events: **INSERT, DELETE, UPDATE**
- SQL Server triggers fire *once per data-modification*
  - NOT once per affected row
- **AFTER** triggers (used in course DI)
  - Fires *after* the triggering event
- **INSTEAD OF** triggers
  - Fires *instead of* the triggering event

DDL triggers

- Respond to schema changes
- Events: **CREATE, ALTER, DROP, GRANT, DENY, REVOKE**

What's a After Trigger used for?

Complex data validation

Enforcing complex business rules

Writing data-audit trails

Maintaining modified data columns

Enforcing custom referential integrity checks and cascading deletes

T-SQL After Triggers

Simplified syntax:

```
CREATE TRIGGER schema.trigger_name
ON schema.table_name
{FOR | AFTER} {INSERT, UPDATE, DELETE}
AS
Trigger Code

ALTER TRIGGER trigger_name
-- etc

DROP Trigger trigger_name
```

SQL Server triggers fire once per data-modification.

Exercise 1

```
ALTER TRIGGER dbo.TriggerOne
ON dbo.Person
AFTER INSERT, UPDATE
AS
  PRINT '1, 2 or many inserts/updates'
go

-- 2 records in ONE insert:
INSERT dbo.Person (LastName, Gender)
    VALUES ('Ebob', 'M'), ('Johnson', 'F')
-- or:
INSERT dbo.Person (LastName, Gender)
SELECT 'Carter', 'M'
UNION ALL
SELECT 'Adams', 'F'
```

SQL Server triggers fire once per data-modification.

Exercise 2

```
-- create a helper table AZ:
CREATE TABLE AZ (col CHAR(1))
go
INSERT AZ VALUES
       ('A'),('B'),('C'),('D'),('E'),('F'),('G'),('H')
go

-- more than 16 million records in ONE insert:
INSERT dbo.Person (LastName, Gender)
SELECT a.col +b.col +c.col +d.col +e.col +f.col +g.col +h.col,
   CASE a.col WHEN 'A' THEN 'M' ELSE 'F' END
FROM AZ a, AZ b, AZ c, AZ d, AZ e, AZ f, AZ g, AZ h
-- actually all CROSS JOINs
```

Tables *Inserted* and *Deleted*

DML Trigger can read the *before* and *after images* of the affected rows

- trigger can make comparisons, calculations, and (if necessary) undo the changes.

***Logical* tables `Inserted` and `Deleted`**

- ***identical*** in structure to the table on which the trigger is defined
- **`Deleted`** holds the *before* images of the data
- **`Inserted`** holds the *after* images of the data
- Scope is limited

Inserted and Deleted tables

	`Inserted` table	`Deleted` table
`INSERT`	Rows being inserted	Empty
`UPDATE`	Rows after the update	Rows before the update
`DELETE`	Empty	Rows being deleted

Execution order After Triggers

1. Check of declarative constraints
2. Execution of DELETE/INSERT/UPDATE
3. Population of *Inserted* and *Deleted* tables
4. Execution of after trigger(s) -- in no specific order!
5. Finish statement

After Trigger Example

Exercise 3

```
ALTER TRIGGER dbo.TriggerOne
ON dbo.Person
AFTER UPDATE
AS
BEGIN
  SET NOCOUNT ON
  IF UPDATE(LastName)
    SELECT 'You modified the LastName column to ' + Inserted.LastName
    FROM Inserted
END
-- IF UPDATE() generally is used to execute data checks only when
needed (only in triggers)
GO
UPDATE Person
  SET LastName = 'Nielsen'
  WHERE PersonID = 32 OR PersonID = 33
```

Exercise 4

Alter the trigger so that message is:

‘You modified the LastName column of person with PersonId 32 to Nielsen’
‘You modified the LastName column of person with PersonId 33 to Nielsen’

Exercise 5

```
ALTER TABLE dbo.Person
  ADD FatherID INT NULL
CONSTRAINT FK_Father FOREIGN KEY REFERENCES Person(PersonID)
```

Business Rule:

Each father should be male

- Write an after trigger on Person that ensures this.
- Test the trigger.

Heading:

```
CREATE TRIGGER dbo.PersonParentsInsUpdTrg
ON dbo.Person
AFTER INSERT, UPDATE
AS
```

Trigger coding Best Practice

Every trigger must be written to handle DML statements that affect multiple rows

The best way to deal with multiple rows is to work with the `inserted` and `deleted` tables with set-oriented operations

When a trigger is used for implementing a complex businnes rule it should undo the data modification by:

- using a ROLLBACK of the TRANSACTION or

- using a THROW statement (this will automatically rollback a transaction, if there is any active transaction

>> more about transactions to come.....

Exercise 6: Trigger template

Give a trigger template, using TRY/CATCH

ICA | INFORMATICA COMMUNICATIE ACADEMIE

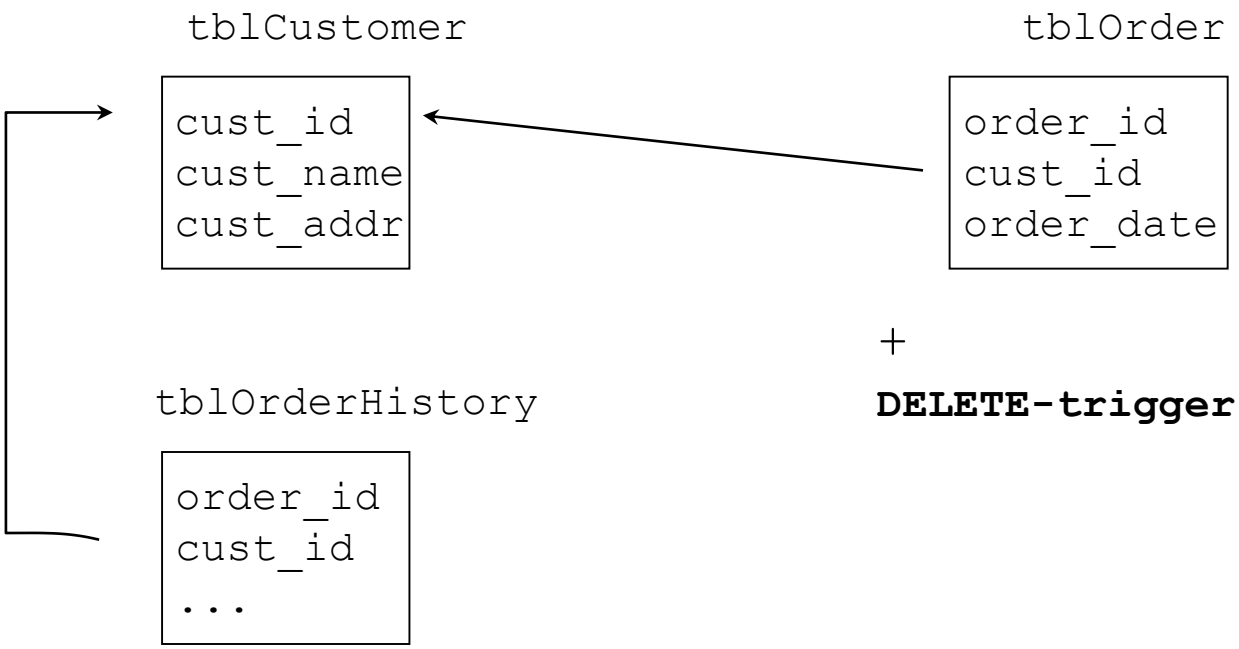
Exercise 7

Studiehandleiding Thema Triggers
Pagina 32
Opdrachten 1 t/m 5

HAN

History tables

When a record is deleted from tblOrder some of its data is inserted in OrderHistory by the DELETE-trigger of tblOrder



Exercise 8: Audit trigger

```
CREATE TABLE ActivityLog (
act_id   int IDENTITY,
logdate  datetime DEFAULT (GETDATE()),
username varchar(25) DEFAULT (USER_NAME()),
note     varchar(50)
)
```

Write a trigger logging any change of the level of complexity of a music piece to `ActivityLog`:

- use `update(niveaucode)` to test if the column `niveaucode` has been "touched"
- Use `'Value ' + the old value + ' to '+ the new value` as the value to insert in the `note` column

Triggers for Referential Integrity

SQL Server 2012 does support declarative cascading RI

But in some situations declarative cascading won't work!

In case of cyclic dependencies:

Self-referencing tables

When there is more than one foreign key between two tables

DRI across databases

Several triggers needed to implement a relationship between a Parent and Child table

- Parent: UPDATE, DELETE after trigger
- Child: INSERT, UPDATE after trigger

Cyclic dependency

```
CREATE TABLE Employee (empId int IDENTITY
CONSTRAINT PK_Employee PRIMARY KEY,
  lastName varchar(25) NOT NULL,
  managerEmpId int    NOT NULL,
CONSTRAINT FK_manager FOREIGN KEY (managerEmpId) REFERENCES
    Employee(empId)
ON UPDATE CASCADE ON DELETE CASCADE)
```

Result of CREATE:

```
Msg 1785, Level 16, State 1, Line 1
Introducing FOREIGN KEY constraint 'FK_manager' on table
'Employee' may cause cycles or multiple cascade paths.
Specify ON DELETE NO ACTION or ON UPDATE NO ACTION,
or modify other FOREIGN KEY constraints.
Msg 1750, Level 16, State 1, Line 1
Could not create constraint. See previous errors.
```

Solution: use triggers for one of the relationships

Exercise 9: RI across databases

1. **Use the DDL script of the Muziekdatabase**
2. **Create database `dbStuk` and `dbBezettingsregel`**
3. **Create tables `Muziekschool`, `Componist`, `Niveau`, `Genre`, `Stuk` in `dbStuk` with all constraints, except:**
   - the FK from Stuk to Stuk.
4. **Create tables `Bezettingsregel`, `Instrument` in `dbBezettingsregel` with all constraints, except:**
   - The declarative RI between **`Bezettingsregel`** and **`Stuk`**
5. **Prevent the deletion of rows in parent `Stuk` from happening when a child row exists in `Bezettingsregel`**
   - Give an after trigger on **`Stuk`**

Veel voorkomende fouten

Geen rekening gehouden met multiple inserts/deletes/updates

Overbodig gebruik van variabelen

Slechte foutafhandeling

Geen gebruik van performance optimalisaties:

    @@ROWCOUNT, NOCOUNT, UPDATE()

Geen RETURN op de juiste plek

The impedance mismatch

Row based vs Set Based Programming

- SQL is a declarative set based programming language
  - The SQL Server itself understands and handles only SQL
- T-SQL extends the declarative set based SQL language with procedural constructs
  - The T part of T-SQL in a way is *wrapped around* SQL

T-SQL = Declarative SQL + Procedural Constructs

- If used together in a smart way it provides tremendous extra power, if used in a dumb way it just hampers (one real dumb thing would be trying to avoid the use of SQL!)
  - The world of rowsets (SQL) and scalars (procedural languages) do not naturally fit together
    - ***This is called the impedance mismatch***

The impedance mismatch

Example SET *coincidentally matching* one SCALAR:

```
DECLARE @scalar NUMERIC(4,0)
SELECT @scalar = jaartal
    FROM dbo.Stuk
    WHERE stuknr = 2 -- note: stuknr is pk!
```

Example SET *not matching* one SCALAR:

```
DECLARE @scalar NUMERIC(4,0)
SELECT @scalar = jaartal
    FROM dbo.Stuk
    WHERE stuknr > 2
```

What would be the value stored in `@scalar`?

Triggers and the Set Based / Row Based Bias

New Business Rule:
A 'stuk' must have a duration of less than 10 minutes

```
CREATE TRIGGER STUK_Ins_Upd ON dbo.Stuk
AFTER INSERT, UPDATE
AS
DECLARE @duration NUMERIC(3,1)
SELECT @duration = speelduur FROM Inserted  -- WRONG!!
IF @duration > 10.0
    THROW 50000,'No pieces allowed exceeding 10 minutes.',1
```

(maar veel beter met een CHECK!!)

Triggers and the Set Based / Row Based Bias

Let's update *a single row* in the database

```
BEGIN TRANSACTION -- komt volgende week
UPDATE dbo.Stuk
SET speelduur = 2 * speelduur -- violation of the business rule!
WHERE Stuknr = 13

SELECT * FROM dbo.Stuk WHERE speelduur >= 10
ROLLBACK TRANSACTION -- komt volgende week
```

Messages:

```
Msg 50000, Level 16, State 1, Procedure STUK_Ins_Upd, Line 7 [Batch Start Line 0]
No pieces allowed exceeding 10 minutes.
```

This is the required behaviour, so everything looks OK, doesn't it (?)

Triggers and the Set Based / Row Based Bias

Exercise 10

But now let's update *a set of rows* in the database

```
BEGIN TRANSACTION -- komt volgende blok
UPDATE dbo.Stuk
SET speelduur = 2 * speelduur -- violation of the business rule!

SELECT * FROM dbo.Stuk WHERE speelduur >= 10
ROLLBACK TRANSACTION
```

Results pane of Management Studio:

```
12  9   I'll never go   10   popA   12.0   1996
13  10  Swinging Lina   5jazz   B   16.0   1997
```

This is not the required behavior! So slightly changing the SQL code causes the trigger to just allow the pass of a number of invalid rows into our database!

Triggers and the Set Based / Row Based Bias

What's the value stored in variable `@duration` ?

```
CREATE TRIGGER STUK_Ins_Upd ON dbo.Stuk
AFTER INSERT, UPDATE
AS
DECLARE @duration NUMERIC(3,1)
SELECT @duration = speelduur FROM Inserted
IF @duration > 10.0
    THROW 50000,'No pieces allowed exceeding 10 minutes.',1
```

That value would be the last speelduur value in the INSERTED virtual table which in our case would be the last row of dbo.Stuk

Solution of the Set Based / Row Based Bias

Exercise 11

Change trigger code to *declarative set based* code

```
CREATE TRIGGER STUK_I_U ON dbo.STUK
AFTER UPDATE, INSERT
AS
IF EXISTS (SELECT 'Violation at hand'
           FROM Inserted
           WHERE speelduur > 10.0)
    -- checking existence of invalid rows!
    BEGIN
        THROW 50000, 'No pieces allowed over 10 minutes long', 1
    END
```

This would do the trick *and* is the very best solution (why?).

Differences Set vs Row Based Programming

Retrieve all music pieces with level indication B

A *smart* declarative SET based solution:

```
SELECT stuknr, titel
    FROM dbo.Stuk
    WHERE niveaucode = 'B'
```

Declarative: because you don’t specify how the result is to be found, but just what condition the elements in the result set should meet (niveaucode = 'B')

Differences Set vs Row Based Programming

Exercise 12: Execute

A *stupid* procedural row based solution using an SQL CURSOR:

```
DECLARE Stupid CURSOR
FOR  SELECT stuknr, titel, niveaucode FROM dbo.Stuk
OPEN Stupid

DECLARE @stuknr numeric(5,0), @titel varchar(5),
 @niveaucode char(1)

FETCH Stupid INTO @stuknr, @titel, @niveaucode
WHILE @@FETCH_STATUS = 0
    BEGIN
        IF @niveaucode = 'B'
        PRINT CAST(@stuknr AS varchar(5)) + ' ' + @titel
        FETCH Stupid INTO @stuknr, @titel, @niveaucode
    END
CLOSE Stupid
DEALLOCATE Stupid
```

Differences Set vs Row Based Programming

If possible write declarative code

- using cursors in a relational environment is like “swearing in church!”

Now some solid arguments:

- a cursor is a performance killer !
- we have seen examples where depending on the kind of query the code was functionally correct when handling one row targeting queries, but plain wrong when handling multiple row queries
- never use procedural code if you have a choice, declarative code performs better and provides the functionality handling a far wider scope of scenarios (single and multi row queries targeting your tables)

Exercise 13: Use cursors to join

This exercise is meant to heal you from any procedural disease you may still be suffering of!

- Never again do what you are asked to do in this exercise because *we (ALL teachers of ISE DI) will haunt you till the end of days* :-))

Write a procedural INNER JOIN using two nested cursors to join

- the music pieces (table **`Stuk`**, columns **`stuknr`**, **`title`**)
- and their line-up (table **`Bezettingsregel`**, columns **`instrumentnaam`**, **`toonhoogte`**, **`aantal`**)
- use a variable of type **`TABLE`** to show the results
- run the query and try to compare its speed with an SQL version of the join