Innehåll Programability Del 2

- ✓ Transaktioner
- ✓ Returvärden och OUTPUT
- ✓ Dynamisk SQL
- ✓ CURSOR / FECTH
- ✓ Trigger

Chapter 8, 10 och 13.

Beginning SQL Server 2008 for Developers

Transaktioner (1 av 3)

En transaktion kan sägas vara när en förändring ska genomföras. Den kan bestå av en sats eller flera satser.

- En ensats transaktion kan bara lyckas eller misslyckas. Implicit transaktion.
- En flersats transaktion kan lyckas helt, delvis lyckas (=delvis misslyckas) och misslyckas helt. Explicit transaktion.

Antag att en ny fakturarad ska skapas. Då ska en ny post läggas in i fakturarad och en avräkning av antal i artikel äga rum:

```
INSERT INTO Fakturarad Fakturaid,Artikelid,Antal,Pris,rabatt,momsid)
VALUES (2,104,23,12.50,0,1);

UPDATE Artikel Set Antal=Antal-23
WHERE Artikelid=104;
```

Om ett fel uppstår efter att första satsen genomförts så att den andra satsen inte kan genomföras så har vi inte en konsistent databas längre. Det ska inte få hända. Data kommer inte längre vara korrekta i databasen.

Transaktioner (2 av 3)

- En transaktion är en odelbar arbetsuppgift där allt eller inget ska genomföras.
- Som standard är varje SQL sats en egen arbetsuppgift.
- Om du har fler SQL satser som måste genomföras så ska dessa grupperas att tillhöra **EN** transaktion.

```
BEGIN TRANSACTION                -- Anger start av transaktion

COMMIT TRANSACTION               -- Avslutar lyckad transaktion och
                                 -- sparar till disk.

ROLLBACK TRANSACTION             -- Återställer misslyckad transaktion
```

Använd TRY / CATCH för felhantering.

Transaktioner (3 av 3)

Om något går fel efter BEGIN TRY innan COMMIT TRAN så genomförs BEGIN CATCH. Om allt går bra genomförs COMMIT TRAN

```
BEGIN TRY
  BEGIN TRAN

    UPDATE Artikel
    SET antal=antal-23                    -- Uppdaterar i Artikel
    WHERE artikelid=104;                  -- Lägger upp en post i
                                          -- fakturarad
    INSERT INTO Fakturarad (Fakturaid,Artikelid,Antal,Pris,rabatt,momsid)
    VALUES (2,104,23,6.50,0,1);
  COMMIT TRAN                             -- Avslutar TRAN och sparar på
                                          -- disk.
END TRY

BEGIN CATCH
  ROLLBACK TRAN                           -- Återställer till status
                                          -- innan BEGIN TRAN
  RAISERROR('Transaktionen gick inte att genomföra!',16,1)
END CATCH
```

Ny fakturarad

Lägga till en ny fakturarad. Hämta data ur artikel och uppdatera artikel.

```
CREATE PROCEDURE usp_InsFakturaRad
@Fakturaid int,
@Artikelid int,
@Antal int,
@Rabatt Decimal(2,2),
@Momsid int
AS
BEGIN
     DECLARE @Pris Decimal(6,2);
     SET @Pris=(SELECT Pris From Artikel WHERE Artikelid=@Artikelid);
     BEGIN TRAN
        INSERT INTO Fakturarad (Fakturaid,Artikelid,Antal,Pris,rabatt,momsid)
        VALUES (@Fakturaid,@Artikelid,@Antal,@Pris,@Rabatt,@Momsid);

        UPDATE Artikel Set Antal=Antal-@antal
        WHERE Artikelid=@Artikelid;
     COMMIT TRAN
END
GO
```

OBS!
Förenklad på transaktionssidan av utrymmesskäl

OBS!
Decimaltal anges med decimalpunkt

```
EXEC usp_InsFakturarad 6,104,2,0.05,1
```

Exempel RaderaFakturarad

```
CREATE PROCEDURE usp_FakturaRadRadera
@Fakturaradid int
AS
BEGIN
    BEGIN TRY
        DECLARE @Artid int, @Antal int
        SELECT @Artid = Artikelid, @Antal=Antal
        From Fakturarad
        WHERE Fakturaradid=@Fakturaradid;
        BEGIN TRAN
            UPDATE Artikel Set Antal=Antal+@antal
            WHERE Artikelid=@Artid;
            DELETE From Fakturarad
            WHERE fakturaradid=@fakturaradid;
        COMMIT TRAN
    END TRY

    BEGIN CATCH
        ROLLBACK TRAN
        RAISERROR ('Borttagningen gick inte att genomföra!',16,1)
    END CATCH
END
```

```
EXEC usp_FakturaRadRadera 6
```

Att radera en hel faktura

Beskriv i ett dataflöde hur du ska gå tillväga för att radera en hel faktura!

Parameter OUTPUT

De parametrar vi hittills behandlat är INPUT parametrar. De för värden till proceduren (förutom return) och tas sedan om hand av den lagrade proceduren.

Om en procedur ska lämna från sig värden så ska dessa definieras som OUTPUT:

```
CREATE PROCEDURE usp_MaxAntal (@max integer OUTPUT)
AS
SELECT @max=max(Antal)
FROM Artikel;
```

OUTPUT anges

Deklarera lokal variabel. OUTPUT måste anropas med en variabel

```
DECLARE @MycketAntal as integer

EXEC usp_MaxAntal @MycketAntal OUTPUT

SELECT @MycketAntal as MaxAntal;
```

OUTPUT måste anges vid anrop

Resultatet

RETUR värden

En funktion levererar alltid returvärden i SQL Server. Lagrade procedurer kan också returnera värden med hjälp av RETURN.

Användbarheten är dock begränsad:

- ✓ Returvärdet kan endast vara ett heltal
- ✓ Värden -1 till -99 är reserverade för SQL Server
- ✓ Konvention: 0 = lyckad operation, -100 eller lägre = Fel

```
CREATE PROCEDURE usp_GetAntalFakturaKund
@Kundid int = Null
AS
IF @Kundid is Null
    RETURN -100

DECLARE @Antalet int
SET @Antalet=0

SELECT @Antalet=count(*)
From Faktura
WHERE Kundid=@Kundid;

RETURN @Antalet
```

RETURN orsakar uthopp ur sproc

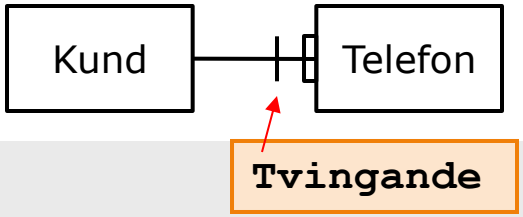
```
DECLARE @retval int

EXEC @Retval= usp_GetAntalFakturaKund 1

IF @Retval=-100
    RAISERROR ('Data saknas !',16,1)
ELSE
    SELECT @Retval as 'Antal fakturor!'
```

Ny kund med en telefon - @@IDENTITY

En kund ska läggas till och dessutom en telefon på kunden. Vi behöver då veta kundid innan vi kan lägga till telefonen:
Senaste IDENTITY värde för sessionen finns i @@IDENTITY




```
CREATE PROCEDURE usp_NewKund
@Namn Varchar(50),
@Telefon Varchar(20),
@Teltypdid int
AS
BEGIN
   INSERT INTO KUND (Namn)
   VALUES (@namn);

   INSERT INTO Telefon (Telenr, Kundid, Teletypid)
   VALUES (@Telefon, @@IDENTITY, @Teltypid);
END
```

Lägger till kunden som får ett nytt Kundid (Identity)

@@IDENTITY

@@IDENTITY innehåller senaste IDENTITY-värde som är Kundens. Kolla också SCOPE_IDENTITY och IDENT_CURRENT

```
EXEC usp_NewKund 'Färgbutiken', '0480-987654',1
```

@@IDENTITY

@@IDENTITY	Innehåller sista inlagda identity värdet i session. Session omfattar också underanrop med batch eller trigger.
SCOPE_IDENTITY	Innehåller sista identity värdet för aktuellt scope. Ett scope är exempelvis den lagrade proceduren. Ett scope omfattar inte underanrop eller trigger.
IDENT_CURRENT	innehåller sista identity värdet i tabellen – oavsett vem/vad som lagt dit den. Syntax: IDENT_CURRENT('table_name')

SCOPE_IDENTITY, IDENT_CURRENT, and @@IDENTITY are similar functions because they return values that are inserted into identity columns.
IDENT_CURRENT is not limited by scope and session; it is limited to a specified table. IDENT_CURRENT returns the value generated for a specific table in any session and any scope. For more information, see IDENT_CURRENT (Transact-SQL).
SCOPE_IDENTITY and @@IDENTITY return the last identity values that are generated in any table in the current session. However, SCOPE_IDENTITY returns values inserted only within the current scope; @@IDENTITY is not limited to a specific scope.

Dymanisk SQL

Prestanda –
Ej kompilerad

Du kan skapa SQL-satsen och sedan köra den utan att ha en lagrad procedur.

```
DECLARE @Artid Int
SET @Artid=1001
EXEC ('SELECT * FROM Artikel WHERE Artid = ' + @Artid)
```

Results | Messages

	Artikelid	Artikelnamn	Antal	Pris	Plats	LargerKostnad
1	1001	Bildskärm platt 17 tum	25	150.00	Hylla 12	3750.00

Byt ut EXEC mot PRINT så får du utskrivet exakt hur din SQL-sats ser ut:

```
DECLARE @Artid Int
SET @Artid=1001
PRINT ('SELECT * FROM Artikel WHERE Artid =' + str(@Artid))
```

Messages

```
SELECT * FROM Artikel WHERE Artikelid =      1001
```

```
DECLARE @Artnamn varchar(30)
SET @Artidnamn='DVD'
PRINT ('SELECT * FROM Artikel
        WHERE Artikelnamn LIKE  "%' + @Artnamn + '%"')
```

Messages

```
SELECT * FROM Artikel WHERE Artikelnamn LIKE "%DVD%"
```

Med PRINT så får du en bra kontroll på hur din SQL-sats verkligen ser ut.
OBS! Det som skickas in till PRINT måste vara en sträng. Därför är Artid ovan konverterad till en sträng.

FETCH Hämta data löpande / Sekventiellt

För att göra en radvis bearbetning av data från en eller flera tabeller så använder du följande kommandon och ordning

1.	DECLARE	Skapa en cursor på servern
2.	OPEN	Skapa ett resultatset i *tempdb*
3.	FETCH	Hämta en rad från cursorpositionen
4.	CLOSE	Ta bort resultat-setet från *tempdb*
5.	DEALLOCATE	Ta bort cursor-definitionen från servern

En cursor är en öppning till tabell/tabeller via en select (DECLARE). Den används sedan för att öppna data, ett resultatset som du kan behandla (OPEN).

FETCH används för att hämta rad för rad som du vill behandla via din kod.

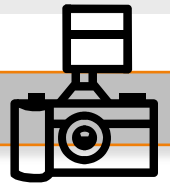
När du är klar stänger du resultatsetet med CLOSE och slutligen tar du bort cursor med DEALLOCATE.

CURSOR

Se books on line

Syntax:

```
DECLARE CursorName [INSENSITIVE] [SCROLL] CURSOR
FOR select_statement
[FOR {READ ONLY | UPDATE [OF column_name [,...n]]}]
```



Parameter	Innebär
INSENSITIVE	Skapar resultat-set i tempdb Stöder inte uppdateringar Ser inte uppdateringar, inserts och deletes gjorda av andra
Inte INSENSITIVE	Skapar kopia av resultatsetets nycklar i tempdb Stöder uppdateringar Ser uppdateringar och deletes gjorda av andra Ser inte inserts gjorda av andra
SCROLL	tillåter läsning framåt och bakåt Skapar en kopia av resultatsetets nycklar i tempdb
FOR READ ONLY och FOR UPDATE	Kontrollerar vilka rader som kan uppdateras Använd READ ONLY för att minimera onödiga låsningar

OPEN

Se books on line

När cursorn öppnas skapas ett resultset i tempdb. Antalet rader du har fått i resultatsetet kan du kontrollera med : @@CURSOR_ROWS

Om du har ett värde <0 (negativt) på @@CURSOR_ROWS anger det att data fylls på asynkront. Asynkron påfyllning av cursors kan sättas med server-parametern Cursor Threshold. (Se books on line)

```
DECLARE myCursor INSENSITIVE CURSOR                -- Deklarerar Cursor
FOR SELECT Kundid, Namn FROM Kund                  -- med innehåll
FOR READ ONLY;                                     -- Endast för läsning

OPEN myCursor                                      -- Öppnar Cursor (resultatset)
DECLARE @KundID int, @Namn nvarchar(30)
FETCH myCursor INTO @KundID, @Namn                 -- Hämtar första posten
    :
    :
CLOSE myCursor                                     -- Stäng Cursor (resultatset)
```

FETCH

Se books on line

Med hjälp av FETCH hämtar vi en ny rad (post) från den position cursorn befinner sig.

```
DECLARE @KundID int, @Namn varchar(30)
FETCH myCursor INTO @KundID, @Namn                    -- Hämtar  data
FETCH NEXT FROM myCursor INTO @KundID, @Namn          -- Hämtar  data
```

- ✓ Cursors deklarerade med SCROLL kan använda NEXT, PRIOR, FIRST, LAST och RELATIVE.
  RELATIVE betyder +/- n rader från aktuell position
  ABSOLUTE betyder +n rader från början eller -n rader från slutet

- ✓ INTO definierar en lista med lokala variabler där den hämtade posten kan läggas

Efter varje FETCH innehåller den globala variabeln @@FETCH_STATUS väsentlig statusinformation

Värde	Beskrivning
0	Success
-1	Vid slutet av record setet
-2	Raden har raderats

FETCH Hämta data löpande forts…

```
DECLARE @iNr int, @Kund varchar(1000), @radAntal int
SET @iNr = 0
SET @Kund = ''

DECLARE myCursor INSENSITIVE CURSOR                 -- Deklarerar Cursor
FOR SELECT Kundid, Namn FROM Kund                   -- med innehåll
FOR READ ONLY;                                      -- Endast för läsning

OPEN myCursor                                       -- Öppnar Cursor (resultatset)
DECLARE @KundID int, @Namn nvarchar(30)             -- variabler
FETCH myCursor INTO @KundID, @Namn                  -- Hämtar första posten
WHILE @@FETCH_STATUS = 0                            -- Så länge det finns poster
    BEGIN
        SET @iNr=@iNr+1                             -- Behandlar data
        SET @Kund=@Kund +';'+@Namn
        FETCH myCursor INTO @KundID, @Namn          -- Hämtar nästa post
    END

CLOSE myCursor                                      -- Stäng Cursor (resultatset)
DEALLOCATE myCursor                                 -- Tar bort deklarationen
SELECT 'Antal är : ' + str(@iNr) as Antalet         -- Skriver ut
SELECT 'Kunderna är : ' + @Kund as Kunderna         -- Skriver ut
```

CURSOR Exempel

```
DECLARE @iNr int, @Kund varchar(1000), @radAntal int
SET @iNr = 0
SET @Kund = ''

DECLARE myCursor SCROLL CURSOR                    -- Deklarerar Cursor
FOR SELECT Kundid, Namn
    FROM Kund                                     -- med innehåll
FOR UPDATE;                                       -- För uppdatering

OPEN myCursor                                     -- Öppnar Cursor (resultatset)
DECLARE @KundID int, @Namn nvarchar(30)
FETCH myCursor INTO @KundID, @Namn                -- Hämtar första posten
SET @radAntal = @@CURSOR_ROWS                     -- Antalet rader finns här
WHILE @@FETCH_STATUS = 0                          -- Så länge det finns poster
    BEGIN
        UPDATE Kund SET Rabatt=0                  -- Uppdaterar posten
        WHERE CURRENT OF myCursor;                -- OBS! WHERE annars alla
        FETCH myCursor INTO @KundID, @Namn        -- Hämta nästa post
    END
CLOSE myCursor                                    -- Stäng Cursor (resultatset)
DEALLOCATE myCursor                               -- Tar bort deklarationen
```

Exempel....

Läs alla rader från cursorn myCursor och anropar en lagrad procedur för varje rad.

```
FETCH myCursor INTO @EmployeeID, @LastName
WHILE @@FETCH_STATUS = 0
BEGIN
        EXEC DoTax @EmployeeID, @LastName
        FETCH myCursor INTO @EmployeeID, @LastName
END
```

Läs varannan rad från cursorn myCursor och anropar en lagrad procedur för varje rad som läses.

```
FETCH  ABSOLUTE 2  FROM myCursor                    -- läser andra raden i resultatset
INTO @EmployeeID, @LastName
WHILE @@FETCH_STATUS = 0
BEGIN
        EXEC DoTax @EmployeeID, @LastName
        FETCH  RELATIVE 2  FROM myCursor            -- läser två rader fram  i resultatset
        INTO @EmployeeID, @LastName
END
```

Trigger - vad är

- En trigger är en variant av en lagrad procedur- Skillnaden är att en procedur startar när någon anropar den.
- En trigger startar på en händelse och är knuten till en händelse i en tabell. Starten sker vid händelser som är av typen DML eller DDL.

  Det kan vara att om du tar bort en fakturarad ska händelsen (DML DELETE) med raderingen i faktura starta en funktionalitet som uppdaterar samma artikel i artikel-tabellen med rätt antal. Dvs öka på antalet på artikeln med samma antal som raderas i fakturarad.
- Det är alltså möjligt att göra extra bearbetning när ändringar genomförs i en tabell. Kan vara ett bättre sätt än att använda en lagrad procedur.
- En trigger startar även om du utför operationen manuellt direkt i tabellen. Det som sker i en lagrad procedur utförs endast när anrop sker av den lagrade proceduren.

Alltså en trigger är en lagrad procedur som startar automatiskt på en händelse i en tabell.

Trigger – används till

- Lägga till avancerade valideringar och affärsregler.
- Logga händelser / operationer- Både DML och DDL.

  Ex: Man vill alltid kunna se ändringar som genomförts i en tabell. När de har gjorts och av vem och naturligtvis vad som har hänt.
- Meddela ansvarig (DBA DataBase Administrator) via e-post när något viktig har hänt.
- Underhålla härledd information såsom beräknade kolumner, löpande totaler, aggregat etc. I en blogg så kanske man alltid ska se antalet inlägg på en tråd när man ser på inlägget.
- DML-trigger som reagerar på händelser såsom:
  INSERT, UPDATE, DELETE
- DDL- trigger som reagerar på händelser såsom:
  CREATE, ALTER, DROP, GRANT, DENY, REVOKE, or UPDATE STATISTICS
- Det sker AFTER eller INSTEAD OF

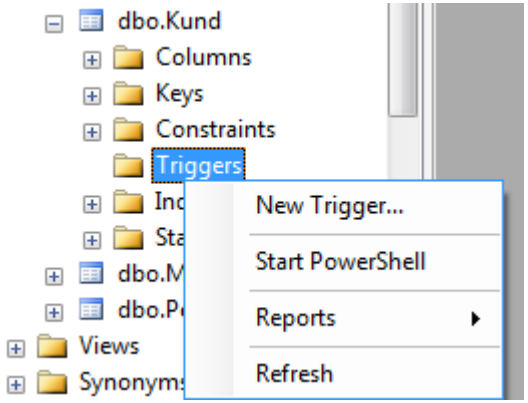
Trigger – syntax

Books Online

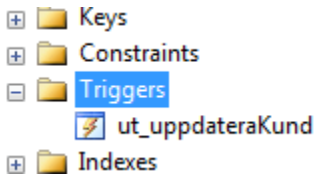
Förenklad modell för DML trigger.

```
CREATE TRIGGER [trigger-name] ON [table]
AFTER INSERT DELETE UPDATE          -- triggar på alla tre
AS
BEGIN
   … rader med kod
   … rader med kod
END
GO
```

Så här skapar du en. Högerklicka på triggers i den tabell där du ska koppla in trigger.

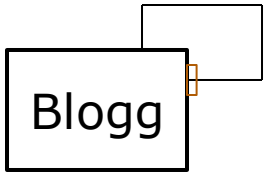


… och här hittar du den



Trigger – Ett exempel

En trigger som kan användas för att tilldela utgångsposten med antal inlägg i en blogg.



	Column Name	Data Type
🔑	Bloggid	int
	Datum	datetime
	Comment	varchar(200)
	Bloggid_U	int
▶	Antal	int

```
CREATE TRIGGER ut_uppdateraBlogg ON Blogg
AFTER INSERT DELETE                    -- startar på insert och delete
AS
BEGIN
   @DECLARE @Bloggid int = 23
   UPDATE Blogg SET antal= (SELECT Count(*) FROM Blogg
                            WHERE Bloggid_U=@Bloggid)
   WHERE Bloggid=@Bloggid;
END
```

När en post läggs till eller raderas i tabellen Blogg kommer fältet Antal i Blogg att uppdateras med antalet poster i Blogg som tillhör tråden.

Trigger - Hur fungerar

När en trigger används så skapas två tabeller, virtuella tabeller, som du kan använda dig av:

Tabellnamn	Innehåller
inserted	Nya eller uppdaterade rader i samband med INSERT och UPDATE Tabellen är tom vid DELETE
deleted	De ursprungliga ändrade eller raderade rader. Dvs innan de ändrades / raderades. Gäller även för INSERT.

- ✓ Tabellerna har samma kolumner som tabellen som triggern ligger i
- ✓ Tabellerna finns endast tillgängliga för triggern
- ✓ Alla påverkade rader finns i de virtuella tabellerna

DML	Inserted	Deleted
Insert	Poster är infogade	Tom
Update	Poster är uppdaterade	Poster innan uppdatering
Delete	Tom	Raderade poster

Trigger – Ett exempel

En trigger som kan användas för att uppdatera antal i Artikel när en fakturarad skapas.

*Detta är ett **exempel** som helst löses med en transaktion i en lagrad procedur. Kan du undvika triggers – gör det då.*

```
CREATE TRIGGER ut_uppdateraAtrikel ON Fakturarad
AFTER INSERT                          -- startar på insert
AS
BEGIN
   UPDATE Artikel SET antal= antal-
      (SELECT Antal
      FROM inserted)
   WHERE Artikelid=(SELECT Artikelid From Inserted);
END
```

Trigger – transaktionsflöde

Att utveckla triggers kräver att man förstår var i händelsekedjan som en trigger exekveras. Följande flöde av MS SQL Server:

1. IDENTY INSERT check
2. Nullability constraint
3. Datatype check
4. INSTEAD OF trigger execution
5. Primary Key constraint
6. Check constraint
7. Foreign Key constraint
8. DML execution and update to the transactions log
9. AFTER trigger execution
10. COMMIT transaction.

Trigger - Tips

- AFTER körs EFTER!!! Ändringar genomförts i databas.
  - ändringar av original DML är redan utförd och kan läsas av andra
  - triggern kan efteråt ändra gjord operation
- INSTEAD OF – Trigger körs i original DML
  - inserted/deleted tabellerna har data som om operationen genomförts
- Intigritets constraints görs innan DML ändringar görs i databasen
  - triggrar körs aldrig om det blir intigritets fel
  - om en kombination av constraints och trigger krävs måste allt implementeras i triggern
- Använd hellre constraints istället för trigger om det går
  Trigger tar mycket mera resurser än constraints
- Avsluta en trigger så fort som möjligt.
- En trigger är alltid en del av den transaktion som startar den.
  En trigger kan därför avslutas med en ROLLBACK Transaction

Trigger - UPDATE

Före ändring

Kundid	Namn	Adress	Postnr	Ort	Orgnr
1	Svensson Mekaniska	Storgatan 23	393 64	KALMAR	999899-1234

```
UPDATE Kund Set namn='SVEMEK'
WHERE KundID=1
```

Tabellen INSERTED = Rader efter ändring

Kundid	Namn	Adress	Postnr	Ort	Orgnr
1	SVEMEK	Storgatan 23	393 64	KALMAR	999899-1234

Tabellen DELETED Rader före ändring

Kundid	Namn	Adress	Postnr	Ort	Orgnr
1	Svensson Mekaniska	Storgatan 23	393 64	KALMAR	999899-1234

```
If UPDATE(Namn)=True                    -- Se också COLUMNS_UPDATE()
   INSERT INTO Log (ID, Tbl, Kol, Old, New, Datum, Usr)
   VALUES (@Kundid, 'Kund','Namn',@Old,@New,getdate,@Usr)
```

Exempel Radera Faktura

Radera en Faktura med sina fakturarader med hjälp av en trigger. På denna sida finns den lagrade proceduren. På nästa sida finns triggern.

```
CREATE PROCEDURE usp_raderaFaktura (@FakturaID int)
AS
BEGIN
  BEGIN TRY
    BEGIN TRAN                          -- Radera fakturarader först
      DELETE FROM Fakturarad
      WHERE FakturaID=@FakturaID;       -- Uppdatering av antal i
                                        -- Artikel-tabellen sköts
                                        -- med trigger (se nästa slide)
      DELETE FROM Faktura               -- radera fakturan sist
      WHERE FakturaID=@FakturaID;
    COMMIT TRAN
  END TRY
  BEGIN CATCH
    ROLLBACK TRAN
    RAISERROR('Borttagningen gick inte att genomföra!', 16, 1)
  END CATCH
END
GO
```

Exempel Trigger

När en fakturarad raderas så startar triggern som uppdaterar Artikel.

```
CREATE TRIGGER ut_uppdateraArtikel ON Fakturarad
AFTER DELETE                                    -- Startar på DELETE
AS
DECLARE @ArtikelID int, @Antal int
DECLARE myCursor INSENSITIVE CURSOR             -- Skapar Cursor
FOR SELECT ArtikelID, Antal
FROM DELETED                                    -- Ur Deleted
FOR READ ONLY;                                  -- Enbart läsa
OPEN myCursor                                   -- Öppnar Cursor
FETCH myCursor INTO @ArtikelID, @Antal          -- Hämta första raden
WHILE @@FETCH_STATUS = 0
   BEGIN
      UPDATE Artikel
      SET Antal = Antal + @Antal
      WHERE ArtikelID = @ArtikelID;
      FETCH myCursor INTO @ArtikelID, @Antal          -- Nästa rad
   END
CLOSE myCursor
DEALLOCATE myCursor;
```

Exempel utan Trigger

Ett (förenklat av utrymmesskäl) exempel där en faktura raderas med uppdatering av antal på artikel.

```
CREATE PROCEDURE usp_raderaFaktura
@fakturaID INT = NULL
AS
BEGIN
   BEGIN TRAN                                  -- startar transaktionen
      UPDATE Artikel                           -- Uppdaterar artikel
      SET Antal = Antal+COALESCE ((SELECT SUM(ANTAL)
        FROM FakturaRad WHERE FakturaID = @fakturaid And
            Fakturarad.ArtikelID = Artikel.ArtikelID ),0)

      DELETE FROM dbo.FakturaRad               -- raderar fakturarader
      WHERE FakturaID = @fakturaID;

      DELETE FROM dbo.Fakturan                 -- raderar fakturan
      WHERE FakturaID = @fakturaID;

   COMMIT TRAN                                 -- avslutar transaktionen
END
```

COALESCE Returnerar icke null värden