Relational Databases and Datawarehousing Database Programming

What do you learn in this chapter ?

- How the relational model has been extended to support advance database applications
- Procedural SQL statements
- Stored procedures and stored functions
- Cursors
- Triggers
- Tables and user defined types

HO GENT

SQL as complete language (PSM)

HO GENT

Persistent Stored Modules

- Originally SQL was not a complete programming language
    - But SQL could be embedded in a standard programming language
- SQL/PSM, as a complete programming language
    - variables, constants, datatypes
    - operators
    - Controle structures
        - if, case, while, for, …
    - procedures, functions
    - exception handling

HO
GENT

Persistent Stored Modules

- PSM = stored procedures and stored functions
- Examples
  - SQL Server: Transact SQL
  - Oracle: PL/SQL
  - DB2: SQL PL
- Procedural SQL extensions are vendor specific, database systems that use these extensions are more difficult to migrate to another RDBMS!

HO GENT

Proprietary languages!

- ## Oracle PL/SQL

```
CREATE PROCEDURE youngestEmployee (employeeName OUT VARCHAR2(20))
AS
BEGIN
      SELECT LastName INTO employeeName FROM employees WHERE birthdate =
      (SELECT MAX(birthdate) FROM employees);
END;
```

- ## MicroSoft
  Transact-SQL

```
CREATE PROCEDURE youngestEmployee @employeeName VARCHAR(20) OUT
AS
SELECT @employeeName = LastName FROM employees
WHERE birthdate = (SELECT MAX(birthdate) FROM employees)
```

- ## MySQL

```
CREATE PROCEDURE youngestEmployee (OUT employeeName VARCHAR(20))
BEGIN
SET employeeName = (SELECT LastName FROM employees
WHERE birthdate = (SELECT MAX(birthdate) FROM employees)
END
```

HO
GENT

Why using PSM's (SP and UDF)?

- PSM vs. 3GL (Java, .NET, C++, Cobol...)
  - Older versions of SQL Server (6.5 & 7) and Oracle:
    - query-optimisation and execution plan caching and reuse is better when using PSM
      → SQL execution through PSM was more performant
    - now: +/- same optimisation, no matter how query arrives at database
    - unfortunately performance is still used as a reason to use PSM's

HO GENT

PSM: advantages

- Code modularisation
  - reduce redundant code: put frequently used queries in SP and reuse in 3GL
    → Less maintenance tasks at schema updates
    → Often used for CRUD-operations
- Customisation of "closed" systems like ERP: via stored procedures and triggers you can influence behaviour
- Security
  - Exclude direct queries on tables
  - SP's determine what is allowed
  - avoid SQL-injection attacks by using input parameters
- Central administration of (parts of) DB code

HO
GENT

PSM: disadvantages

- Reduced scalability: business logic and db processing run on same server, can cause bottlenecks.
- Vendor lock-in
  - syntax = non standard: port from e.g. MS SQL Server to Oracle very time consuming
  - but portability comes with a price (ex. built-in functions can't be used)
- Two programming languages: Java/.Net/… vs SP/UDF/…
- Two debug environments
- SP/UDF: limited OO support

HO GENT

Stored Procedures

HO
GENT

Stored procedure

> **a stored procedure** is a named collection of SQL and control-of-flow commands (program) that is stored as a database object

- What?
  - Analogous to procedures/functions/methods in other languages
  - Can be called from a program, trigger or stored procedure
  - Is saved in the catalogue
  - Accepts input and output parameters
  - Returns status information about the correct or incorrect execution of the stored procedure
  - Contains the tasks to be executed

HO
GENT

Local variables

- A variable name always starts with a @

```
DECLARE @variable_name1 data_type [, @variable_name2 data_type ...]
```

- Assign a value to a variable

```
SET     @variable_name = expression
SELECT  @variable_name = column_specification
```

- Set and select are equivalent, but set is ANSI standard

- With select you can assign a value to several variables at once:

```
DECLARE @maxQuantity SMALLINT, @maxUnitPrice MONEY, @minUnitPrice MONEY
SET @maxQuantity = (SELECT MAX(quantity) FROM OrderDetails)
SELECT @maxQuantity = MAX(quantity) FROM OrderDetails
SELECT @maxUnitPrice = MAX(UnitPrice), @minUnitPrice = MIN(UnitPrice) FROM Products
```

HO
GENT

Local variables

- PRINT: SQL Server Management Studio shows a message in the message tab

```
PRINT string_expression
```

- As an alternative you can also use SELECT

```
DECLARE @maxQuantity SMALLINT
SELECT @maxQuantity = MAX(quantity) FROM OrderDetails

-- Result in tab Messages
PRINT 'The maximum ordered quantity is ' + STR(@maxQuantity)

-- Result in tab Result
SELECT 'The maximum ordered quantity is ' + STR(@maxQuantity)
```

The maximum ordered quantity is   130

	(No column name)
1	The maximum ordered quantity is 130

Operators in Transact SQL

- Arithmetic operators
  - -, +, *, /, %(modulo)
- Comparison operators
  - <, >, =, ... , IS NULL, LIKE, BETWEEN, IN
- Alphanumeric operators
  - + (string concatenation)
- Logic operators
  - AND, OR, NOT

HO GENT

Control flow with Transact SQL

```sql
CREATE PROCEDURE ShowFirstXEmployees @x INT, @missed INT OUTPUT
AS
DECLARE @empid INT, @fullname VARCHAR(100), @city NVARCHAR(30), @total int

SET @empid = 1
SELECT @total = COUNT(*) FROM Employees
-- SET @total = (SELECT COUNT(*) FROM Employees)
SET @missed = 10

IF @x > @total
    SELECT @x = COUNT(*) FROM Employees
ELSE
    SET @missed = @total - @x

WHILE @empid <= @x
BEGIN
    SELECT @fullname = firstname + ' ' + lastname, @city = city FROM Employees WHERE employeeid = @empid
    PRINT 'Full Name : ' + @fullname
    PRINT 'City : '+ @city
    PRINT '------------------------'
    SET @empid = @empid + 1
END
```

Control flow with Transact SQL

```
-- Testcode
DECLARE @numberOfMissedEmployees INT
SET @numberOfMissedEmployees = 0
EXEC ShowFirstXEmployees 5, @numberOfMissedEmployees OUT -- don't forget OUT
PRINT 'Number of missed employees: ' + STR(@numberOfMissedEmployees)
```

```
Full Name : Nancy Davolio
City : Seattle
-----------------------

Full Name : Andrew Fuller
City : Tacoma
-----------------------

Full Name : Janet Leverling
City : Kirkland
-----------------------

Full Name : Margaret Peacock
City : Redmond
-----------------------

Full Name : Steven Buchanan
City : London
-----------------------

Number of missed employees:        4
```

HO
GENT

Comments

- Inline comments

```
-- This is an inline comment
```

- Block comments

```
/*
This is a block comment
*/
```

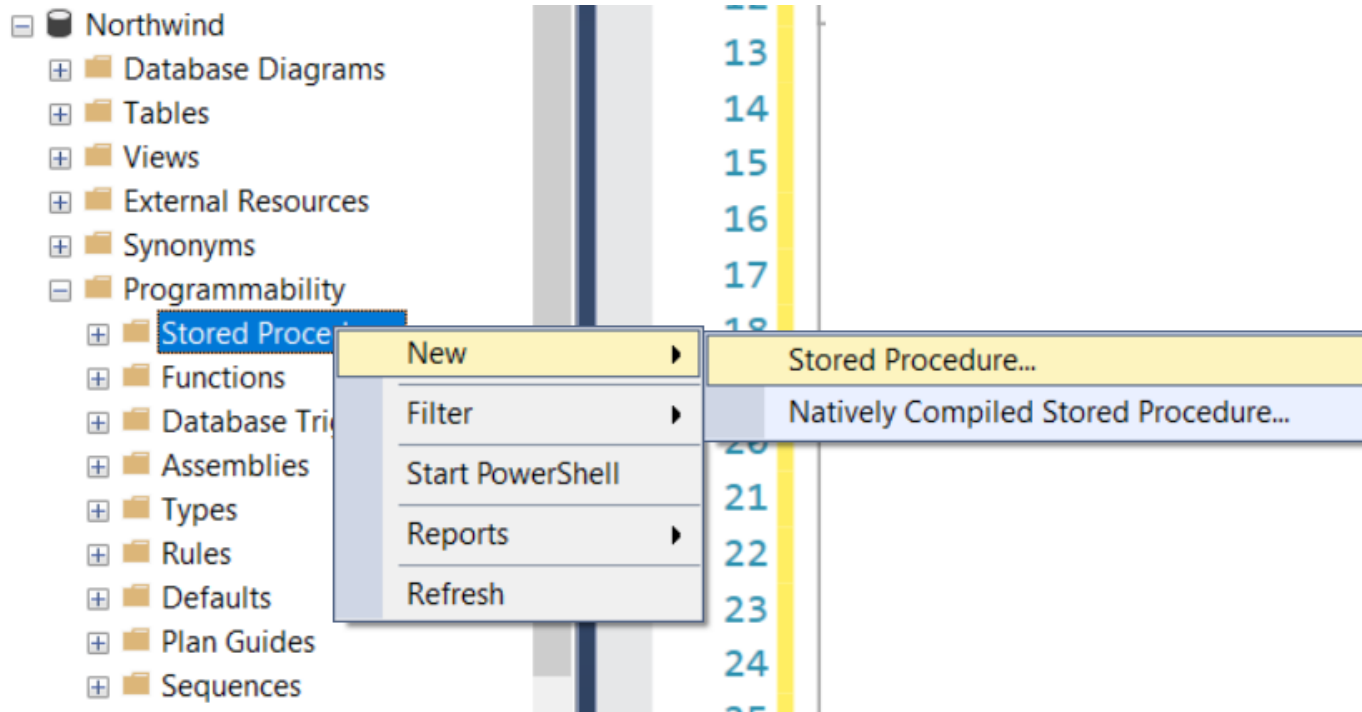HO
GENT

Creation of SP

```
CREATE PROCEDURE <proc_name> [parameter declaratie]
AS
<sql_statements>
```

- Create db object: via DDL instruction

- Syntax control upon creation
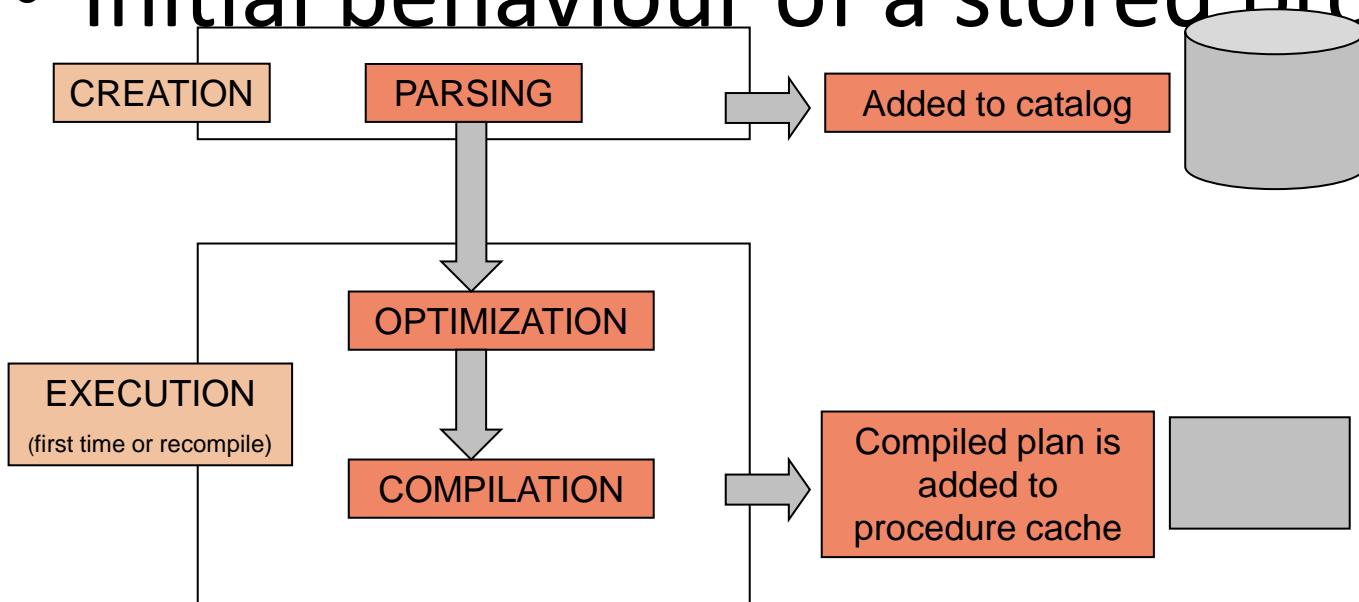 The SP is only stored in de DB if it is syntactically correct

```
CREATE PROCEDURE OrdersSelectAll
AS
SELECT * FROM Orders
```

HO GENT

Creation of SP

Procedural database objects

- Initial behaviour of a stored procedure

CREATION	PARSING	→	Added to catalog

OPTIMIZATION

EXECUTION

(first time or recompile)

COMPILATION → Compiled plan is added to procedure cache

HO
GENT

Changing and removing a SP

```
ALTER PROCEDURE <proc_name> [parameter
declaration]
AS
<sql_statements>
```

```sql
ALTER  PROCEDURE OrdersSelectAll @customerID int
AS
SELECT * FROM orders
WHERE customerID = @customerID
```

```
DROP PROCEDURE <proc_name>
```

```sql
DROP PROCEDURE OrdersSelectAll
```

HO
GENT

Return value of SP

- After execution a SP returns a value
  - Of type int
  - Default return value = 0
- RETURN statement
  - Execution of SP stops
  - Return value can be specified

```
-- Creation of SP with explicit return value
CREATE PROCEDURE OrdersSelectAll AS
SELECT * FROM Orders
RETURN @@ROWCOUNT
```

```
-- Use of SP with return value.
-- Result of print comes in Messages tab.
DECLARE @numberOfOrders INT
EXEC @numberOfOrders = OrdersSelectAll
PRINT 'We have ' + STR(@numberOfOrders) + ' orders.'
```

HO
GENT

SP with parameters

- Types of parameters

  – A parameter is passed to the SP with an input parameter

  – With an output you can possibly pass a value to the SP and get a value back

```
CREATE PROCEDURE OrdersSelectAllForCustomer @customerID int, @numberOfOrders int OUTPUT
AS
SELECT @numberOfOrders = COUNT(*)
FROM orders
WHERE customerID = @customerID
```

```
-- Use of a default value for the input parameter
CREATE PROCEDURE OrdersSelectAllForCustomer @customerID int = 5, @numberOfOrders int OUTPUT
AS
SELECT @numberOfOrders = COUNT(*)
FROM orders
WHERE customerID = @customerID
```

HO
GENT

SP with parameters

- Calling the SP
  - Always provide keyword OUTPUT for output parameters
  - 2 ways to pass actual parameters
    - use formal parameter name (order unimportant)
    - positional

```
-- Pass param by explicit use of formal parameters
DECLARE @nmbrOfOrders int
EXECUTE OrdersSelectAllForCustomer @customerID = 5, @numberOfOrders = @nmbrOfOrders OUTPUT
PRINT @nmbrOfOrders
```

```
-- Positional parameter passing
DECLARE @nmbrOfOrders int
EXECUTE OrdersSelectAllForCustomer 5, @nmbrOfOrders OUTPUT
PRINT @nmbrOfOrders
```

HO
GENT

Error handling in Transact SQL

- RETURN
  - Immediate end of execution of the batch procedure
- @@error
  - Contains error number of last executed SQL instruction
  - Value = 0 if OK
- RAISERROR
  - Returns user defined error or system error

- Use of TRY …. CATCH block

HO
GENT

Error handling → using @@error

```
CREATE PROCEDURE ProductInsert @productName nvarchar(50) = NULL, @categoryID int = NULL AS
DECLARE @errormsg int
INSERT INTO Products(ProductName, CategoryID, Discontinued) VALUES (@productName,@categoryID, 0)

-- save @@error to avoid overwriting by consecutive statements
SET @errormsg = @@error

IF @errormsg = 0
    PRINT 'SUCCESS! The product has been added.'
ELSE IF @errormsg = 515
    PRINT 'ERROR! ProductName is NULL.'
ELSE IF @errormsg = 547
    PRINT 'ERROR! CategoryID doesn''t exist.'
ELSE PRINT 'ERROR! Unable to add new product. Error:' + str(@errormsg)

RETURN @errormsg

-- Testcode
BEGIN TRANSACTION
    EXEC ProductInsert 'Wokkels', 10
    SELECT * FROM Products WHERE productName LIKE '%Wokkels%'
ROLLBACK;
```

HO
GENT

Error handling in Transact SQL

- All system error messages are in the system table sysmessages

```
SELECT * FROM master.dbo.sysmessages WHERE error = @@ERROR
```

- Create your own messages using raiserror
  - RAISERROR(msg, severity, state)
    - msg – the error message
    - severity – value between 0 and 18
    - state – value between 1 and 127, to distinguish between consecutive calls with same message.

HO
GENT

Error handling → using RAISERROR

```sql
CREATE PROCEDURE ProductInsert @productName nvarchar(50) = NULL, @categoryID int = NULL AS
DECLARE @errormsg int
INSERT INTO Products(ProductName, CategoryID, Discontinued) VALUES (@productName,@categoryID, 0)

-- save @@error to avoid overwriting by consecutive statements
SET @errormsg = @@error

IF @errormsg = 0
     PRINT 'SUCCESS! The product has been added.'
ELSE IF @errormsg = 515
     RAISERROR ('ProductName or CategoryID is NULL.',18,1)
ELSE IF @errormsg = 547
     RAISERROR ('CategoryID doesn''t exist.',18,1)
ELSE PRINT 'ERROR! Unable to add new product. Error:' + str(@errormsg)

RETURN @errormsg

-- Testcode
BEGIN TRANSACTION
     EXEC ProductInsert 'Wokkels', 10
     SELECT * FROM Products WHERE productName LIKE '%Wokkels%'
ROLLBACK;
```

Exception handling: catch-block functions

- ERROR_LINE(): line number where exception occurred

- ERROR_MESSAGE(): error message

- ERROR_PROCEDURE(): SP where exception occurred

- ERROR_NUMBER(): error number

- ERROR_SEVERITY(): severity level

HO
GENT

Exception handling

```sql
ALTER PROCEDURE DeleteShipper @ShipperID int = NULL, @NumberOfDeletedShippers int OUT
AS
BEGIN
     BEGIN TRY
          DELETE FROM Shippers WHERE ShipperID = @ShipperID
          SET @NumberOfDeletedShippers = @@ROWCOUNT
     END TRY
     BEGIN CATCH
          PRINT 'Error Number = ' + STR(ERROR_NUMBER());
          PRINT 'Error Procedure = ' + ERROR_PROCEDURE();
          PRINT 'Error Message = ' + ERROR_MESSAGE();
     END CATCH
END

-- Testcode
BEGIN TRANSACTION
DECLARE @nrOfDeletedShippers int;
EXEC DeleteShipper 3, @nrOfDeletedShippers OUT;
PRINT 'Number of deleted shippers ' + STR(@nrOfDeletedShippers);
ROLLBACK
```

HO
GENT

Exceptions: real life example

```sql
CREATE PROCEDURE DeleteShipper @ShipperID int, @NumberOfDeletedShippers int OUT
AS
BEGIN
      BEGIN TRY
            BEGIN TRANSACTION
            DELETE FROM Shippers WHERE ShipperID = @ShipperID
            SET @NumberOfDeletedShippers = @@ROWCOUNT
            COMMIT
      END TRY
      BEGIN CATCH
            ROLLBACK
            INSERT INTO log values(GETDATE(), ERROR_MESSAGE(), ERROR_NUMBER(), ERROR_PROCEDURE(),
            ERROR_LINE(), ERROR_SEVERITY())
      END CATCH
END
```

HO
GENT

Throw

- Is an alternative to RAISERROR
- Raises an exception and transfers execution to a CATCH block of a TRY...CATCH construct in SQL Server.
- Without parameters: only in catch block (= rethrowing the caught exception)
- With parameters: also outside catch block
- Create your own user defined exceptions
  - THROW(error_number, message, state)
    - error_number: represents the exception. Is an int between 50000 and 2147483647.
    - state: value between 1 and 127, to distinguish between consecutive calls with same message.

HO
GENT

Throw: without parameters

```
-- Throw: Example 1 - Without parmaters => rethrowing the exception

ALTER PROCEDURE DeleteShipper @ShipperID int = NULL, @NumberOfDeletedShippers int OUT
AS
BEGIN
    BEGIN TRY
        DELETE FROM Shippers WHERE ShipperID = @ShipperID
        SET @NumberOfDeletedShippers = @@ROWCOUNT
    END TRY
    BEGIN CATCH
        PRINT 'This is an error';
        THROW-- if you put this in comment => the exception is caught and isn't shown in Messages
    END CATCH
END

-- Testcode
BEGIN TRANSACTION
DECLARE @nrOfDeletedShippers int;
EXEC DeleteShipper 3, @nrOfDeletedShippers OUT;
PRINT 'Number of deleted shippers ' + STR(@nrOfDeletedShippers);
ROLLBACK
```

HO
GENT

Throw: with parameters

```sql
-- Throw: Example 2 - With parameters => create your own user defined exception

ALTER PROCEDURE DeleteShipper @ShipperID int = NULL, @NumberOfDeletedShippers int OUT
AS
BEGIN
    BEGIN TRY
        DELETE FROM Shippers WHERE ShipperID = @ShipperID
        SET @NumberOfDeletedShippers = @@ROWCOUNT
    END TRY
    BEGIN CATCH
        PRINT 'This is an error';
        THROW 50001, 'The shipper isn''t deleted', 1
    END CATCH
END

-- Testcode
BEGIN TRANSACTION
DECLARE @nrOfDeletedShippers int;
EXEC DeleteShipper 3, @nrOfDeletedShippers OUT;
PRINT 'Number of deleted shippers ' + STR(@nrOfDeletedShippers);
ROLLBACK
```

HO
GENT

SP example: DeleteShipper revisited

```
ALTER PROCEDURE DeleteShipper @ShipperID int = NULL, @NumberOfDeletedShippers int OUT AS
BEGIN
      IF @ShipperID IS NULL
      BEGIN
            PRINT 'Please provide a shipperID'
            RETURN
      END

      IF NOT EXISTS (SELECT * FROM Shippers where ShipperID = @ShipperID)
      BEGIN
            PRINT 'The shipper doesn''t exist.'
            RETURN
      END

      IF EXISTS (SELECT * FROM Orders where ShipVia = @ShipperID)
      BEGIN
            PRINT 'The shipper already has orders and can''t be deleted.'
            RETURN
      END

      DELETE FROM Shippers WHERE ShipperID = @ShipperID
      PRINT 'The shipper has succesfully deleted'
END
```

```
-- Testcode
BEGIN TRANSACTION
DECLARE @nrOfDeletedShippers int;
EXEC DeleteShipper NULL, @nrOfDeletedShippers OUT;
PRINT 'Number of deleted shippers ' +
STR(@nrOfDeletedShippers);
ROLLBACK
```

SP example: Insert Shipper with identity

```sql
CREATE PROCEDURE InsertShipper @CompanyName NVARCHAR(40), @phone NVARCHAR(24) = NULL, @shipperID INT OUT
AS
INSERT INTO Shippers(CompanyName, Phone)
VALUES (@CompanyName, @Phone)

SET @shipperID = @@identity

-- Testcode
BEGIN TRANSACTION
DECLARE @NewShipperID INT;
EXEC InsertShipper 'Solid Shippings', '(503) 555-9874', @NewShipperID OUT;
PRINT 'ID of inserted shipper: ' + STR(@NewShipperID);
ROLLBACK
```

HO
GENT

Functions

- Standard SQL functions: MIN, MAX, AVG, SUM, COUNT
- Non-standard built-in functions:
  SQL Server: DATEDIFF, SUBSTRING, LEN, ROUND, …
  http://technet.microsoft.com/en-us/library/ms174318.aspx
- User defined functions

**HO
GENT**

Why user defined functions?

- Give the age of each employee:

```
SELECT lastname, firstname, birthdate, GETDATE() As today, DATEDIFF(YEAR,birthdate,GETDATE()) age
FROM employees
```

	lastname	firstname	birthdate	today	age
1	Davolio	Nancy	1978-12-08 00:00:00.000	2021-03-01	43
2	Fuller	Andrew	1982-02-19 00:00:00.000	2021-03-01	39
3	Leverling	Janet	1993-08-30 00:00:00.000	2021-03-01	28
4	Peacock	Margaret	1967-09-19 00:00:00.000	2021-03-01	54
5	Buchanan	Steven	1975-03-04 00:00:00.000	2021-03-01	46
6	Suyama	Michael	1983-07-02 00:00:00.000	2021-03-01	38

Will probably not be happy

HO
GENT

Why user defined functions?

- Give the age of each employee:

```
SELECT lastname, firstname, birthdate, GETDATE() As today, DATEDIFF(DAY,birthdate,GETDATE()) / 365 age
FROM employees
```

	lastname	firstname	birthdate	today	age
1	Davolio	Nancy	1978-12-08 00:00:00.000	2021-03-01	42
2	Fuller	Andrew	1982-02-19 00:00:00.000	2021-03-01	39
3	Leverling	Janet	1993-08-30 00:00:00.000	2021-03-01	27
4	Peacock	Margaret	1967-09-19 00:00:00.000	2021-03-01	53
5	Buchanan	Steven	1975-03-04 00:00:00.000	2021-03-01	46
6	Suyama	Michael	1983-07-02 00:00:00.000	2021-03-01	37

Better but doesn't take leap years into account

HO
GENT

Solution: User Defined Function

```sql
CREATE FUNCTION GetAge (@birthdate AS DATE, @eventdate AS DATE)
RETURNS INT
AS
BEGIN
 RETURN
 DATEDIFF(year, @birthdate, @eventdate)
 - CASE WHEN 100 * MONTH(@eventdate) + DAY(@eventdate)
          < 100 * MONTH(@birthdate) + DAY(@birthdate)
 THEN 1 ELSE 0
 END;
END;
```

```sql
-- How to use?
SELECT lastname, firstname, birthdate, GETDATE() As today,  dbo.GetAge(birthdate,GETDATE()) age
FROM employees
```

	lastname	firstname	birthdate	today	age
1	Davolio	Nancy	1978-12-08 00:00:00.000	2021-10-18 23:24:04.607	42
2	Fuller	Andrew	1982-02-19 00:00:00.000	2021-10-18 23:24:04.607	39
3	Leverling	Janet	1993-08-30 00:00:00.000	2021-10-18 23:24:04.607	27
4	Peacock	Margaret	1967-09-19 00:00:00.000	2021-10-18 23:24:04.607	53

HO GENT

Inline Table Valued User Defined Functions

- Give per product category the price of the cheapest product that costs more than x € and a product with that price.

**HO
GENT**

Inline Table Valued User Defined Functions

- Give per category the price of the cheapest product that costs more than x € and all products with that price.

```sql
CREATE FUNCTION minimum (@limit INT) RETURNS TABLE AS
RETURN
SELECT CategoryID cat, MIN(UnitPrice) minprice
FROM Products WHERE UnitPrice > @limit
GROUP BY CategoryID;
-- USE
SELECT p.CategoryID,p.ProductID,p.UnitPrice,m.minprice
FROM Products p join minimum(5) m ON p.CategoryID = m.cat
WHERE p.UnitPrice=m.minprice
```

- Besides views, subqueries and CTE's, this is another way to reuse SELECT statements, **now even parameterised**!

- This is called an **inline table valued function** because it returns a table in a single SELECT statement in the return.

HO
GENT

UDF: Exercise

```
-- Exercise: Write a function that calculates the netto salary per month for each employee
-- If salary < 4000 EUR per month => tax is 30%
-- If salary < 5500 EUR per month => tax is 35%
-- Else => tax is 40%


-- Give an overview of firstname, lastname, birthdate, salary and netto salary for each employee

-- Give an overview of all employees that earn more than 2800 each month
```

HO
GENT

Exercises

```
-- Exercise 1
-- In case a supplier goes bankrupt, we need to inform the customers on this
-- Write a SP that gives all information about the customers that ordered a product of this supplier during
the last 6 months,
-- given the companyname of the supplier, so we will be able to inform the appropriate customers
-- First check if the companyname IS NOT NULL and if there is a supplier with this companyname
-- Then check if there isn't by chance more than 1 supplier with this companyname
-- Use in the procedure '2018-10-21' instead of GETDATE(), otherwise the procedure won't return any records.
-- The procedure returns the number of found customers using an OUTPUT parameter

TO DO
-- (1) Test if companyname is NOT NULL
-- (2) Test if companyname exists
-- (3) Test if there is more than 1 supplier with the given companyname
-- (4) SELECT statement to get customers from supplier → use DATEDIFF(MONTH, orderDate, '2018-10-21')
-- (5) number of customers --> @@rowcount

-- Write testcode
-- (*) in which companyName IS NULL
-- (*) in which companyName does not exist
-- (*) in which companyName = Refrescos Americanas LTDA.
```

HO GENT

Exercises

```
-- Exercise 2
-- We'd like to have 1 stored procedure to insert new OrderDetails, however make sure that:
-- (*) the OrderID and ProductID exist;
-- (*) the UnitPrice is optional, use it when it's given else retrieve the product's price from the product
table.
-- If the difference between the given UnitPrice and the UnitPrice in the table Products is larger than 15%:
write out a message and don't insert the new OrderDetail
-- (*) If the discount is smaller than 0.0 or larger than 0.25: write out a message and don't insert the new
OrderDetail.
-- The discount is optional. If there is no value for discount => the discount is 0.0
-- (*) If the quantity is larger than 2 * the max ordered quantity of this product untill now: write out a
message and don't insert the new OrderDetail

-- Write testcode in which you check all the special cases
-- Put everything in a transaction, so the original data in the tables isn't changed. You can see messages in
the Messages tab
-- An example that should work: OrderID = 10249 + ProductID = 72 + UnitPrice = 35.00 + qauntity = 10 +
discount = 0.15
-- OrderID = 10249 contains already 2 rows:
-- OrderID    ProductID    UnitPrice    Quantity    Discount
-- 10249      14           18,60        9           0
-- 10249      51           42,40        40          0
```

Exercises

```
-- Exercise 3
-- Create a stored procedure for deleting a shipper. You can only delete a shipper if
-- (*) The shipperID exists
-- (*) There are no Orders for this shipper

-- Write two versions of your procedure:
-- (*) In the first version you check these conditions before deleting the shipper,
-- so you don't rely on SQL Server messages. Generate an appropriate error message if the shipper can't be
deleted.
-- (*) In the second version you try to delete the shipper and catch the exceptions that might occur.

-- Write testcode to delete shipper with shipperID = 10 (doesn't exist) / 5 (exists + no shippings) / 3
(exists + already shippings).
-- Put everything in a transaction. Messages are visible on the Messages tab
```

HO
GENT

Cursors

HO
GENT

Cursors

- SQL statements process complete result sets and not individual rows.
  Cursors allow to process individual rows to perform complex row specific operations that can't (easily) be performed with a single SELECT, UPDATE or DELETE statement.

- A cursor is a database object that refers to the result of a query. It allows to specify the row from the result set you wish to process.
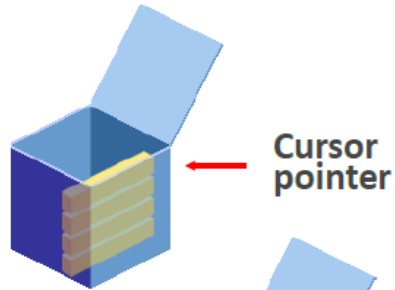
HO GENT

Cursors

- 5 important cursor related statements
  - DECLARE CURSOR – creates and defines the cursor
  - OPEN – opens the declared cursor
  - FETCH – fetches 1 row
  - CLOSE – closes the cursor (counterpart of OPEN)
  - DEALLOCATE – remove the cursor definition (counterpart of DECLARE)
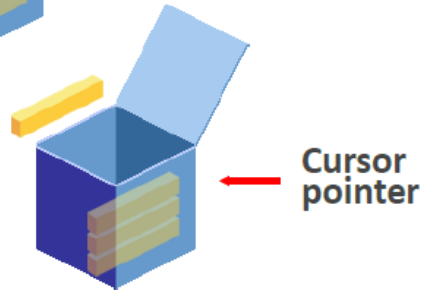
HO GENT
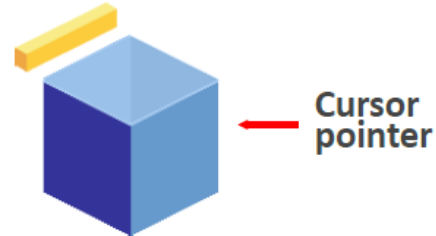
Cursors

Cursor declaration

```
DECLARE <cursor_name> [INSENSITIVE][SCROLL] CURSOR FOR
<SELECT_statement>
[FOR {READ ONLY | UPDATE[OF <column list>]}]
```

- INSENSITIVE
  - The cursor uses a temporary copy of the data
    - Changes in underlying tables after opening the cursor are not reflected in data fetched by the cursor
    - The cursor can't be used to change data (read-only, see below)
  - If INSENSITIVE is omitted, deletes and updates are reflected in the cursor
    → less performant because each row fetch executes a SELECT

Cursor declaration

```
DECLARE <cursor_name> [INSENSITIVE][SCROLL] CURSOR FOR
<SELECT_statement>
[FOR {READ ONLY | UPDATE[OF <column list>]}]
```

- SCROLL
  - All fetch operations are allowed
    - FIRST, LAST, PRIOR, NEXT, RELATIVE and ABSOLUTE
    - Might result in difficult to understand code
  - If SCROLL is omitted only NEXT can be used
- READ ONLY
  - Prohibits data changes in underlying tables through cursor
- UPDATE
  - Data changes are allowed
  - Specify columns that can be changed via the cursor

Opening a cursor

```
OPEN <cursor name>
```

- The cursor is opened
- The cursor is "filled"
  - The select statement is executed. A "virtual table" is filled with the "active set".
- The cursor's current row pointer is positioned just before the first row in the result set.

Fetching data with a cursor

```
FETCH [NEXT | PRIOR | FIRST | LAST | {ABSOLUTE | RELATIVE
<row number>}]
FROM <cursor name>
[INTO <variable name>[,...<last variable name>]]
```

- The cursor is positioned
  - On the next (or previous, first, last, …) row
  - Default only NEXT is allowed
    - For other ways use a SCROLL-able cursor
- Data is fetched
  - without INTO clause resulting data is shown on screen
  - with INTO clause data is assigned to the specified variables
    - Declare a corresponding variable for each column in the cursor SELECT

Closing a cursor

```
CLOSE <cursor name>
```

- The cursor is closed
  - The cursor definition remains
  - Cursor can be reopened

Deallocating a cursor

```
DEALLOCATE <cursor name>
```

- The cursor definition is removed
  - If this was the last reference to the cursor all cursor resources
    (the "virtual table") are released.
  - If the cursor has not been closed yet
    DEALLOCATE will close the cursor

```sql
DECLARE @supplierID INT, @companyName NVARCHAR(30), @city NVARCHAR(15)

-- declare cursor
DECLARE suppliers_cursor CURSOR
FOR
SELECT SupplierID, CompanyName, City
FROM Suppliers
WHERE Country = 'USA'

-- open cursor
OPEN suppliers_cursor

-- fetch data
FETCH NEXT FROM suppliers_cursor INTO @supplierID, @companyName, @city

WHILE @@FETCH_STATUS = 0
BEGIN
PRINT 'Supplier: ' + str(@SupplierID) + ' ' + @companyName + ' ' + @city
        FETCH NEXT FROM suppliers_cursor INTO @supplierID, @companyName, @city
END

-- close cursor
CLOSE suppliers_cursor

-- deallocate cursor
DEALLOCATE suppliers_cursor
```

```
Supplier:          2 New Orleans Cajun Delights New Orleans
Supplier:          3 Grandma Kelly's Homestead Ann Arbor
Supplier:         16 Bigfoot Breweries Bend
Supplier:         19 New England Seafood Cannery Boston
```

Nested cursors

- In real-life programs you often need to declare and use two or more cursors in the same block.

- Often these cursors are related to each other by parameters.

- One common example is the need for multi-level reports in which each level of the report uses rows from a different cursor.

```sql
DECLARE @supplierID INT, @companyName NVARCHAR(30), @city NVARCHAR(15)
DECLARE @productID INT, @productName NVARCHAR(40), @unitPrice MONEY

DECLARE suppliers_cursor CURSOR FOR
SELECT SupplierID, CompanyName, City FROM Suppliers WHERE Country = 'USA'

OPEN suppliers_cursor -- open cursor

FETCH NEXT FROM suppliers_cursor INTO @supplierID, @companyName, @city  -- fetch data

WHILE @@FETCH_STATUS = 0
BEGIN
        PRINT 'Supplier: ' + str(@SupplierID) + ' ' + @companyName + ' ' + @city

        --  begin inner cursor
        DECLARE products_cursor CURSOR FOR
        SELECT ProductID, ProductName, UnitPrice FROM Products WHERE SupplierID = @supplierID

        OPEN products_cursor  -- open cursor

        FETCH NEXT FROM products_cursor INTO @productID, @productName, @unitPrice          -- fetch data

        WHILE @@FETCH_STATUS = 0
        BEGIN
                PRINT '- ' + STR(@productID) + ' ' + @productName + ' ' + STR(@unitPrice) + 'EUR'
                FETCH NEXT FROM products_cursor INTO @productID, @productName, @unitPrice
        END

        CLOSE products_cursor
        DEALLOCATE products_cursor
        -- end inner cursor

  FETCH NEXT FROM suppliers_cursor INTO @supplierID, @companyName, @city
END

CLOSE suppliers_cursor
DEALLOCATE suppliers_cursor
```

```
Supplier:       2 New Orleans Cajun Delights New Orleans
-            4 Chef Anton's Cajun Seasoning       22EUR
-            5 Chef Anton's Gumbo Mix      21EUR
-            65 Louisiana Fiery Hot Pepper Sauce       21EUR
-            66 Louisiana Hot Spiced Okra        17EUR
Supplier:       3 Grandma Kelly's Homestead Ann Arbor
-            6 Grandma's Boysenberry Spread       25EUR
-            7 Uncle Bob's Organic Dried Pears        30EUR
-            8 Northwoods Cranberry Sauce        40EUR
```

Update and delete via cursors

```
DELETE FROM <table name>
WHERE CURRENT OF <cursor name>
```

```
UPDATE <table name>
SET ...
WHERE CURRENT OF <cursor name>
```

- Positioned update/delete
  - Deletes/updates the row the cursor referred in WHERE CURRENT OF points to
  - = cursor based positioned update/delete

Update and delete via cursors

```sql
-- declare cursor
DECLARE shippers_cursor CURSOR FOR
SELECT ShipperID, CompanyName FROM Shippers

-- open cursor
OPEN shippers_cursor

-- fetch data
FETCH NEXT FROM shippers_cursor INTO @shipperID, @companyName

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT '- ' + STR(@shipperID) + ' ' + @companyName
    IF @shipperID > 4
        DELETE FROM Shippers WHERE CURRENT OF shippers_cursor

    FETCH NEXT FROM shippers_cursor INTO @shipperID, @companyName
END

CLOSE shippers_cursor
DEALLOCATE shippers_cursor
```

HO
GENT

Exercises

```
-- Exercise 1
-- Create the following overview of the number of products per category
/**
Category:        1 Beverages -->          13
Category:        2 Condiments -->           13
Category:        3 Confections -->           13
Category:        4 Dairy Products -->           10
Category:        5 Grains/Cereals -->            7
Category:        6 Meat/Poultry -->          6
Category:        7 Produce -->          5
Category:        8 Seafood -->          12
**/
```

Exercises

```
-- Exercise 2
-- Give an overview of the employees per country. Use a nested cursor.
/*

* UK
        -           5 Steven Buchanan London
        -           6 Michael Suyama London
        -           7 Robert King London
        -           9 Anne Dodsworth London
* USA
        -           1 Nancy Davolio Seattle
        -           2 Andrew Fuller Tacoma
        -           3 Janet Leverling Kirkland
        -           4 Margaret Peacock Redmond
        -           8 Laura Callahan Seattle
*/
```

Exercises

```
-- Exercise 3
/*
Create an overview of bosses and employees who have
to report to this boss and
also give the number of employees who have to report to this boss.
Use a nested cursor.

* Andrew Fuller
    -           1 Nancy Davolio
    -           3 Janet Leverling
    -           4 Margaret Peacock
    -           5 Steven Buchanan
    -           8 Laura Callahan
Total number of employees =            5
* Steven Buchanan
    -           6 Michael Suyama
    -           7 Robert King
    -           9 Anne Dodsworth
Total number of employees =            3
*/
```

Triggers

HO
GENT

Triggers

A **trigger:** a database program, consisting of procedural and declarative instructons, saved in the catalogue and activated by the DBMS if a certain operation on the database is executed and if a certain condition is satisfied.

- Comparable to SP but can't be called explicitly
  - A trigger is automatically called by the DBMS with some DML, DDL, LOGON-LOGOFF commands
    - DML trigger: with an insert, update or delete for a table or view
      (in this course we further elaborate this type of cursors)
    - DDL trigger: executed with a create, alter or drop of a table
    - LOGON-LOGOFF trigger: executed when a user logs in or out
      (MS SQL Server: only LOGON triggers, Oracle: both)

Triggers

- DML triggers
  - Can be executed with
    - insert
    - update
    - delete
  - Are activated
    - before – before the IUD is processed (not supported by SQL Server)
    - instead of – instead of IUD command
    - after – after the IUD is processed (but before COMMIT), this is the default
  - In some DMBS (e.g. Oracle, Db2) you can also specify how many times the cursor is activated
    - for each row
    - for each statement

Procedural database objects

- Procedural programs

Types	Saved as	Execution	Supports parameters
script	separate file	client tool (ex. Management Studio)	no
stored procedure	database object	via application or SQL script	yes
user defined function	database object	via applicaton or SQL script	yes
trigger	database object	via DML statement	no

Why using triggers?

- Validation of data and complex constraints
    - An employee can't be assigned to > 10 projects
    - An employee can only be assigned to a project that is assigned to his department
- Automatic generation of values
    - If an employee is assigned to a project the default value for the monthly bonus is set according to the project priority and his job category
- Support for alerts
    - Send automatic e-mail if an employee is removed from a project

Why using triggers?

- Auditing
  - Keep track of who did what on a certain table
- Replication and controlled update of redundant data
  - If an ordersdetail record changes, update the orderamount in the orders table
  - Automatic update of datawarehouse tables for reporting (see "Datawarehousing")

Advantages

- Major advantage
  - Store functionality in the DB and execute consistently with each database change.
- Consequences
  - no redundant code
    - functionality is centralized in DBMS, not scattered over different applications (desktop, web, mobile), written by different authors
  - written and tested 'once' by an experienced DBA
  - security
    - triggers are in the DB so all security rules apply
  - more processing power
    - for DBMS and DB
  - fits into client-server model
    - 1 call to db-server: al lot can be executed without further communication

Disadvantages

- Complexity
  - DB design & implementation are more complex by shifting functionality from application to DB
  - Very difficult to debug
- Hidden functionality
  - The user can be confronted with unexpected side effects from the trigger, possibly unwanted
  - Triggers can cascade, which is not always easy to predict when designing the trigger
- Performance
  - At each database change the triggers have to be reevaluated
- Portability
  - Restricted to the chosen database dialect (ex. Transact-SQL from MS)

Comparison of trigger functionality

	Oracle	MS SQL Server	MySQL
BEFORE *For validation*	X	simulate via AFTER-trigger+ROLLBACK	X
AFTER	X	X	X
INSTEAD OF *for views*	X	X	N/A
FOR EACH STATEMENT	X	default	default
FOR EACH ROW	X Acces to values before/after per row via :NEW/:OLD vars	N/A Acces to values before/after per row via deleted/inserted pseudo-tables and cursors	X Acces to values before/after per row via NEW/OLD vars
TRANSACTIONS	COMMIT/ROLLBACK Not allowed	COMMIT/ROLLBACK Allowed	COMMIT/ROLLBACK Not allowed

"Virtual" tables with triggers

- 2 temporary tables
  - deleted table
    contains copies of updated and deleted rows
    - During update or delete, rows are moved from the triggering table to the deleted table
    - Those two table have no rows in common

  - inserted table
    contains copies of updated or inserted rows
    - During update or insert, each affected row is copied from the triggering table to the inserted table
    - All rows from the inserted table appear in the triggering table as well.

"Virtual" tables with triggers

- deleted and inserted table

DML Operation	INSERTED Table	DELETED Table
INSERT	Will have the inserted rows	Empty
DELETE	Empty	Will have the deleted rows
UPDATE	Will have the rows after update	Will have the rows before update

"Virtual" tables with triggers

"Virtual" tables with triggers

"Virtual" tables with triggers

Creation of an 'after' trigger

```
CREATE TRIGGER triggerName
ON table
FOR [INSERT, UPDATE, DELETE]
AS ...
```

Simplified syntax for after trigger

- Only by SysAdmin or dbo
- Linked to one table; not to a view
- Is executed
  - after execution of the triggering action (insert, update, delete)
  - after copy of the changes to the temporary tables 'inserted' and 'deleted'
  - before COMMIT

'Delete after' – trigger

- Trigger is a delete instruction
  - *deleted* is a logical table with columns equal to columns of triggering table, containing a copy of the deleted rows

```
-- If a record in OrderDetails is removed => UnitsInStock in Products should be updated
-- 1st try
CREATE OR ALTER TRIGGER deleteOrderDetails ON OrderDetails FOR delete
AS
DECLARE @deletedProductID INT = (SELECT ProductID From deleted)
DECLARE @deletedQuantity SmallINT = (SELECT Quantity From deleted)
UPDATE Products
SET UnitsInStock = UnitsInStock + @deletedQuantity
FROM Products WHERE ProductID = @deletedProductID
```

```
-- Testcode
BEGIN TRANSACTION
SELECT * FROM Products WHERE ProductID = 14 OR ProductID = 51
DELETE FROM OrderDetails WHERE OrderID = 10249
SELECT * FROM Products WHERE ProductID = 14 OR ProductID = 51
ROLLBACK
```

Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the su

```sql
-- If a record in OrderDetails is removed => UnitsInStock in Products should be updated
-- In the previous solution: more than 1 record was found in deleted
-- => error. Use a cursor instead to loop through all the records in deleted
-- 2nd try
CREATE OR ALTER TRIGGER deleteOrderDetails ON OrderDetails FOR delete
AS
DECLARE deleted_cursor CURSOR
FOR
SELECT ProductID, Quantity
FROM deleted

DECLARE @ProductID INT, @Quantity SmallINT

-- open cursor
OPEN deleted_cursor

-- fetch data
FETCH NEXT FROM deleted_cursor INTO @ProductID, @Quantity

WHILE @@FETCH_STATUS = 0
BEGIN
    UPDATE Products
    SET UnitsInStock = UnitsInStock + @Quantity
    FROM Products WHERE ProductID = @ProductID
    FETCH NEXT FROM deleted_cursor INTO @ProductID, @Quantity
END

-- close cursor
CLOSE deleted_cursor

-- deallocate cursor
DEALLOCATE deleted_cursor
```

Insert after – trigger

- Triggering instruction is an insert statement
  - inserted – logical table with columns equal to columns of triggering table, containing a copy of inserted rows

```sql
-- If a new record is inserted in OrderDetails => check if the unitPrice is not too low or too high
CREATE OR ALTER TRIGGER insertOrderDetails ON OrderDetails FOR insert
AS
DECLARE @insertedProductID INT = (SELECT ProductID From inserted)
DECLARE @insertedUnitPrice Money = (SELECT UnitPrice From inserted)
DECLARE @unitPriceFromProducts Money = (SELECT UnitPrice FROM Products WHERE ProductID = @insertedProductID)
IF @insertedUnitPrice NOT BETWEEN @unitPriceFromProducts * 0.85 AND @unitPriceFromProducts * 1.15
BEGIN
        ROLLBACK TRANSACTION
        RAISERROR ('The inserted unit price can''t be correct', 14,1)
END
```

```sql
-- Testcode
BEGIN TRANSACTION
INSERT INTO OrderDetails
VALUES (10249, 72, 60.00, 10, 0.15)
SELECT * FROM OrderDetails WHERE OrderID = 10249
ROLLBACK
```

Insert after – trigger

- Triggering instruction is an insert statement
  - Remark: when triggering by INSERT-SELECT statement more than one record can be added at once. The trigger code is executed only once, but will insert a new record for each inserted record

Update after – trigger

- Triggering instruction is an update statement

```sql
-- If a record is updated in OrderDetails => check if the new unitPrice is not too low or too high
-- If so, rollback the transaction and raise an error
CREATE OR ALTER TRIGGER updateOrderDetails ON OrderDetails FOR update
AS
DECLARE @updatedProductID INT = (SELECT ProductID From inserted)
DECLARE @updatedUnitPrice Money = (SELECT UnitPrice From inserted)
DECLARE @unitPriceFromProducts Money = (SELECT UnitPrice FROM Products WHERE ProductID = @updatedProductID)
IF @updatedUnitPrice NOT BETWEEN @unitPriceFromProducts * 0.85 AND @unitPriceFromProducts * 1.15
BEGIN
	ROLLBACK TRANSACTION
	RAISERROR ('The updated unit price can''t be correct', 14,1)
END
```

```sql
-- Testcode
BEGIN TRANSACTION
UPDATE OrderDetails SET UnitPrice = 60 WHERE OrderID = 10249 AND ProductID = 14
SELECT * FROM OrderDetails WHERE OrderID = 10249
ROLLBACK
```

The IF update clase

- Conditional execution of triggers:
  execute only if a specific column is mentioned in update or insert

```sql
CREATE OR ALTER TRIGGER updateOrderDetails ON OrderDetails FOR update
AS
-- If a record is updated in OrderDetails => check if the new unitPrice is not too low or too high
-- If so, rollback the transaction and raise an error
IF update(unitPrice)
BEGIN
        DECLARE @updatedProductID INT = (SELECT ProductID From inserted)
        DECLARE @updatedUnitPrice Money = (SELECT UnitPrice From inserted)
        DECLARE @unitPriceFromProducts Money = (SELECT UnitPrice FROM Products WHERE ProductID = @updatedProductID)
        IF @updatedUnitPrice NOT BETWEEN @unitPriceFromProducts * 0.85 AND @unitPriceFromProducts * 1.15
        BEGIN
                ROLLBACK TRANSACTION
                RAISERROR ('The updated unit price can''t be correct', 14,1)
        END
        END

-- If a record is updated in OrderDetails => check if the new discount is not too low or too high
-- If so, rollback the transaction and raise an error
IF update(Discount)
BEGIN
        DECLARE @updatedDiscount REAL = (SELECT Discount FROM inserted)
        IF @updatedDiscount NOT BETWEEN 0 AND 0.25
        BEGIN
                ROLLBACK TRANSACTION
                RAISERROR ('The updated discount can''t be correct', 14,1)
        END
END
```

Triggers and transactions

- A trigger is part of the same transaction as the triggering instruction
- Inside the trigger this transaction can be ROLLBACKed
- Although a trigger in SQL Server occurs after the triggering instruction, that instruction can still be undone in the trigger

1 trigger for insert and/or update and/or delete

- 1 Trigger can be used for update and/or insert and/or delete
- If necessary, you can distinguish between insert, update and delete

```
-- To check if this is an insert operation
IF NOT EXISTS (SELECT * FROM deleted)
BEGIN

END

-- To check if this is a delete operation
IF NOT EXISTS (SELECT * FROM inserted)
BEGIN

END
```

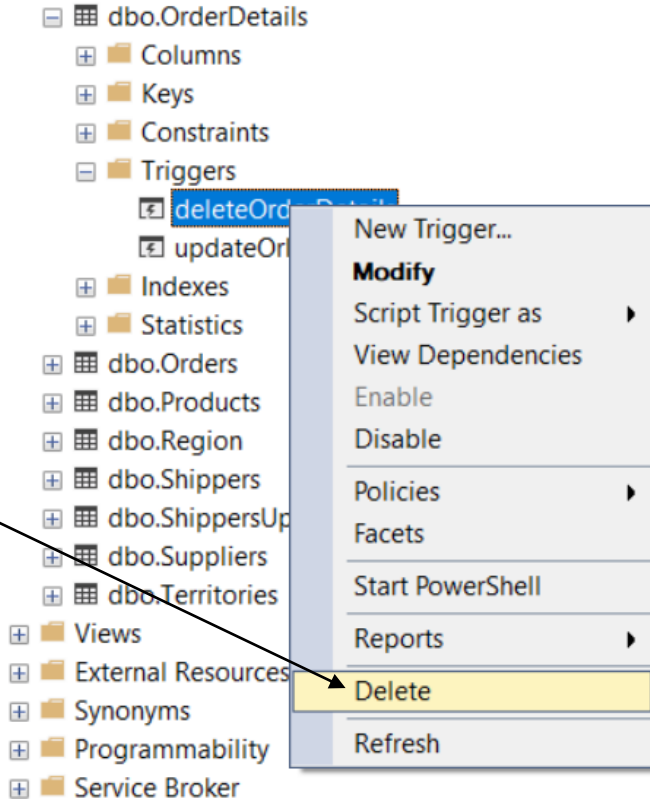# 1 trigger for insert and/or update and/or delete

```
-- If a record is inserted or updated in OrderDetails => check if the new unitPrice is not too low or too high
-- If so, rollback the transaction and raise an error
CREATE OR ALTER TRIGGER updateOrInsertOrderDetails ON OrderDetails FOR update, insert
AS
DECLARE @updatedProductID INT = (SELECT ProductID From inserted)
DECLARE @updatedUnitPrice Money = (SELECT UnitPrice From inserted)
DECLARE @unitPriceFromProducts Money = (SELECT UnitPrice FROM Products WHERE ProductID = @updatedProductID)
IF @updatedProductID NOT BETWEEN @unitPriceFromProducts * 0.85 AND @unitPriceFromProducts * 1.15
BEGIN
        ROLLBACK TRANSACTION
        RAISERROR ('The unit price can''t be correct', 14,1)
END
```

DROP a trigger

- Method 1

```
DROP TRIGGER deleteOrderDetail
```

- Method 2

Remarks

- In addition to differences in syntax, the SQL products also differ in the functionality of triggers. Some interesting questions are:
  - Can multiple triggers be defined for a single table and a specific transaction?
    Sequence problems that can affect the result
  - Can processing a statement belonging to a trigger action trigger another trigger?
    One mutation in an application can lead to a waterfall of mutations, recursion
  - When exactly is a trigger action processed?
    Immediately after the change or before the commit statement
  - Can triggers be defined on catalog tables?

Exercises

```
-- Exercise 1
-- Create a trigger that, when adding a new employee, sets the reportsTo attribute
-- to the employee to whom the least number of employees already report.




-- Testcode
BEGIN TRANSACTION
INSERT INTO Employees(LastName,FirstName)
VALUES ('New','Emplo');

SELECT EmployeeID, LastName, FirstName, ReportsTo
FROM Employees
ROLLBACK
```

Exercises

```sql
-- Exercise 2
/*
Create a new table called ProductsAudit with the following columns:
AuditID --> Primary Key + Identity
UserName --> NVARCHAR(128) + Default value = SystemUser
CreatedAt --> DateTime + Default value = UTC Time
Operation --> NVARCHAR(10): The name of the operation we performed on a row (Updated, Created, Deleted)

If the table is already present, drop it.
Create a trigger for all actions (Update, Delete, Insert) to persist the mutation of the Products table.
Use system functions to populate the UserName and CreatedAt.
*/


-- TestCode
BEGIN TRANSACTION
DECLARE @productId INT;
SET @productId = 100;
INSERT INTO Products(ProductName, Discontinued) VALUES('New product100', 0)
UPDATE Products SET productName = 'abc' WHERE ProductID = @productId
DELETE FROM Products WHERE ProductID = @productId
SELECT * FROM ProductsAudit -- Changes should be seen here.
ROLLBACK
```

Tables and User Defined Types

HO
GENT

User defined types

- ~Abstract data types
- Can be used as built-in types
- 2 sorts
  - Distinct types
  - Structured types: tables
- Despite the SQL-2008-standard, many differences between DBMS's

Distinct type

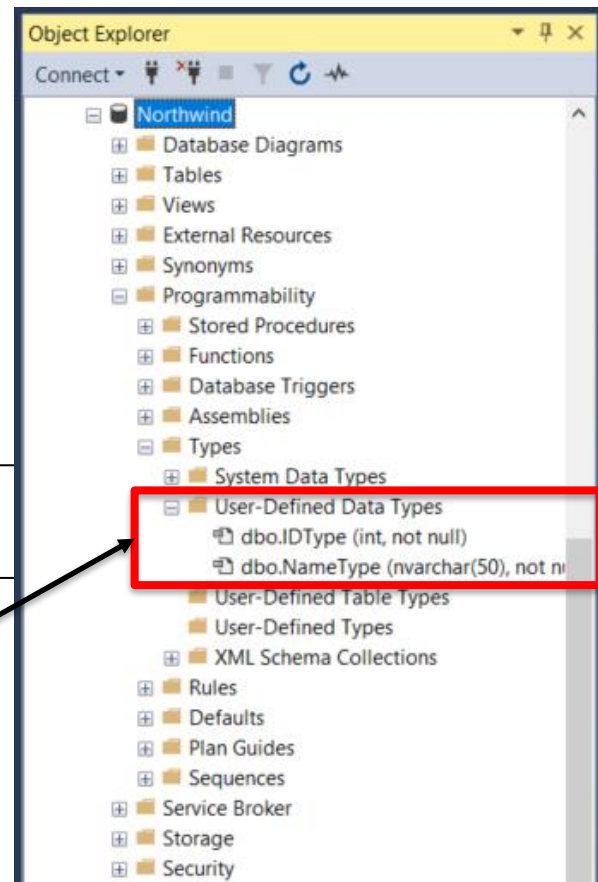- Based on a basic type

- Allows to refine existing data types

```sql
CREATE TYPE IDType FROM INT NOT NULL;
CREATE TYPE NameType FROM NVARCHAR(50) NOT NULL;
```

Distinct Type

Basic type of which distinct type inherits

- *The new type is stored in the database*
- *The distinct type can be used as if it were a built-in type*

Object Explorer

Connect

- Northwind
  - Database Diagrams
  - Tables
  - Views
  - External Resources
  - Synonyms
  - Programmability
    - Stored Procedures
    - Functions
    - Database Triggers
    - Assemblies
    - Types
      - System Data Types
      - User-Defined Data Types
        - dbo.IDType (int, not null)
        - dbo.NameType (nvarchar(50), not n...
      - User-Defined Table Types
      - User-Defined Types
      - XML Schema Collections
    - Rules
    - Defaults
    - Plan Guides
    - Sequences
  - Service Broker
  - Storage
  - Security

Distinct type: Use

- Ensure that all person names in the database use the same type

- The distinct type can be used throughout the database

```
CREATE TABLE PERSON
(id    IDType PRIMARY KEY,
 name NameType);
```

Distinct type: Drop

- Remove from the database

```
DROP Type IDType;
```

- The DROP TYPE statement will not execute when any of the following is true:
  - There are tables in the database that contain columns of the alias data type or the user-defined type
  - There are computed columns, CHECK constraints, schema-bound views
  - There are functions, stored procedures, or triggers created in the database, and these routines use variables or parameters of the alias or user-defined type

```
ALTER Type [Name of the type] is not possible.
```

Table variables

- Local temporary tables: #table

- Global temporary tables: ##table

- Table variables: @table

- Table types

Table variables > Local temporary tables

- Stored in tempdb under "System Databases"
- Only visible:
  - In the creating session
  - At the creating level
  - In all underlying levels
- Removed if creating level goes out-of-scope

Table variables > Local temporary tables

```sql
IF OBJECT_ID('tempdb.dbo.#OrderTotalsByYear') IS NOT NULL
DROP TABLE dbo.#OrderTotalsByYear;

CREATE TABLE #OrderTotalsByYear
(OrderYear INT NOT NULL PRIMARY KEY,
TotalQuantity INT NOT NULL);

INSERT INTO #OrderTotalsByYear(OrderYear, TotalQuantity)
SELECT YEAR(o.OrderDate) AS orderyear, SUM(od.Quantity) AS qty
FROM Orders AS o JOIN OrderDetails AS od ON o.orderid = od.orderid
GROUP BY YEAR(orderdate);

SELECT cur.OrderYear, cur.TotalQuantity AS TotalQuantity, prv.TotalQuantity AS
TotalQuantityPreviousYear
FROM dbo.#OrderTotalsByYear AS cur LEFT OUTER JOIN dbo.#OrderTotalsByYear AS prv
ON cur.OrderYear = prv.OrderYear + 1;
```

	OrderYear	TotalQuantity
1	2016	9581
2	2017	25489
3	2018	16247

	OrderYear	TotalQuantity	TotalQuantityPreviousYear
1	2016	9581	NULL
2	2017	25489	9581
3	2018	16247	25489

Table variables > Global temporary tables

- Stored in tempdb under "System Databases"
- Visible in all sessions
- Removed if creating session disconnects and there are no more references

Table variables > Global temporary tables

```sql
CREATE TABLE ##ExampleGlobals
(id INT NOT NULL PRIMARY KEY,
 username VARCHAR(50) NOT NULL);

INSERT INTO ##ExampleGlobals(id, username)
VALUES(5, 'jjanssens');

SELECT *
FROM ##ExampleGlobals;

DROP TABLE ##ExampleGlobals;
```

	id	username
1	5	jjanssens

Table variables > @table

```
DECLARE @OrderTotalsByYear AS TABLE
(OrderYear INT NOT NULL PRIMARY KEY,
TotalQuantity INT NOT NULL);
```

Table Types

```
CREATE TYPE TotalOrdersPerYear AS TABLE
(OrderYear INT NOT NULL PRIMARY KEY,
TotalQuantity INT NOT NULL);

DECLARE @TotOrdersPerYear AS TotalOrdersPerYear;

INSERT INTO @TotOrdersPerYear
SELECT YEAR(o.OrderDate) AS orderyear, SUM(od.Quantity) AS qty
FROM Orders AS o JOIN OrderDetails AS od ON o.orderid = od.orderid
GROUP BY YEAR(orderdate);

SELECT * FROM @TotOrdersPerYear;
```

	OrderYear	TotalQuantity
1	2016	9581
2	2017	25489
3	2018	16247

Table Types and Variables

- Table types are stored in the DB

- Table variables only exist during batch executions (= sequence of statements)

- Advantages of table variables and table types:

  - Shorter and cleaner code

  - Table type variables can also be passed as parameters to stored procedures and functions

Local temporary tables vs. table variables

- Both are only visible in creating session
- Table variables have more limited scope:
  - Only visible in current batch
  - Not visible in other batches in same session

Local temporary tables vs. table variables

```
IF OBJECT_ID('tempdb.dbo.#OrderTotalsByYear') IS NOT NULL
DROP TABLE dbo.#OrderTotalsByYear;

CREATE TABLE #OrderTotalsByYear
(OrderYear INT NOT NULL PRIMARY KEY,
TotalQuantity INT NOT NULL);

DECLARE @OrderTotalsByYear AS TABLE
(OrderYear INT NOT NULL PRIMARY KEY,
TotalQuantity INT NOT NULL);

SELECT * FROM #OrderTotalsByYear
SELECT * FROM @OrderTotalsByYear

GO -- execute previous commands and start new batch (same as button "Execute" in SSMS)

SELECT * FROM #OrderTotalsByYear
SELECT * FROM @OrderTotalsByYear -- Must declare the table variable @OrderTotalsByYear
```

Temporary tables: Example

- We want to delete all orders that contain products from a given supplier

- We first have to delete the orderdetails because of the FK constraint with orders

- But after deleting the orderdetails we loose the link with the supplier and don't know which orders to delete anymore

- Solution: save the orderid's in a temporary table

Temporary tables: Example

```sql
CREATE OR ALTER PROCEDURE DeleteOrdersSupplier @supplierid INT, @nrdeletedorders INT OUTPUT,
@nrdeletedorderdetails INT OUTPUT AS
BEGIN
SET NOCOUNT ON
CREATE TABLE #OrdersSupplier (orderid INT)-- table with 1 column (orderid)

-- insert the OrderID's of the orders with products from the UK that haven't been shipped yet
INSERT INTO #OrdersSupplier
SELECT DISTINCT od.OrderID FROM OrderDetails od JOIN Orders o on od.OrderID = o.OrderID
WHERE ProductID IN
(SELECT ProductID FROM Products WHERE SupplierID = @supplierid);

-- First delete the OrderDetails
DELETE FROM OrderDetails WHERE OrderID IN (SELECT OrderID FROM #OrdersSupplier)
-- How many orderdetails were deleted?
SET @nrdeletedorderdetails = @@rowcount
-- Afterwards delete the Orders
DELETE FROM Orders WHERE OrderID IN (SELECT OrderID FROM #OrdersSupplier)
-- How many orders were deleted?
SET @nrdeletedorders = @@rowcount
END
```

	OrderID
1	10255
2	10258
3	10264
4	10285
5	10289
6	10294
7	10298
8	10317
9	10327
10	10335
11	10342
12	10348
13	10354
14	10370

Temporary tables: Example

```
-- Test of procedure DeleteOrdersSupplier
BEGIN TRANSACTION

-- Choose a value for SupplierID
DECLARE @supplid INT = 10

-- Show which orders should be deleted
SELECT DISTINCT od.OrderID FROM OrderDetails od JOIN Orders o on od.OrderID = o.OrderID
WHERE ProductID IN (SELECT ProductID FROM Products WHERE SupplierID = @supplid)

DECLARE @nroforders INT
DECLARE @nroforderdetails INT
EXEC DeleteOrdersSupplier @supplid, @nroforders OUT, @nroforderdetails OUT
PRINT 'Nr of deletedorderdetails = ' + str(@nroforderdetails)
PRINT 'Nr of deletedorders = ' + str(@nroforders)

-- Check if the orders are deleted from the database
SELECT DISTINCT od.OrderID FROM OrderDetails od JOIN Orders o on od.OrderID = o.OrderID
WHERE ProductID IN (SELECT ProductID FROM Products WHERE SupplierID = @supplid)
ROLLBACK
```