Relational Databases and Datawarehousing – SQL

Subqueries, Insert-Update-Delete-Merge, Views,
Common Table Expressions

SUBQUERIES

SUBQUERIES basic form

- Nested subqueries
  - Basic form

```
SELECT
FROM
WHERE condition
```

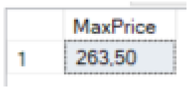
$\longrightarrow$ Contains in its left and/or right hand side statement another SELECT

    - Outer level query = the first SELECT. This is the main question.
    - Inner level query = the SELECT in the WHERE clause (or HAVING clause). This is the subquery:
      - Always executed first
      - Always between ().
      - Subqueries can be nested at > 1 level.
  - A subquery can return one value or a list of values

SUBQUERY that returns a single value

- The result of the query can be used anywhere you can use an expression.
  - With all relational operators: =, >, <, <=,>=,<>
  - Example:
    - What is the UnitPrice of the most expensive product?

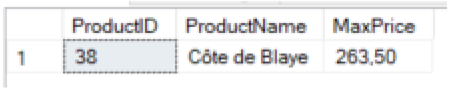
```
SELECT MAX(UnitPrice) As MaxPrice FROM Products
```



	MaxPrice
1	263,50

- What is the most expensive product?

```
SELECT ProductID, ProductName, UnitPrice As MaxPrice
FROM Products
WHERE UnitPrice = (SELECT MAX(UnitPrice) FROM Products)
```



	ProductID	ProductName	MaxPrice
1	38	Côte de Blaye	263,50

- Returns the previous query always the same resultset as the following?

```
SELECT TOP 1 ProductID, ProductName, UnitPrice As MaxPrice
FROM Products ORDER BY UnitPrice DESC
```

First the table Products is searched to determine the highest salary (= subquery). Then the table Products is searched a second time (= main query) to evaluate each unitprice against the determined maximum.

SUBQUERY that returns a <u>single value</u>

- Other examples
  - Give the products that cost more than average

```
SELECT ProductID, ProductName, UnitPrice As MaxPrice
FROM Products
WHERE UnitPrice > (SELECT AVG(UnitPrice) FROM Products)
```

  - Who is the youngest employee from the USA?

```
SELECT LastName, FirstName
FROM Employees
WHERE Country = 'USA'
AND BirthDate = (SELECT MAX(BirthDate) FROM Employees WHERE Country = 'USA')
```

SUBQUERY that returns a single column

- The resulting column can be used as a list
  - Operators IN, NOT IN, ANY, ALL
  - IN operator (=ANY operator)
    - Example: Give all employees that have processed orders

```
SELECT e.EmployeeID, e.FirstName + ' ' + e.LastName As Name
FROM Employees e
WHERE e.EmployeeID IN (SELECT DISTINCT EmployeeID FROM Orders)
```

    - This can also be accomplished with JOIN

```
SELECT DISTINCT e.EmployeeID, e.FirstName + ' ' + e.LastName As Name
FROM Employees e JOIN Orders o ON e.EmployeeID = o.EmployeeID
```

SUBQUERY that returns a single column

- The resulting column can be used as a list
  - Operators IN, NOT IN, ANY, ALL
  - NOT IN operator
    - Example: Give all customers that have not placed any orders yet

```
SELECT *
FROM Customers
WHERE CustomerID NOT IN (SELECT DISTINCT CustomerID FROM Orders)
```

    - This can also be accomplished with JOIN

```
SELECT *
FROM Customers c LEFT JOIN Orders o ON c.CustomerID = o.CustomerID
WHERE o.CustomerID is NULL
```

ANY and ALL keywords

- These keywords are used in combination with the relational operators and subqueries that return a column of values

ANY and ALL keywords

- ALL returns TRUE if all values returned in the subquery satisfy the condition
  - ALL operator → >ALL means greater than every value
    - Example: Give all products that are more expensive than the most expensive product with CategoryName = 'Seafood'

```sql
SELECT *
FROM Products
WHERE UnitPrice > ALL(SELECT p.UnitPrice FROM Products p INNER JOIN Categories c ON
p.CategoryID = c.CategoryID AND c.CategoryName = 'Seafood')
```

    - This can also be accomplished without ALL

```sql
SELECT *
FROM Products
WHERE UnitPrice > (SELECT MAX(UnitPrice) FROM Products p INNER JOIN Categories c ON
p.CategoryID = c.CategoryID AND c.CategoryName = 'Seafood')
```

ANY and ALL keywords

- ANY returns TRUE if at least one value returned in the subquery satisfies the condition
  - >ANY means greater than at least one value
    - Example: Give all products that are more expensive than one of the products with CategoryName = 'Seafood'

```
SELECT *
FROM Products
WHERE UnitPrice > ANY(SELECT p.UnitPrice FROM Products p INNER JOIN Categories c ON
p.CategoryID = c.CategoryID AND c.CategoryName = 'Seafood')
```

    - This can also be accomplished without ANY

```
SELECT *
FROM Products
WHERE UnitPrice > (SELECT MIN(p.UnitPrice) FROM Products p INNER JOIN Categories c ON
p.CategoryID = c.CategoryID AND c.CategoryName = 'Seafood')
```

Correlated subqueries

- In a correlated subquery the inner query depends on information from the outer query.
  The subquery contains a search condition that refers to the main query, which makes the subquery depends on the main query
- The subquery is executed for each row in the main query. => $O(n^2)$
- The order of execution is from top to bottom, not from bottom to top as in a simple subquery, which is $O(n)$.

Correlated subqueries

- For performance reasons use joins or simple subquery if possible
- Principle

```
SELECT …
FROM table a
WHERE expression operator (SELECT ...
                            FROM table
                            WHERE expression operator a.columnname)
```

Correlated subqueries

- Example: Give employees with a salary larger than the average salary

```
SELECT FirstName + ' ' + LastName As FullName, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees)
```

This is a completely different (and more complex) task.

- Example: Give the employees whose salary is larger than the average of the salary of the employees <u>who report to the same boss</u>.

```
SELECT FirstName + ' ' + LastName As FullName, ReportsTo, Salary
FROM Employees As e
WHERE Salary > (SELECT AVG(Salary) FROM Employees WHERE ReportsTo = e.ReportsTo)
```

<u>Remark</u>: in the inner query you can use fields from the tables in the outer query but NOT vice versa.

0. Row 1 in the outer query

1. Outer query passes column values for that row to inner query

2. Inner query use those values to evaluate inner query.

3. Inner query returns value to outer query, which decides if row in outer query will be kept.

4. This process repeats for each row in outer query.

Back to step 1.

HO
GENT

Subqueries and the EXISTS operator

- The operator EXISTS tests the existence of a result set
  - Example: Give all customers that already placed an order

```
SELECT *
FROM Customers As c
WHERE EXISTS
(SELECT * FROM Orders WHERE CustomerID = c.customerID)
```

- There is also NOT EXISTS
  - Example: Give all customers that have not placed any orders yet

```
SELECT *
FROM Customers As c
WHERE NOT EXISTS
(SELECT * FROM Orders WHERE CustomerID = c.customerID)
```

3 ways to accomplish the same result

- Example: Give all customers that did not place any orders yet
  - OUTER JOIN

```
SELECT *
FROM Customers c LEFT JOIN Orders o ON c.CustomerID = o.CustomerID
WHERE o.CustomerID is NULL
```

Which one will, in general, be the slowest?

  - Simple subquery

```
SELECT *
FROM Customers
WHERE CustomerID NOT IN (SELECT DISTINCT CustomerID FROM Orders)
```

  - Correlated subquery

```
SELECT *
FROM Customers As c
WHERE NOT EXISTS
(SELECT * FROM Orders WHERE CustomerID = c.customerID)
```

HO
GENT

Subqueries in the FROM-clause

- Since the result of a query is a table it can be used in the FROM-clause.
- In MS-SQL Server the table in the subquery must have a name.
  You can optionally also rename the columns
- A subquery in the FROM clause is called a derived table.

Subqueries in the FROM-clause

- Example: Give per region the total sales (region USA+Canada = North America, rest = Rest of World).

```
-- Solution 1
SELECT
CASE c.Country
WHEN 'USA' THEN 'Northern America'
WHEN 'Canada' THEN 'Northern America'
ELSE 'Rest of world'
END AS Regionclass, COUNT(o.OrderID) As
NumberOfOrders
FROM Customers c JOIN Orders o
ON c.CustomerID = o.CustomerID
GROUP BY
CASE c.Country
WHEN 'USA' then 'Northern America'
WHEN 'Canada' then 'Northern America'
ELSE 'Rest of world'
END
```

```
-- Solution 2 -> avoid copy-paste
(subquery in FROM)
SELECT Regionclass, COUNT(OrderID)
FROM
(
SELECT
CASE c.Country
WHEN 'USA' THEN 'Northern America'
WHEN 'Canada' THEN 'Northern America'
ELSE 'Rest of world'
END AS Regionclass, o.OrderID
FROM Customers c JOIN Orders o
ON c.CustomerID = o.CustomerID
)
AS Totals(Regionclass, OrderID)
GROUP BY Regionclass
```

HO
GENT

Subqueries in the SELECT-clause

- In a SELECT clause scalar (simple or correlated) subqueries can be used
  - Example: Give for each employee how much they earn more (or less) than the average salary of all employees with the same supervisor.

```
SELECT Lastname, Firstname, Salary,
Salary -
(
    SELECT AVG(Salary)
    FROM Employees
    WHERE ReportsTo = e.ReportsTo
)
FROM Employees e
```

Subqueries in the SELECT- and FROM-clause

– Example: Give per category the price of the cheapest product and a product that has that price.

```
SELECT Category, MinUnitPrice,
(
    SELECT TOP 1 ProductID
    FROM Products
    WHERE CategoryID = Category AND UnitPrice = MinUnitPrice
) As ProductID
FROM
(
    SELECT CategoryID, MIN(UnitPrice)
    FROM Products p
    GROUP BY CategoryID
) AS CategoryMinPrice(Category, MinUnitPrice)
```

Application: running totals

– Example: Give the cumulative sum of freight per year

```
SELECT OrderID, OrderDate, Freight,
(
SELECT SUM(Freight)
FROM Orders
WHERE YEAR(OrderDate) = YEAR(o.OrderDate) and OrderID <= o.OrderID
) As TotalFreight
FROM Orders o
ORDER BY Orderid;
```

Some exercises

```
-- 1. Give the id and name of the products that have not been purchased yet.
-- 2. Select the names of the suppliers who supply products that have not been ordered yet.
-- 3. Give a list of all customers from the same country as the customer Maison Dewey
-- 4. Calculate how much is earned by the management (like 'president' or 'manager'), the submanagement
(like 'coordinator') and the rest
-- 5. Give for each product how much the price differs from the average price of all products of the same
category
-- 6. Give per title the employee that was last hired
-- 7. Which employee has processed most orders?
```

DML

SQL - DML basic tasks

- SELECT → consulting data
- INSERT → adding data
- UPDATE → changing data
- DELETE → removing data
- MERGE → combine INSERT, UPDATE and DELETE

Tip for not destroying your database

- The statements in this chapter are destructive.
- SQL has no UNDO by default!
- BUT you can 'simulate' UNDO if you take precautions.

Transactions are discussed in detail in one of the next chapters.

```
/* Tip for not destroying your database */
BEGIN TRANSACTION -- starts a new "transaction" -> Saves previous state of DB in buffer

-- several "destructive" commands can go here:
INSERT INTO Products(ProductName)
values ('TestProduct');

-- only you (in  your session) can see changes
SELECT * FROM Products WHERE ProductID = (SELECT MAX(ProductID) FROM Products)

ROLLBACK;   --> ends transaction and restores database in previous state
-- COMMIT;  --> ends transaction and makes changes permanent
```

DML

INSERT: add new records

HO
GENT

Adding data - INSERT

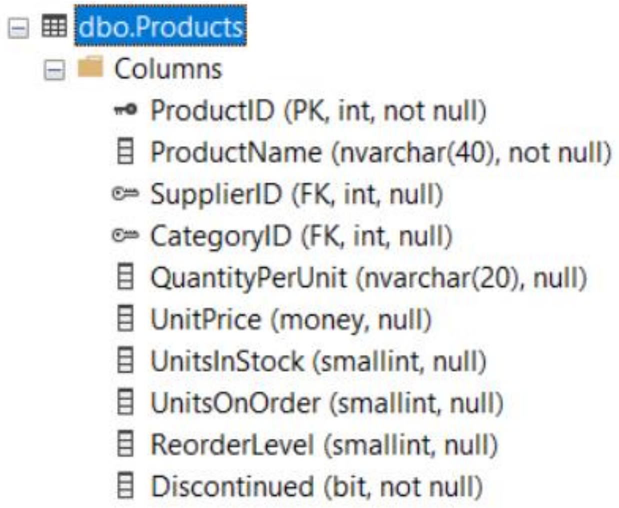
- The INSERT statement adds data in a table
  - Add one row through via specification
  - Add selected row(s) from other tables

INSERT of 1 row

- Method 1: specify only the (not NULL) values for specific columns.
- Method 2: specify all column values
- If the identity is generated automatically, this column can't be mentioned.

```
INSERT INTO Products (ProductName, CategoryID, Discontinued)
VALUES ('Toblerone', 3, 0)

INSERT INTO Products
VALUES ('Sultana', null, 3, null, null, null, null, null, 1)
```

HO
GENT


dbo.Products
Columns
ProductID (PK, int, not null)
ProductName (nvarchar(40), not null)
SupplierID (FK, int, null)
CategoryID (FK, int, null)
QuantityPerUnit (nvarchar(20), null)
UnitPrice (money, null)
UnitsInStock (smallint, null)
UnitsOnOrder (smallint, null)
ReorderLevel (smallint, null)
Discontinued (bit, not null)

INSERT of 1 row

- The number of specified columns corresponds to the number of values.
- The specified values and correspond ding columns have compatible data types.
- If no column names are specified the values are assigned in the column order as specified by the CREATE TABLE statement.
- Unmentioned columns get the value NULL or the DEFAULT value if any.
- NULL can also be specified as a value.

HO
GENT

INSERT of rows selected from other tables

- Mandatory fields have to be specified, unless they have a DEFAULT value.
- Constraints (see further) are validated.
- Unmentioned columns get the value NULL or the DEFAULT value if any.

```sql
INSERT INTO Customers (CustomerID, ContactName, ContactTitle, CompanyName)
SELECT substring(FirstName,1,2) + substring(LastName,1,3), FirstName + ' ' +
LastName, Title, 'EmployeeCompany'
FROM Employees
```

DML

UPDATE: modify values

HO
GENT

Changing data - UPDATE

- Changing all rows in a table
  - Example: Increase the price of all products with 10%

```
UPDATE Products
SET UnitPrice = UnitPrice * 1.1
```

- Changing 1 row or a group of rows
  - Example: Increase the price of all the 'Bröd' products with 10%

```
UPDATE Products
SET UnitPrice = UnitPrice * 1.1
WHERE ProductName LIKE '%Bröd%'
```

  - Example: Increase the price of all the 'Bröd' products with 10% and set all units in stock to 0

```
UPDATE Products
SET UnitPrice = UnitPrice * 1.1, UnitsInStock = 0
WHERE ProductName LIKE '%Bröd%'
```

HO GENT

Changing data - UPDATE

- Change rows based on data in another table
  - Standard SQL does not offer JOINs in an update statement → you can only use subqueries to refer to another table
  - Example: Due to a change in the euro – dollar exchange rate, we have to increase the unit price of products delivered by suppliers from the USA by 10%.

```
UPDATE Products
SET UnitPrice = (UnitPrice * 1.1)
WHERE SupplierID IN
(SELECT SupplierID FROM Suppliers WHERE Country = 'USA')
```

subquery

HO
GENT

DML

DELETE: remove records

Removing data - DELETE

- Delete some rows
  - Example: Delete the 'Bröd' products

```
DELETE FROM Products
WHERE ProductName LIKE '%Bröd%'
```

- Delete all rows in a table
  - via DELETE the identity values continues

```
-- the identity value continues
DELETE FROM Products
```

  - via TRUNCATE the identity value restarts from 1
    TRUNCATE is also more performant, but does not offer WHERE clause: it's all or nothing

```
-- the identity value restarts from 1
TRUNCATE TABLE Products
```

HO
GENT

DELETE - based on data in another table

- Change rows based on data in another table
  - Again no JOIN, only subquery
  - Example: Delete the orderdetails for all orders from the most recent orderdate

```
DELETE FROM OrderDetails
WHERE OrderID IN
(SELECT OrderID FROM Orders WHERE OrderDate = (SELECT MAX(OrderDate) from Orders))
```

DML

MERGE: combine INSERT, UPDATE, DELETE

Merge

- With MERGE you can combine INSERT, UPDATE and DELETE.
- Very common use case: users work on an Excel sheet to update a relatively large amount of records because Excel offers a better overview than their ERP tool.
- They can update records, add new ones and delete records in Excel.
- After uploading the edited Excel file to a temporary table, the merge statement performs all UPDATEs, INSERTs and DELETEs at once.

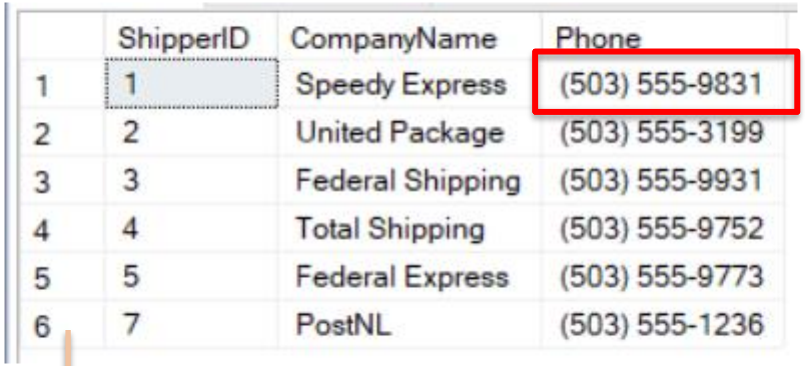
Merge

- Script to create temporary table

```sql
/* First execute following script to simulate the Excel file that has been imported
to a temporary table ShippersUpdate */

DROP TABLE IF EXISTS ShippersUpdate;
-- Add everything from Shippers to ShippersUpdate
SELECT * INTO ShippersUpdate FROM Shippers

-- Add an extra record to ShippersUpdate
INSERT INTO ShippersUpdate VALUES ('Pickup','(503) 555-9647')

-- Update a record of ShippersUpdate
UPDATE ShippersUpdate SET Phone = '(503) 555-4512' WHERE ShipperID = 1

-- Remove a record from ShippersUpdate
DELETE FROM ShippersUpdate WHERE shipperID = 4
```

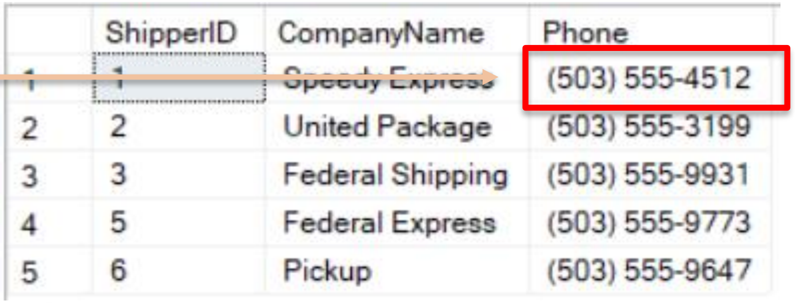
Merge

- Original table Shippers



	ShipperID	CompanyName	Phone
1	1	Speedy Express	(503) 555-9831
2	2	United Package	(503) 555-3199
3	3	Federal Shipping	(503) 555-9931
4	4	Total Shipping	(503) 555-9752
5	5	Federal Express	(503) 555-9773
6	7	PostNL	(503) 555-1236

Temporary table ShippersUpdate



	ShipperID	CompanyName	Phone
1	1	Speedy Express	(503) 555-4512
2	2	United Package	(503) 555-3199
3	3	Federal Shipping	(503) 555-9931
4	5	Federal Express	(503) 555-9773
5	6	Pickup	(503) 555-9647

Last row is added to Shippers at the beginning of the script on the next slide

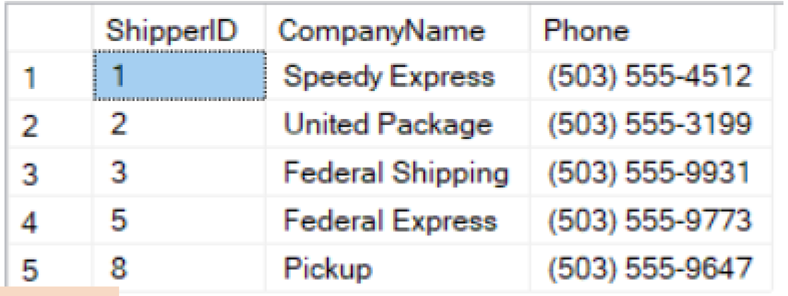
Total Shipping is not in ShippersUpdate
Pickup is in ShippersUpdate and not in Shippers

Merge

```
BEGIN TRANSACTION

INSERT INTO Shippers
VALUES ('PostNL', '(503) 555-1236')

SELECT * FROM Shippers;
SELECT * FROM ShippersUpdate;
```

Merge

```
MERGE Shippers as t -- t = target
USING ShippersUpdate as s -- s = source
ON (t.ShipperID = s.ShipperID)

-- Which rows are in source and have different values for CompanyName or Phone?
-- Update those rows in target with the values coming from source
WHEN MATCHED AND t.CompanyName <> s.CompanyName OR ISNULL(t.Phone,'') <> ISNULL(s.Phone,'')
THEN UPDATE SET t.CompanyName = s.CompanyName, t.Phone=s.Phone

-- Which rows are in target and not in source?
-- Add those rows to source
WHEN NOT MATCHED BY target --> new rows
THEN INSERT (CompanyName, Phone) VALUES (s.CompanyName,s.Phone)

-- Which rows are in source and not in target?
-- Delete those rows from target
WHEN NOT MATCHED BY source  --> rows to delete
THEN DELETE;

-- Check the result
SELECT * FROM Shippers
ROLLBACK;
```



	ShipperID	CompanyName	Phone
1	1	Speedy Express	(503) 555-4512
2	2	United Package	(503) 555-3199
3	3	Federal Shipping	(503) 555-9931
4	5	Federal Express	(503) 555-9773
5	8	Pickup	(503) 555-9647

Remark: the option tot delete records is a non standard extension by MS SQL.

Views

Views – Introduction

- Definition
  - A view is a saved SELECT statement
  - A view can be seen as a virtual table composed of other tables & views
  - No data is stored in the view itself, at each referral the underlying SELECT is re-executed;

Views – Introduction

- Advantages
  - Hide complexity of the database
    - Hide complex database design
    - Make large and complex queries accessible and reusable
    - Can be used as a partial solution for complex problems
  - Used for securing data access: revoke access to tables and grant access to customised views.
  - Organise data for export to other applications

HO
GENT

Definition of a view

```
CREATE VIEW view_name [(column_list)]AS select_statement
[with check option]
```

- number of columns in (column_list) = # colunms in select
  - If no column names are specified, they are taken from the select
  - Column names are mandatory if the select statement contains calculations or joins in which some column names appear more than once
- the select statement may not contain an order by
- with check option: in case of mutation through the view (insert, update, delete) it is checked if the new data also conforms to the view conditions

Views – CRUD operations

- Creating a view

```
-- Creating a view
CREATE VIEW V_ProductsCustomer(productcode, company, quantity)
AS SELECT od.ProductID, c.CompanyName, sum(od.Quantity)
FROM Customers c
JOIN Orders o ON o.CustomerID = c.CustomerID
JOIN OrderDetails od ON o.OrderID = od.OrderID
GROUP BY od.ProductID, c.CompanyName;
```

- Using a view

```
-- Using a view
SELECT * FROM V_ProductsCustomer;
```

- Changing a view

```
-- Changing a view
ALTER VIEW V_ProductsCustomer(productcode, company)
AS SELECT od.ProductID, c.CompanyName
FROM Customers c
JOIN Orders o ON o.CustomerID = c.CustomerID
JOIN OrderDetails od ON o.OrderID = od.OrderID
GROUP BY od.ProductID, c.CompanyName;
```

- Deleting a view

```
-- Dropping a view
DROP VIEW V_ProductsCustomer;
```

Example: views as partial solution for complex problems

- Example: Create a view with the number of orders per employee per year
- Example: Calculate per employee and per year the running total of processed orders.

```sql
-- Create a view with the number of orders per employee per year
CREATE VIEW vw_number_of_orders_per_employee_per_year
AS
SELECT EmployeeID, YEAR(OrderDate) As OrderYear, COUNT(OrderID) As NumberOfOrders
FROM Orders
GROUP BY EmployeeID, YEAR(OrderDate)

-- Check the result
SELECT * FROM vw_number_of_orders_per_employee_per_year
```

Example: views as partial solution for complex problems

```
-- Add the running total. Use the view.
SELECT EmployeeID, OrderYear, NumberOfOrders,
(SELECT SUM(NumberOfOrders)
FROM vw_number_of_orders_per_employee_per_year
WHERE OrderYear <= vw.OrderYear AND EmployeeID = vw.EmployeeID
) As TotalNumberOfOrders
FROM vw_number_of_orders_per_employee_per_year vw
ORDER BY EmployeeID, OrderYear
```

- Drawback of using views in this way:
  views are stored in the database and might create a mess if you have hundreds of them.
  We'll try to solve this using cte's.

Update of views

- An updatable view
  - Has no distinct or top clause in the select statement
  - Has no statistical functions in the select statement
  - Has no calculated value in the select statement
  - Has no group by in the select statement
  - Does not use a union
- All other views are read-only views
- In general views are updatable if the system is able to translate the updates to individual records and fields in the underlying tables, so use your common sense.

Working with updatable views

- UPDATE
  - You can only update one table at once
  - without check option
    - After the update a row can disappear from the view
  - with check option
    - An error is generated if after the update the row would no longer be part of the view

Working with updatable views

- INSERT
  - You can only insert in one table
  - All mandatory columns have to appear in the view and the insert
    - identity columns with a NULL or DEFAULT constraint can be omitted
- DELETE
  - The delete can only be used with a VIEW based on exactly one table.

Working with updatable views

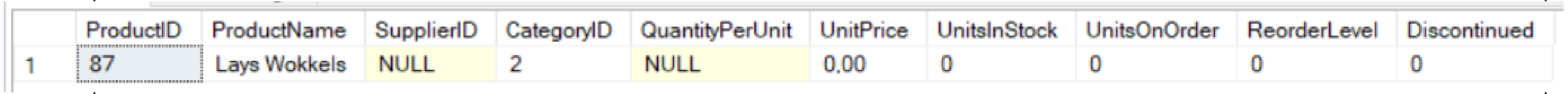
- Sometimes, you create a view to reveal the partial data of a table.
- However, a simple view is updatable therefore it is possible to update data which is not visible through the view.
- This update makes the view inconsistent.
- To ensure the consistency of the view, you use the WITH CHECK OPTION clause when you create or modify the view.

Working with updatable views

- The WITH CHECK OPTION is an optional clause of the CREATE VIEW statement.
- The WITH CHECK OPTION prevents a view from updating or inserting rows that are not visible through it.
- In other words, whenever you update or insert a row of the base tables through a view,
- SQLServer ensures that the insert or update operation is conformed with the definition of the view.

Views with/without check option: example

```
DROP VIEW IF EXISTS productsOfCategory1;
DROP VIEW IF EXISTS productsOfCategory1Bis;

-- Create a view without "with check option"
CREATE VIEW productsOfCategory1
AS SELECT * FROM Products WHERE CategoryID = 1

-- Insert product from CategoryID 2 => although Wokkels is from CategoryID = 2, it can be added through the
view
INSERT INTO productsOfCategory1 (ProductName, CategoryID)
VALUES ('Lays Wokkels', 2)

SELECT * FROM Products WHERE ProductName LIKE '%Wokkels%'
```

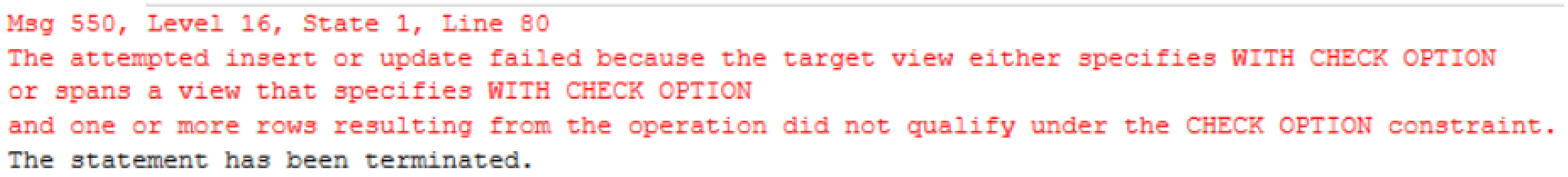


	ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel	Discontinued
1	87	Lays Wokkels	NULL	2	NULL	0,00	0	0	0	0

```
DELETE FROM Products
WHERE ProductName LIKE '%Wokkels%'
```

Views with/without check option: example

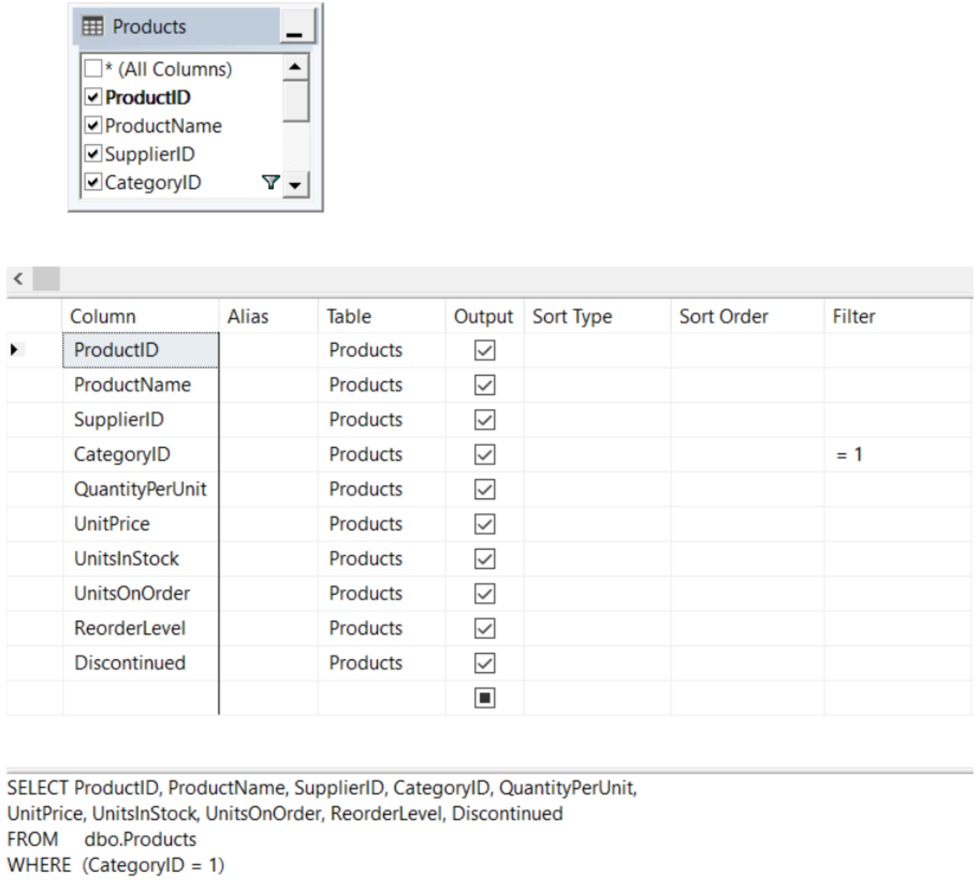
```sql
-- Create a view with "with check option"
CREATE VIEW productsOfCategory1Bis
AS SELECT * FROM Products WHERE CategoryID = 1
WITH CHECK OPTION

-- Insert product from producttype 2 => because Wokkels is from CategoryID = 2, it can't be inserted
through the view
INSERT INTO productsOfCategory1Bis (ProductName, CategoryID)
VALUES ('Lays Wokkels', 2)
```



```
Msg 550, Level 16, State 1, Line 80
The attempted insert or update failed because the target view either specifies WITH CHECK OPTION
or spans a view that specifies WITH CHECK OPTION
and one or more rows resulting from the operation did not qualify under the CHECK OPTION constraint.
The statement has been terminated.
```

Views in SQL Server Management Studio

- Simple views can also be made with the graphical user interface
- This is not possible for views with subqueries or common table expressions



Products

- [ ] * (All Columns)
- [x] **ProductID**
- [x] ProductName
- [x] SupplierID
- [x] CategoryID

Column	Alias	Table	Output	Sort Type	Sort Order	Filter
ProductID		Products	✓			
ProductName		Products	✓			
SupplierID		Products	✓			
CategoryID		Products	✓			= 1
QuantityPerUnit		Products	✓			
UnitPrice		Products	✓			
UnitsInStock		Products	✓			
UnitsOnOrder		Products	✓			
ReorderLevel		Products	✓			
Discontinued		Products	✓			
			■			

```
SELECT ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit,
UnitPrice, UnitsInStock, UnitsOnOrder, ReorderLevel, Discontinued
FROM     dbo.Products
WHERE  (CategoryID = 1)
```

Views in SQL Server Management Studio

- The easiest way to change the SQL code of an existing view is by right clicking on the name of the view and:
  - Script View as…
  - ALTER to…
  - New Query Editor Window

Exercises

```
-- Exercise 1
-- The company wants to weekly check the stock of their products.
-- If the stock is below 15, they'd like to order more to fulfill the need.

-- (1.1) Create a QUERY that shows the ProductId, ProductName and the name of the supplier, do not forget the
WHERE clause.
-- (1.2) Turn this SELECT statement into a VIEW called: vw_products_to_order.
-- (1.3) Query the VIEW to see the results.

-- Exercise 2
-- The company has to increase prices of certain products. To make it seem the prices are not increasing
dramatically they're planning to spread the price increase over multiple years. In total they'd like a 10%
price for certain products. The list of impacted products can grow over the coming years.
-- We'd like to keep all the logic of selecting the correct products in 1 SQL View, in programming terms
'keeping it DRY'.
-- The updating of the items is not part of the view itself.
-- The products in scope are all the products with the term 'Bröd' or 'Biscuit'.

-- (2.1) Create a simple SQL Query to get the correct resultset
-- (2.2) Turn this SELECT statement into a VIEW called: vw_price_increasing_products.
-- (2.3) Query the VIEW to see the results.
-- (2.4) Increase the price of the resultset of the VIEW: vw_price_increasing_products by 2%. Use a transaction
```

COMMON TABLE EXPRESSIONS

HO GENT

Common Table Expressions: the WITH component

- Example: Give the average number of orders for all customers
- Solution 1 → Using subqueries

```sql
-- Solution 1
-- First calculate the number of orders per customer (Subquery in FROM)
-- Afterwards calculate the AVG

SELECT AVG(numberOfOrders * 1.0) As AverageNumberOfOrdersPerCustomer -- * 1.0 to force floating point
FROM
(
SELECT customerid, COUNT(orderid)
FROM orders
GROUP BY customerid
) AS numberOfOrdersPerCustomer(customerid, numberOfOrders)
```

Common Table Expressions: the WITH component

- Solution 2 → Using CTE's (**C**ommon **T**able **E**xpression)
- Using the WITH-component you can give the subquery its own name (with column names) and reuse it in the rest of the query (possibly several times!)

```
-- Solution 2

WITH numberOfOrdersPerCustomer(customerid, numberOfOrders) AS
(SELECT customerid, COUNT(orderid)
FROM orders
GROUP BY customerid)

SELECT AVG(numberOfOrders * 1.0) As AverageNumberOfOrdersPerCustomer
FROM numberOfOrdersPerCustomer
```

Common Table Expressions: the WITH component

- the WITH-component has two application areas:
  - Simplify SQL-instructions, e.g. simplified alternative for simple subqueries or avoid repetition of SQL constructs
  - Traverse recursively hierarchical and network structures

CTE’s versus Views

- Similarities
  - WITH ~ CREATE VIEW
  - Both are virtual tables:
    the content is derived from other tables
- Differences
  - A CTE only exists during the SELECT-statement
  - A CTE is not visible for other users and applications

CTE’s versus Subqueries

- Similarities
  - Both are virtual tables:
    the content is derived from other tables
- Differences
  - A CTE can be reused in the same query
  - A subquery is defined in the clause where it is used (SELECT/FROM/WHERE/…)
  - A CTE is defined on top of the query
  - A simple subquery can always be replaced by a CTE

CTE’s to simplify queries

- Example: Give per category the minimum price and all products with that minimum price

```
-- Solution 1 -> with subqueries

SELECT CategoryID, ProductID, UnitPrice
FROM Products p
WHERE UnitPrice = (SELECT MIN(UnitPrice) FROM Products WHERE CategoryID = p.CategoryID)
```

```
-- Solution 2 -> with CTE's
WITH CategoryMinPrice(Category, MinPrice)
AS (SELECT CategoryID, MIN(UnitPrice)
    FROM Products AS p
    GROUP BY CategoryID)

SELECT Category, MinPrice, p.ProductID
FROM Products AS p
JOIN CategoryMinPrice AS c ON p.CategoryID = c.Category AND p.UnitPrice = c.MinPrice;
```

CTE’s with more than 1 WITH - component

- Example: Give per year per customer the relative contribution of this customer to the total revenue
- Step 1: Calculate the total revenue per year

```sql
-- Step 1 -> Total revenue per year

SELECT YEAR(OrderDate), SUM(od.UnitPrice * od.Quantity)
FROM Orders o INNER JOIN OrderDetails od
ON o.OrderID = od.OrderID
GROUP BY YEAR(OrderDate)
```

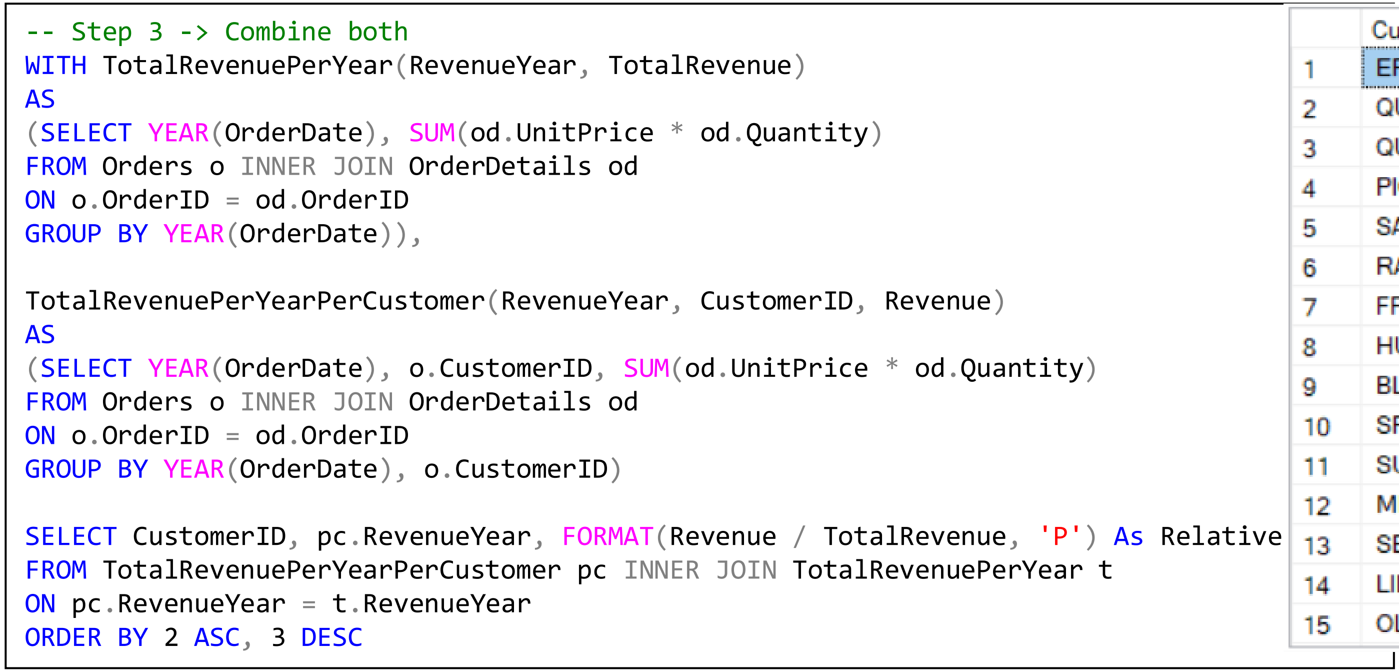
CTE’s with more than 1 WITH - component

- Step 2: Calculate the total revenue per year per customer

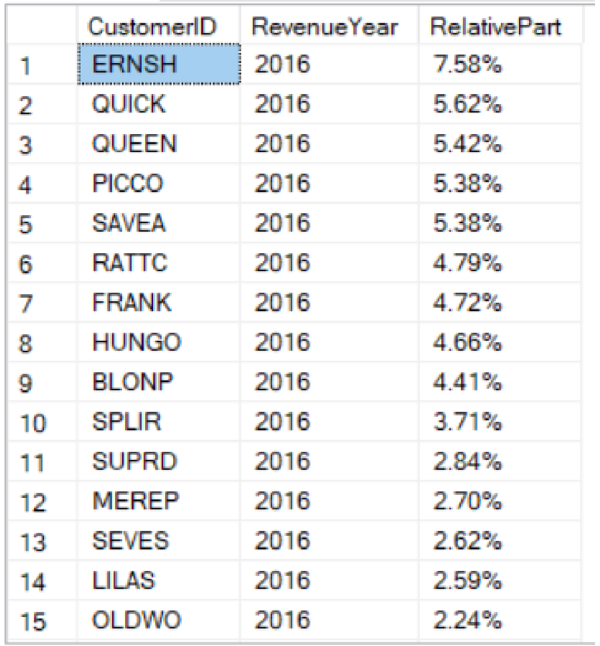
```
-- Step 2 -> Total revenue per year per customer

SELECT YEAR(OrderDate), o.CustomerID, SUM(od.UnitPrice * od.Quantity)
FROM Orders o INNER JOIN OrderDetails od
ON o.OrderID = od.OrderID
GROUP BY YEAR(OrderDate), o.CustomerID
```

CTE’s with more than 1 WITH - component

- Step 3: Combine both



```
-- Step 3 -> Combine both
WITH TotalRevenuePerYear(RevenueYear, TotalRevenue)
AS
(SELECT YEAR(OrderDate), SUM(od.UnitPrice * od.Quantity)
FROM Orders o INNER JOIN OrderDetails od
ON o.OrderID = od.OrderID
GROUP BY YEAR(OrderDate)),

TotalRevenuePerYearPerCustomer(RevenueYear, CustomerID, Revenue)
AS
(SELECT YEAR(OrderDate), o.CustomerID, SUM(od.UnitPrice * od.Quantity)
FROM Orders o INNER JOIN OrderDetails od
ON o.OrderID = od.OrderID
GROUP BY YEAR(OrderDate), o.CustomerID)

SELECT CustomerID, pc.RevenueYear, FORMAT(Revenue / TotalRevenue, 'P') As Relative
FROM TotalRevenuePerYearPerCustomer pc INNER JOIN TotalRevenuePerYear t
ON pc.RevenueYear = t.RevenueYear
ORDER BY 2 ASC, 3 DESC
```



	CustomerID	RevenueYear	RelativePart
1	ERNSH	2016	7.58%
2	QUICK	2016	5.62%
3	QUEEN	2016	5.42%
4	PICCO	2016	5.38%
5	SAVEA	2016	5.38%
6	RATTC	2016	4.79%
7	FRANK	2016	4.72%
8	HUNGO	2016	4.66%
9	BLONP	2016	4.41%
10	SPLIR	2016	3.71%
11	SUPRD	2016	2.84%
12	MEREP	2016	2.70%
13	SEVES	2016	2.62%
14	LILAS	2016	2.59%
15	OLDWO	2016	2.24%

Recursive SELECT's

- Recursivity means:
  - We continue to execute a table expression until a condition is reached.
- This allows you to solve problems like:
  - Who are the friends of my friends etc. (in a social network)?
  - What is the hierarchy of an organisation ?
  - Find the parts and subparts of a product (Bill of materials).

Recursive SELECT's

- Example:
  Give the integers from 1 to 5

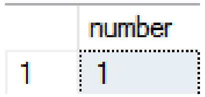
```
WITH numbers(number) AS
(SELECT 1
  UNION all
   SELECT number + 1
   FROM numbers
   WHERE number < 5)

SELECT * FROM numbers;
```

- Characteristics of recursive use of WITH:
  - The with component consists of (at least) 2 expressions, combined with union all
  - A temporary table is consulted in the second expression
  - At least one of the expressions may not refer to the temporary table.

Recursive SELECT’s: how does it work?

- 1. SQL searches the table expressions that don't contain recursivity and executes them one by one.

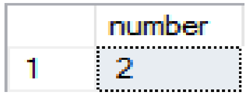
```sql
SELECT 1
```



	number
1	1

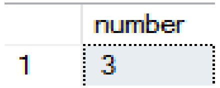
- 2. Execute all recursive expressoins. The numbers table, that got a value of 1 in step 1, is used.
  This row is added to the numbers table.

```sql
SELECT number + 1
FROM numbers
WHERE number < 5
```



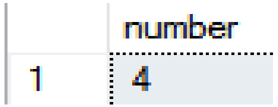
	number
1	2

Recursive SELECT's: how does it work?

- 3. Now the recursion starts: the 2nd expression is re-executed, giving as result:
  

	number
1	3

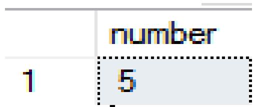
  Remark: not all rows added in the previous steps are processed, but only those rows (1 row in this example), that were added in the previous step (step 2).
- 4. Since step 3 also gave a result, the recursive expression is executed again, producing as intermediate result:



	number
1	4

Recursive SELECT’s: how does it work?

- 5. And this happens again:

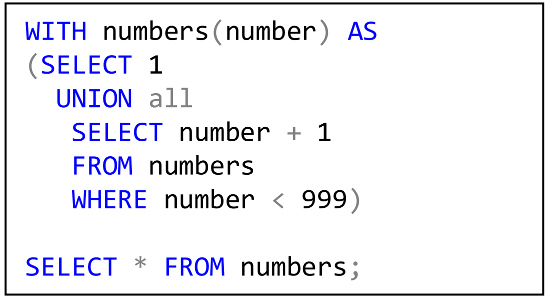


	number
1	5

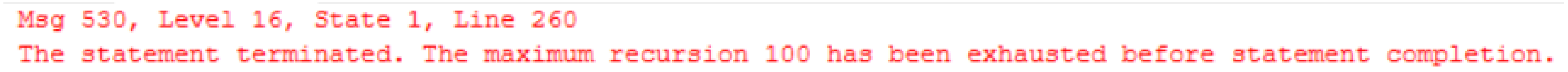
- If the expression is now processed again, it does not return a result, since in the previous step no rows were added that correspond to the condition number < 5.
  Here SQL stops the processing of the table expression and the final result is known.
- Summary: the 1st (non-recursive) expression is executed once and the 2nd expression is executed until it does not return any more results.

Recursive SELECT’s: max number of recursions = 100

- Example: Give the numbers from 1 to 999



```sql
WITH numbers(number) AS
(SELECT 1
  UNION all
   SELECT number + 1
   FROM numbers
   WHERE number < 999)

SELECT * FROM numbers;
```



```
Msg 530, Level 16, State 1, Line 260
The statement terminated. The maximum recursion 100 has been exhausted before statement completion.
```

Recursive SELECT’s: OPTION maxrecursion

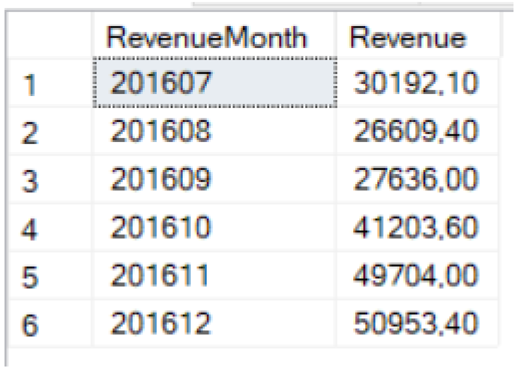
- Example: Give the numbers from 1 to 999

```
WITH numbers(number) AS
(SELECT 1
  UNION all
   SELECT number + 1
   FROM numbers
   WHERE number < 999)

SELECT * FROM numbers
OPTION (maxrecursion 1000);
```

Application: generate missing months

- Example: Give the total revenue per month in 2016
  Not all months occur

```
SELECT YEAR(OrderDate)*100 + Month(OrderDate) AS RevenueMonth, SUM(od.UnitPrice * od.Quantity) AS Revenue
FROM Orders o INNER JOIN OrderDetails od ON o.OrderID = od.OrderID
WHERE YEAR(OrderDate) = 2016
GROUP BY YEAR(OrderDate)*100 + Month(OrderDate)
```

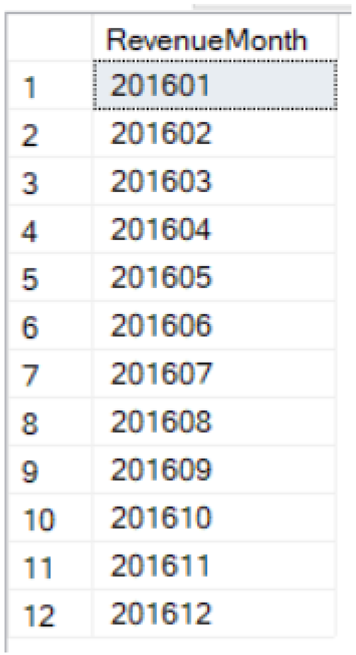


	RevenueMonth	Revenue
1	201607	30192,10
2	201608	26609,40
3	201609	27636,00
4	201610	41203,60
5	201611	49704,00
6	201612	50953,40

Application: generate missing months

- Solution: Generate all months with CTE

```sql
WITH Months AS
(SELECT 201601 as RevenueMonth
UNION ALL
SELECT RevenueMonth + 1
FROM Months
WHERE RevenueMonth < 201612)
SELECT * FROM Months;
```

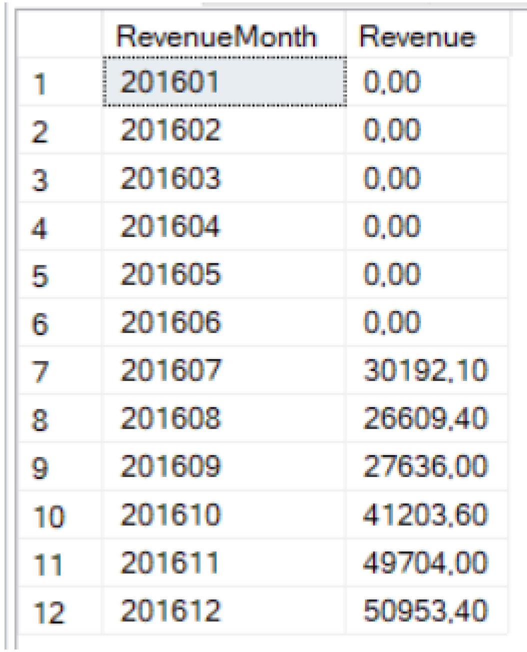


	RevenueMonth
1	201601
2	201602
3	201603
4	201604
5	201605
6	201606
7	201607
8	201608
9	201609
10	201610
11	201611
12	201612

Application: generate missing months

- Solution: … and combine with LEFT JOIN

```
WITH Months(RevenueMonth) AS
(SELECT 201601 as RevenueMonth
UNION ALL
SELECT RevenueMonth + 1
FROM Months
WHERE RevenueMonth < 201612),

Revenues(RevenueMonth, Revenue)
AS
(SELECT YEAR(OrderDate)*100 + Month(OrderDate) AS RevenueMonth,
SUM(od.UnitPrice * od.Quantity) AS Revenue
FROM Orders o INNER JOIN OrderDetails od ON o.OrderID = od.OrderID
WHERE YEAR(OrderDate) = 2016
GROUP BY YEAR(OrderDate)*100 + Month(OrderDate))

SELECT m.RevenueMonth, ISNULL(r.Revenue, 0) As Revenue
FROM Months m LEFT JOIN Revenues r ON m.RevenueMonth = r.RevenueMonth
```

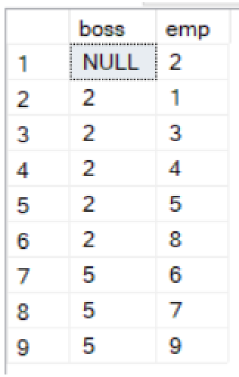


	RevenueMonth	Revenue
1	201601	0,00
2	201602	0,00
3	201603	0,00
4	201604	0,00
5	201605	0,00
6	201606	0,00
7	201607	30192,10
8	201608	26609,40
9	201609	27636,00
10	201610	41203,60
11	201611	49704,00
12	201612	50953,40

Recursively traversing a hierarchical structure

- Example: Give all employees who report directly or indirectly to Andrew Fuller (ReportsTo IS NULL)
  - Step 1 returns all employees that report directly to Andrew Fuller
  - Step 2 adds the second 'layer': who reports to someone who reports to A. Fuller

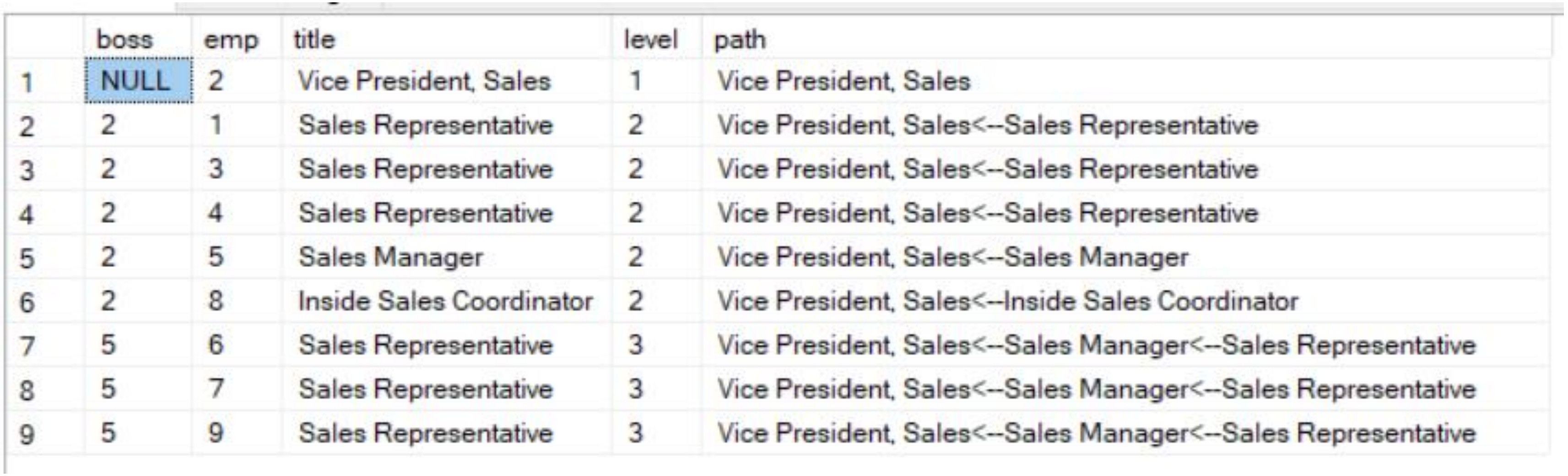
```
WITH Bosses (boss, emp)
AS
(SELECT ReportsTo, EmployeeID
FROM Employees
WHERE ReportsTo IS NULL
UNION ALL
SELECT e.ReportsTo, e.EmployeeID
FROM Employees e INNER JOIN Bosses b ON e.ReportsTo = b.emp)

SELECT * FROM Bosses
ORDER BY boss, emp;
```



	boss	emp
1	NULL	2
2	2	1
3	2	3
4	2	4
5	2	5
6	2	8
7	5	6
8	5	7
9	5	9

Recursively traversing a hierarchical structure

- Draw the organization chart.



	boss	emp	title	level	path
1	NULL	2	Vice President, Sales	1	Vice President, Sales
2	2	1	Sales Representative	2	Vice President, Sales<--Sales Representative
3	2	3	Sales Representative	2	Vice President, Sales<--Sales Representative
4	2	4	Sales Representative	2	Vice President, Sales<--Sales Representative
5	2	5	Sales Manager	2	Vice President, Sales<--Sales Manager
6	2	8	Inside Sales Coordinator	2	Vice President, Sales<--Inside Sales Coordinator
7	5	6	Sales Representative	3	Vice President, Sales<--Sales Manager<--Sales Representative
8	5	7	Sales Representative	3	Vice President, Sales<--Sales Manager<--Sales Representative
9	5	9	Sales Representative	3	Vice President, Sales<--Sales Manager<--Sales Representative

Recursively traversing a hierarchical structure

- Draw the organization chart.

```
WITH Bosses (boss, emp, title, level, path)
AS
(SELECT ReportsTo, EmployeeID, Title, 1, convert(varchar(max), Title)
FROM Employees
WHERE ReportsTo IS NULL
UNION ALL
SELECT e.ReportsTo, e.EmployeeID, e.Title, b.level + 1, convert(varchar(max), b.path + '<--' + e.title)
FROM Employees e INNER JOIN Bosses b ON e.ReportsTo = b.emp)

SELECT * FROM Bosses
ORDER BY boss, emp;
```

Exercises

- Give per region the number of orders (region USA + Canada = North America, rest = Rest of World). Use cte's.

```
-- Solution 1

SELECT
CASE c.Country
WHEN 'USA' THEN 'Northern America'
WHEN 'Canada' THEN 'Northern America'
ELSE 'Rest of world'
END AS Regionclass, COUNT(o.OrderID) As NumberOfOrders
FROM Customers c JOIN Orders o
ON c.CustomerID = o.CustomerID
GROUP BY
CASE c.Country
WHEN 'USA' then 'Northern America'
WHEN 'Canada' then 'Northern America'
ELSE 'Rest of world'
END
```

```
-- Solution 2 -> avoid copy-paste (subquery in FROM)

SELECT Regionclass, COUNT(OrderID)
FROM
(
SELECT
CASE c.Country
WHEN 'USA' THEN 'Northern America'
WHEN 'Canada' THEN 'Northern America'
ELSE 'Rest of world'
END AS Regionclass, o.OrderID
FROM Customers c JOIN Orders o
ON c.CustomerID = o.CustomerID
)
AS Totals(Regionclass, OrderID)
GROUP BY Regionclass
```

Exercises

```
-- 2 Make a histogram of the number of orders per
customer, so show how many times each number occurs.
-- E.g. in the graph below: 1 customer placed 1 order, 2
customers placed 2 orders, 7 customers placed 3 orders,
etc.

/*

nrNumberOfCustomers
11
22
37
46
510
68
77
...

*/
```

Exercises

```
-- 3. Give the customers of the Country in which most customers live
-- 4. Give all employees except for the eldest. Solve this first using a subquery and afterwards using a cte
-- 5. What is the total number of customers and suppliers?
-- 6. Give per title the eldest employee
-- 7. Give per title the employee that earns most
-- 8. Give the titles for which the eldest employee is also the employee who earns most
```

Exercises

```
-- 9. Execute the following script:
CREATE TABLE Parts
(
    [Super]   CHAR(3) NOT NULL,
    [Sub]     CHAR(3) NOT NULL,
    [Amount]  INT NOT NULL,
    PRIMARY KEY(Super, Sub)
);

INSERT INTO Parts VALUES ('O1','O2',10);
INSERT INTO Parts VALUES ('O1','O3',5);
INSERT INTO Parts VALUES ('O1','O4',10);
INSERT INTO Parts VALUES ('O2','O5',25);
INSERT INTO Parts VALUES ('O2','O6',5);
INSERT INTO Parts VALUES ('O3','O7',10);
INSERT INTO Parts VALUES ('O6','O8',15);
INSERT INTO Parts VALUES ('O8','O11',5);
INSERT INTO Parts VALUES ('O9','O10',20);
INSERT INTO Parts VALUES ('O10','O11',25);

-- Show all parts that are directly or indirectly part of O2,
-- so all parts of which O2 is composed.
-- Add an extra column with the path as shown:
```

```
/*
SUPER  SUB   PAD
O2     O5    O2 <-O5
O2     O6    O2 <-O6
O6     O8    O2 <-O6 <-O8
O8     O11   O2 <-O6 <-O8 <-O11
*/
```

Exercises – Solutions

- Give per region the number of orders (region USA + Canada = North America, rest = Rest of World). Use cte's.

```
-- Solution 3 -> with CTE

WITH OrdersPerRegion(RegionClass, OrderID)
AS (select
case c.country
when 'USA' then 'Northern America'
when 'Canada' then 'Northern America'
else 'Rest of world'
end as region, OrderID
from Customers c join Orders o
on c.CustomerID = o.CustomerID
)

SELECT RegionClass, COUNT(OrderID) As NumberOfOrders
FROM OrdersPerRegion
GROUP BY RegionClass
```

Exercises – Solutions

```sql
-- 2
with NrOfOrders(nr) as
(select count(*)
 from orders
 group by customerid)

   select nr, count(*) as NumberOfCustomers
   from NrOfOrders
   group by nr
   order by nr;

-- 3. Give the customers of the Country in which most customers live
WITH cte1(Country, NumberOfCustomers)
AS
(SELECT Country, COUNT(CustomerID) FROM Customers GROUP BY Country),

cte2(MaximumNumberOfCustomers)
AS
(SELECT MAX(NumberOfCustomers) FROM cte1)

SELECT CustomerID, CompanyName, c.Country
FROM Customers c JOIN cte1 ON c.country = cte1.country JOIN cte2 ON cte1.NumberOfCustomers =
cte2.MaximumNumberOfCustomers
```

Exercises – Solutions

```
-- 4. Give all employees except for the eldest
-- Solution 1 (using Subqueries)
SELECT employeeid, firstname + ' ' + lastname As employeeName, birthdate
FROM employees
WHERE birthdate > (SELECT MIN(birthdate) FROM employees)

-- Solution 2 (using CTE's)
WITH eldest(min_birthdate) AS
(SELECT min(birthdate)
FROM employees)

SELECT employeeid, firstname + ' ' + lastname As employeeName, birthdate
FROM employees CROSS JOIN eldest
WHERE birthdate > eldest.min_birthdate
```

Exercises – Solutions

```
-- 5. What is the total number of customers and suppliers?
WITH numberOfCustomers(nrOfCust) as (SELECT COUNT(CustomerID) FROM Customers),
numberOfSuppliers(nrOfSup) as (SELECT COUNT(SupplierID) FROM Suppliers)

SELECT((SELECT nrOfCust from numberOfCustomers) + (SELECT nrOfSup FROM numberOfSuppliers))

-- Other solution

WITH numberOfCustomers(nrOfCust) as (SELECT COUNT(CustomerID) FROM Customers),
numberOfSuppliers(nrOfSup) as (SELECT COUNT(SupplierID) FROM Suppliers)

SELECT nrOfCust + nrOfSup As 'Total number of customers and suppliers'
FROM numberOfCustomers CROSS JOIN numberOfSuppliers
```

Exercises – Solutions

```sql
-- 6. Give per title the eldest employee
WITH eldestPerTitle(title, min_birthdate) AS
(SELECT title, min(birthdate)
FROM employees
GROUP BY title)

SELECT employeeid, ept.title, ept.min_birthdate
FROM Employees e JOIN eldestPerTitle ept
ON e.title = ept.title
WHERE e.BirthDate = ept.min_birthdate
```

Exercises – Solutions

```sql
-- 7. Give per title the employee that earns most
WITH mostEarningPerTitle(title, max_salary) AS
(SELECT title, max(salary)
FROM employees
GROUP BY title)

SELECT employeeid, firstname + ' ' + lastname, mept.title, mept.max_salary
FROM Employees e JOIN mostEarningPerTitle mept
ON e.title = mept.title
WHERE e.salary = mept.max_salary
```

Exercises – Solutions

```
-- 8. Give the titles for which the eldest employee is also the employee who earns most
WITH eldestPerTitle(title, min_birthdate) AS
(SELECT title, min(birthdate)
FROM employees
GROUP BY title),

mostEarningPerTitle(title, max_salary) AS
(SELECT title, max(salary)
FROM employees
GROUP BY title)

SELECT employeeid, ept.title, ept.min_birthdate, mept.max_salary
FROM Employees e JOIN eldestPerTitle ept
ON e.title = ept.title
JOIN mostEarningPerTitle mept ON e.title = mept.title
WHERE e.BirthDate = ept.min_birthdate AND e.salary = mept.max_salary
```

Exercises – Solutions

```
-- 9
WITH Relation(Super, Sub, [Path]) AS
    (
        -- Default
        SELECT
         Super
        ,Sub
        ,[Path] =  CAST(CONCAT(Super, ' <- ',Sub) AS NVARCHAR(MAX)) -- Don't forget to CAST
        FROM Parts
        WHERE Super = 'O2'

        UNION ALL
        -- Recursion
        SELECT
         Parts.Super
        ,Parts.Sub
        ,[Path] = CONCAT(Relation.[Path], ' <- ',Parts.Sub)
        FROM Parts
            JOIN Relation ON Parts.Super = Relation.Sub
    )
    SELECT * FROM Relation;
```