SUBQUERIES

SUBQUERIES basic form

- Nested subqueries
  - Basic form

```
SELECT
FROM
WHERE condition
```

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 sub query:
  - Always executed first
  - Always between ().
  - Subqueries can be nested at > 1 level.

- A subquery can
  - return one value
  - return a list of values

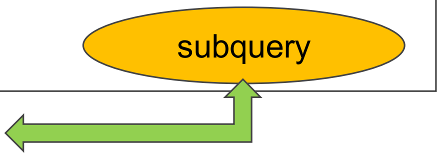
SUBQUERY that returns a single value

- The result of the query can be used anywhere you can use an expression.
  - With all relational operators: =, >, <, <=,>=,<>
  - Example:
    - What is the highest salary? (db xTreme)

```
select max(salary)
from employee
```

    - Who has the highest salary?

```
select lastname,firstname,salary
from employee
where salary =
(select max(salary) from employee)
```



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

SUBQUERY that returns a single value

- Other examples
  - Determine the salary of the employees that earn more than average

```
select lastname,firstname,salary
from employee
where salary >(select avg(salary) from employee)
```

  - -- Who is the youngest employee from Canada?

```
select lastname,firstname
from employee
where country='Canada'
and birthdate =(select max(birthdate) from employee where country='Canada');
```

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)
    - » DB Tennis: give all players that played matches (can also be accomplished with join)

```
SELECT playerno,name,initials
from players
where playerno in (select playerno from matches)
```

    - » Give name of the players who live in the same town as R. Permenter

```
select name

from players

where town= (select town from players where name='Parmenter' and initials='R');
```

    - » Why is the query below not working?

```
select name from players
where town=(select town from players where name ='Permenter');
```

SUBQUERY that returns a single column

- NOT IN / <>ALL operator
  - » Give all players that did not play any matches

```
select playerno
from players
where playerno not in (select playerno from matches)
```

  - » This can't be solved with INNER JOIN, only with OUTER JOIN and EXISTS (see below)

ANY and ALL keywords

- These keywords are used in combination with the relational operators and subqueries that return a column of values
  - **ALL** returns TRUE if all values returned in the subquery satisfy the condition
  - **ANY** returns TRUE if at least one value returned in the subquery satisfies the condition
  - Example: give the highest playerno and the corresponding leagueno.

```
SELECT PLAYERNO,LEAGUENO
FROM PLAYERS
WHERE PLAYERNO >= ALL (SELECT PLAYERNO FROM PLAYERS WHERE LEAGUENO IS NOT NULL);
```

  - Example: Give the playernos with at least one penalty that is larger than a penalty paid by player 27; player 27 himself should not appear in the result.

```
SELECT DISTINCT PLAYERNO
FROM PENALTIES
WHERE PLAYERNO <>27
AND AMOUNT > ANY (SELECT AMOUNT FROM PENALTIES WHERE PLAYERNO=27);
```

ANY and ALL keywords

```
SELECT PLAYERNO,LEAGUENO
FROM PLAYERS
WHERE PLAYERNO >= ALL (SELECT PLAYERNO FROM PLAYERS WHERE LEAGUENO IS NOT NULL);
```

Returns the same result as:

```
SELECT PLAYERNO,LEAGUENO
FROM PLAYERS
WHERE PLAYERNO = (SELECT MAX(PLAYERNO) FROM PLAYERS WHERE LEAGUENO IS NOT NULL);
```

```
SELECT DISTINCT PLAYERNO
FROM PENALTIES
WHERE PLAYERNO <>27
AND AMOUNT >(SELECT MIN(AMOUNT) FROM PENALTIES WHERE PLAYERNO=27);
```

Returns the same result as:

```
SELECT DISTINCT PLAYERNO
FROM PENALTIES
WHERE PLAYERNO <>27
AND AMOUNT > ANY (SELECT AMOUNT FROM PENALTIES WHERE PLAYERNO=27);
```

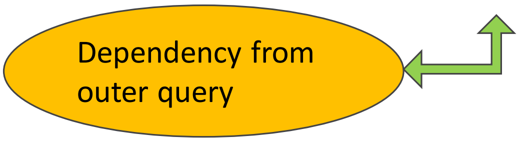
Correlated subqueries

- In a correlated subquery the inner query depends on information from the outer query.
  - – the contains a search condition that refers to the main query, which make the subquery depends on the main query
- The subquery is executed for each row in the main query.

  → O(n²)

  →The order of execution is from top to bottom, not from bottom to top as in a simple subquery, which is O(n).
- For performance reasons use joins or simple subquery is possible
- Principle

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



Dependency from outer query

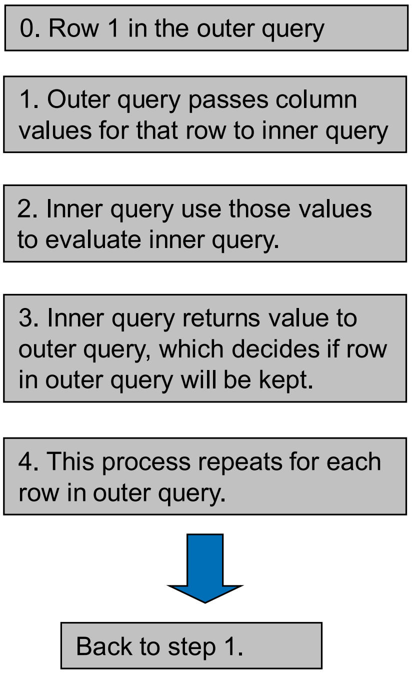
Correlated subqueries

- Example: give employees with a salary larger than the average salary (db Xtreme)

```
SELECT lastname, firstname, salary
FROM employee
WHERE salary >
(
    SELECT AVG(salary)
    FROM employee
);
```

- Give the employees whose salary is larger than the average of the salary of the employees **who report to the same boss**.

```
SELECT lastname, firstname, salary
FROM employee AS e
WHERE salary >
(
    SELECT AVG(salary)
    FROM employee
    WHERE reportsto = e.reportsto
);
```



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.

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

Subqueries and the EXISTS operator

- The operator EXISTS tests the existence of a result set.
- There is also NOT EXISTS
  - Example: give the players that did not play any matches yet.

```
SELECT *
FROM players AS p
WHERE NOT EXISTS
(
    SELECT * FROM matches
    WHERE playerno = p.playerno
);
```

- Give the players that did play matches

```
SELECT *
FROM players AS p
WHERE EXISTS
(
    SELECT * FROM matches
    WHERE playerno = p.playerno
);
```

3 ways to accomplish the same result

Who did not play any matches?

```
-- OUTER JOIN
SELECT p.playerno
FROM players AS p
     LEFT JOIN
     matches AS m
ON p.playerno = m.playerno
WHERE m.playerno IS NULL;
```

```
-- SIMPLE SUBQUERY
SELECT playerno FROM players
WHERE playerno NOT IN
(
 SELECT playerno FROM matches
);
```

```
-- CORRELATED SUBQUERY
SELECT playerno
FROM players AS p
WHERE NOT EXISTS
(
SELECT NULL FROM matches
WHERE playerno = p.playerno
);
```

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

```sql
-- Solution 1
select
case c.country
when 'USA' then 'Northern America'
when 'Canada' then 'Northern America'
else 'Rest of world'
end as regionclass, sum(orderamount)
from customer 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 ;
-- drawback: copy-paste of case
```

```sql
-- Solution 2
-- avoid copy-paste via subquery in FROM
select regionclass, sum(total) from
(
select
case c.country
when 'USA' then 'Northern America'
when 'Canada' then 'Northern America'
else 'Rest of world'
end as region, orderamount
from customer c join orders o
on c.CustomerID=o.CustomerID
)
as totals(regionclass,total)
group by regionclass;
```

Subqueries in the SELECT-clause

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

```sql
SELECT lastname, firstname, salary,
salary -
(
    SELECT AVG(salary)
    FROM employee
    WHERE supervisorid = e.supervisorid
)
FROM employee e;
```

Subqueries in the SELECT- and FROM-clause

- (db xtreme): give per productclass the price of the cheapest product and a product that has that price.

```
SELECT class, unitprice,
(
    SELECT TOP 1 productid
    FROM product
    WHERE productclassid = class AND
          price = unitprice
)
FROM
(
    SELECT productclassid, MIN(price)
    FROM product AS p
    GROUP BY productclassid
) AS pcmin(class, unitprice);
```

Application: running totals

Running total of orderamount per year:

```
SELECT orderid,orderdate,orderamount,
(select sum(orderamount)
from orders where year(orderdate) =
year(o.orderdate) and orderdate <=
o.orderdate)
FROM orders o
order by orderdate;
```

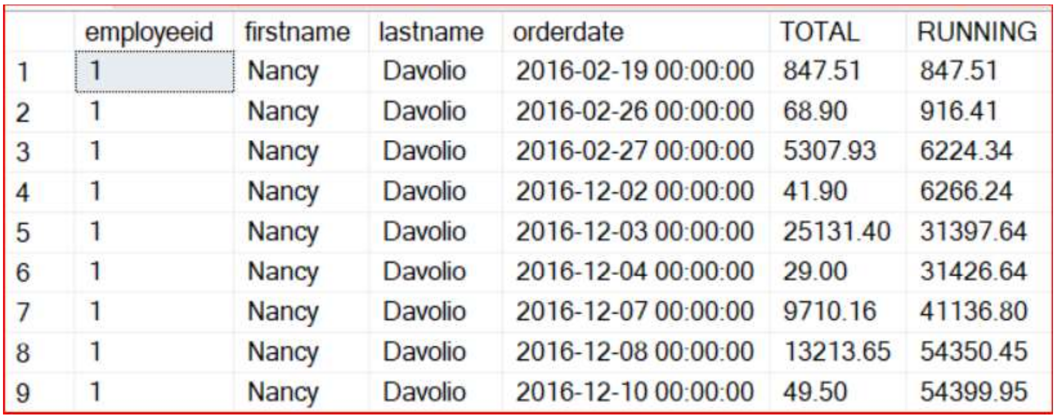
Application: monthly gross margin

```
SELECT isnull(ord.month, pur.month), isnull(ord.amount, 0) - isnull(pur.amount, 0) AS margin
FROM(
    (
        SELECT format(orderdate, 'yyyy-MM'), SUM(orderamount)
        FROM orders
        GROUP BY format(orderdate, 'yyyy-MM')
    ) AS ord(month, amount)
    FULL JOIN
    (
        SELECT format(orderdate, 'yyyy-MM'), SUM(p.price * pu.UnitsOnOrder)
        FROM purchases AS pu
             JOIN
             product AS p
             ON pu.PRODUCTID = p.PRODUCTID
        GROUP BY format(orderdate, 'yyyy-MM')
    ) AS pur(month, amount)
    ON ord.month = pur.month)
ORDER BY 1;
```

Some exercises

Database Xtreme:

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 sold (ordered) yet.
3. Select the products (all data) with a price that is higher than the average price of the "Bicycle" products. Select in descending order of price.
4. Show a list of the orderID's of the orders for which the order amount differs from the amount calculated through the ordersdetail.
5. Which employee has processed most orders?
6. Give per employee and per order date the total order amount. Also add the name of the employee and the running total per employee when ordering by orderdate:

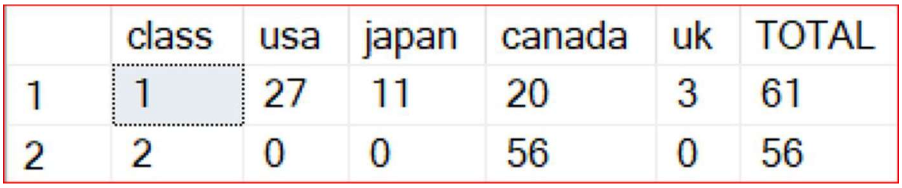


	employeeid	firstname	lastname	orderdate	TOTAL	RUNNING
1	1	Nancy	Davolio	2016-02-19 00:00:00	847.51	847.51
2	1	Nancy	Davolio	2016-02-26 00:00:00	68.90	916.41
3	1	Nancy	Davolio	2016-02-27 00:00:00	5307.93	6224.34
4	1	Nancy	Davolio	2016-12-02 00:00:00	41.90	6266.24
5	1	Nancy	Davolio	2016-12-03 00:00:00	25131.40	31397.64
6	1	Nancy	Davolio	2016-12-04 00:00:00	29.00	31426.64
7	1	Nancy	Davolio	2016-12-07 00:00:00	9710.16	41136.80
8	1	Nancy	Davolio	2016-12-08 00:00:00	13213.65	54350.45
9	1	Nancy	Davolio	2016-12-10 00:00:00	49.50	54399.95

Some exercises

Database Xtreme:

7. How many products of each class do we buy per supplier country ?
   Provide the results as a pivot table (you can hard code the country names). Also provide a TOTALs column.



	class	usa	japan	canada	uk	TOTAL
1	1	27	11	20	3	61
2	2	0	0	56	0	56

Database tennis:

8. Give the name and number of the players that already got more penalties than they played matches.

SQL - DML basic tasks

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

CHANGE DATA

INSERT: add new rows

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.

```
begin transaction  -- starts a new "transaction" --> Saves previous state of DB in buffer

-- several "destructive" commands can go here:
delete from Employee;
insert into product
values (10001, 'Drinking bottle', null, null, null, null, null, null, null, null, null);

-- only you (in  your session) can see changes
select * from Product where ProductID = 10001;

rollback;   --> ends transaction and restores database in previous state

-- commit;  --> ends transaction and makes changes permanent
```

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

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

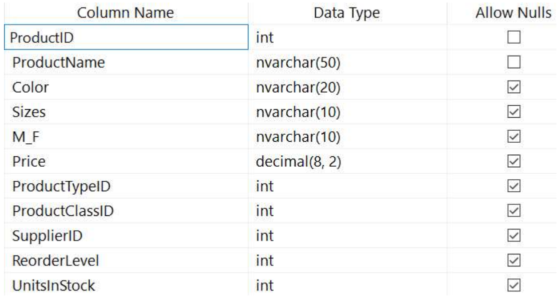
- Example: Add product “Energy bar” with category 1
  - method 1: specify only the (not NULL) values for specific columns

```
insert into product (ProductID,ProductName)
values (10000,'Energy bar')
```

  - method 2: specify all column values

```
insert into product
values (10001,'Drinking bottle',null,null,null,null,null,null,null,null,null)
```

INSERT of 1 row



Column Name	Data Type	Allow Nulls
ProductID	int	□
ProductName	nvarchar(50)	□
Color	nvarchar(20)	☑
Sizes	nvarchar(10)	☑
M_F	nvarchar(10)	☑
Price	decimal(8, 2)	☑
ProductTypeID	int	☑
ProductClassID	int	☑
SupplierID	int	☑
ReorderLevel	int	☑
UnitsInStock	int	☑

The number of specified columns corresponds to the number of values.

The specified values and corresponding 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.

INSERT of row(s) selected from other tables

- Examples: add all employees to the customer table

```
INSERT INTO customer

SELECT substring(firstname,1,3) + substring(lastname,1,3), lastname, firstname, title, address,
city, region, postalcode, country, homephone, null

FROM Employee
```

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.

CHANGE DATA

UPDATE: modify values

Changing data - UPDATE

- Changing all rows in a table
  - example: increase the price of all products with 10%

```
UPDATE Product
SET price = (price * 1.1)
```

- Changing 1 row or a group of rows
  - example: increase the price of the product “Wheeler” with 10%

```
UPDATE product
SET price = (price * 1.1)
WHERE productname = ‘Wheeler'
```

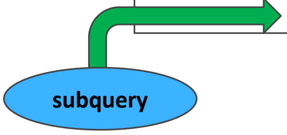
  - example: increase the price of the product “Wheeler” with 10% and set all units in stock to 0

```
UPDATE product
SET price = (price * 1.1), unitsinstock = 0
WHERE productname = ‘Wheeler'
```

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 must increase the unit price of products delivered by suppliers from the USA by 10%.

```
UPDATE product
SET price = (price * 1.1)
WHERE supplierid IN
      (SELECT supplierid FROM supplier WHERE country = 'USA')
```



CHANGE DATA

DELETE: remove rows

Removing data - DELETE

- Deleting rows
  - example: delete product Wheeler'

```
DELETE
FROM product
WHERE productname = Wheeler'
```

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

```
DELETE
FROM product
```

  - via TRUNCATE the identity value (see further) restarts from 1
  - TRUNCATE is also more performant, but does not offer where clause: all or nothing

```
TRUNCATE TABLE product
```

DELETE - based on data in another table

Example: delete the order details for all orders from the most recent order date.

→ Again no JOIN, only subquery

```
delete from ordersdetail
where orderid in
     (select orderid from orders
      where orderdate = (select MAX(orderdate) from Orders));
```

CHANGE DATA

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 rows because Excel offers a better overview than their ERP tool.
- They can update rows , add new ones and delete rows in Excel.
- After uploading the edited Excel file to a temporary table, the **MERGE** statement performs all UPDATEs, INSERTs and DELETEs at once.

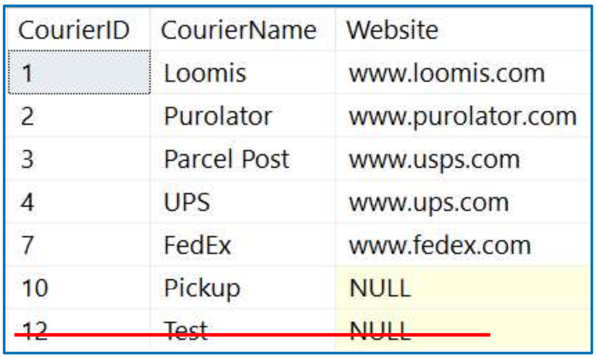
MERGE

- First execute following script to simulate the Excel file has been imported to a temporary table ‘courier_update’.

```
drop table if exists courier_update;

select * into courier_update from Courier;

insert into courier_update values (11,'BPost','www.bpost.be')

update courier_update set Website = 'www.pickup.com' where CourierID = 10

delete from courier_update where CourierID=12

select * from Courier;

select * from Courier_update;
```

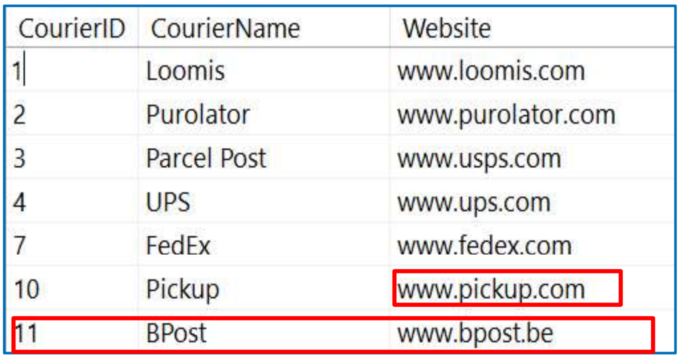
MERGE: example

Original table:
COURIER (DB Xtreme):



CourierID	CourierName	Website
1	Loomis	www.loomis.com
2	Purolator	www.purolator.com
3	Parcel Post	www.usps.com
4	UPS	www.ups.com
7	FedEx	www.fedex.com
10	Pickup	NULL
~~12~~	~~Test~~	~~NULL~~

Temporary table
COURIER_UPDATE



CourierID	CourierName	Website
1	Loomis	www.loomis.com
2	Purolator	www.purolator.com
3	Parcel Post	www.usps.com
4	UPS	www.ups.com
7	FedEx	www.fedex.com
10	Pickup	www.pickup.com
11	BPost	www.bpost.be

Remark: there is 1 deleted row, 1 added row and 1 updated row.

MERGE

Statement (courierid → Identity=No, s = source, t=target):

```
begin transaction
select * from courier
select * from courier_update
merge courier as t
using courier_update as s
on (t.courierid = s.courierid)

when matched and t.couriername <> s.couriername or isnull(t.website,'') <> isnull(s.website,'') -> rows to update
then update set t.couriername = s.couriername, t.website=s.website

when not matched by target -> new rows
then insert (courierid,couriername,website) values (s.courierid,s.couriername,s.website)

when not matched by source  -> rows to delete
then delete;

select * from courier
rollback
```

Remark: the option to delete rows is a non standard extension of MS SQL Server.

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;
- 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

Definition of a view

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

syntax of CREATE VIEW

- # nr 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

```
CREATE VIEW V_ProductsCustomer(productcode, customername, sumquantity)
AS SELECT productid, customername, sum(quantity)
   FROM customer
JOIN orders ON orders.customerid = customer.customerid
JOIN ordersdetail ON orders.orderid = ordersdetail.orderid
GROUP BY  productid,customername
```

example: creation of a view

```
SELECT * FROM V_ProductsCustomer
```

example: use of a view

```
ALTER VIEW V_ProductsCustomer(productcode, customername, avgquantity)
AS SELECT productid, customername, avg(quantity)
   FROM customer
JOIN orders ON orders.customerid = customer.customerid
JOIN ordersdetail ON orders.orderid = ordersdetail.orderid
GROUP BY  productid,customername
```

example: changing a view

```
DROP VIEW V_ProductsCustomer
```

example: deleting a view

Example: views as partial solution for complex problems

Gross margin problem with views instead of subqueries:

```
create view sales(month, total) as
      select format(orderdate,'yyyy-MM'), sum(orderamount)
      from orders
      group by format(orderdate,'yyyy-MM');

create view pur(month, total) as
      select format(orderdate,'yyyy-MM'), sum(p.price*pu.UnitsOnOrder)
      from purchases pu join product p on pu.PRODUCTID = p.PRODUCTID
      group by format(orderdate,'yyyy-MM');

select isnull(s.month, p.month) MONTH,
        isnull(s.total,0) - isnull(p.total,0) MARGIN
from sales s full join pur p on s.month=p.month
order by 1;
```

Drawback of using views in this case:
views are stored in the database and might create a mess if you have hundreds of them.

Update of views

- An updatable view
  - Has no distinct of 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

- Rule of thumb: in general views are updatable if the system is able to translate the updates to individual rows 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 may 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.

Views with/without check option: example

```
CREATE VIEW productsOfType6
AS SELECT * FROM product WHERE ProductTypeID = 6
```

example: create view without "with check option"

```
INSERT INTO productsOfType6 (productid,productname, producttypeid)
VALUES (10000,'Wheeler', 1)
```

example: insert product from producttype 1

Although 'Wheeler' does not belong to producttype 6, it can be added through the view

```
CREATE VIEW productsOfType6Bis
AS SELECT * FROM product WHERE ProductTypeID = 6
WITH CHECK OPTION
```

example: insert statement above generates error message

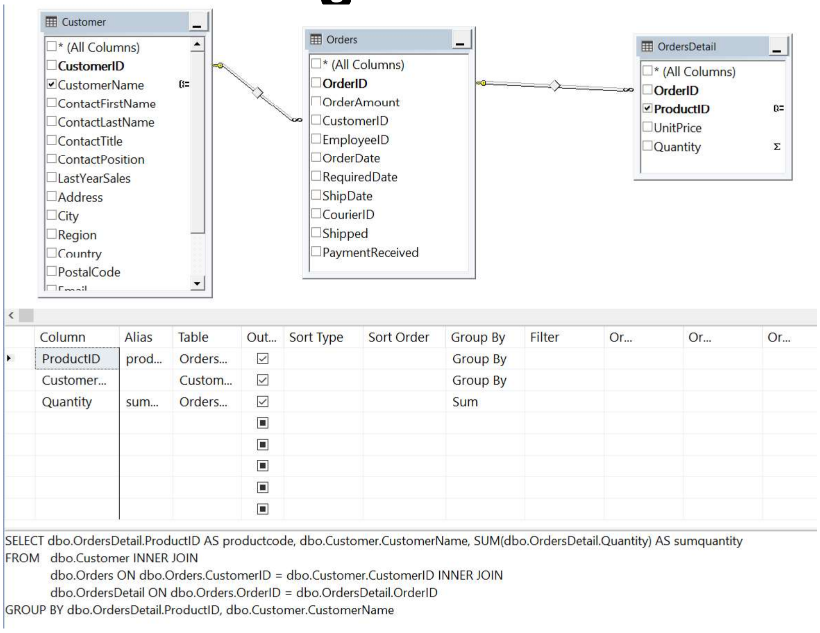
```
Msg 550, Level 16, State 1, Line 13
The attempted insert or update failed because the target view either specifies WITH CHECK OPTION
or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from the operation
did not qualify under the CHECK OPTION constraint.
The statement has been terminated.
```

Views in SQL Server Management Studio

Simple views can also be made with the graphical user interface

Example: `V_ProductsCustomer`

Not possible for views with subqueries or common table expressions



Column	Alias	Table	Out...	Sort Type	Sort Order	Group By	Filter	Or...	Or...	Or...
ProductID	prod...	Orders...	☑			Group By				
Customer...		Custom...	☑			Group By				
Quantity	sum...	Orders...	☑			Sum				
			■							
			■							
			■							
			■							
			■							

```
SELECT dbo.OrdersDetail.ProductID AS productcode, dbo.Customer.CustomerName, SUM(dbo.OrdersDetail.Quantity) AS sumquantity
FROM   dbo.Customer INNER JOIN
         dbo.Orders ON dbo.Orders.CustomerID = dbo.Customer.CustomerID INNER JOIN
         dbo.OrdersDetail ON dbo.Orders.OrderID = dbo.OrdersDetail.OrderID
GROUP BY dbo.OrdersDetail.ProductID, dbo.Customer.CustomerName
```

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

SQL Advanced

COMMON TABLE EXPRESSIONS

SQL Advanced - CTE

Common Table Expressions: the WITH component

The WITH-component has two application areas:

1. Simplify SQL-instructions, ex. simplified alternative for simple subqueries or avoid repetition of SQL constructs
2. Traverse recursively hierarchical and network structures

Example: give the average number of penalties of all players? This can be solved using a subquery (SELECT AVG(COUNT)) is not possible):

```sql
SELECT AVG(number * 1.0) -- *1.0 to force floating point
FROM
(
    SELECT COUNT(pe.playerno)
    FROM players AS pl
         LEFT JOIN penalties AS pe
         ON pl.PLAYERNO = pe.PLAYERNO
    GROUP BY pl.PLAYERNO
) AS fines(number);
```

Common Table Expressions: the WITH component

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):

```
WITH fines(number)
     AS (SELECT COUNT(pe.playerno)
         FROM players AS pl
              LEFT JOIN penalties AS pe
              ON pl.PLAYERNO = pe.PLAYERNO
         GROUP BY pl.PLAYERNO)

SELECT AVG(number * 1.0)
     FROM fines;
```

An expression like this is called: **common table expression**, shortened as **CTE**

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 avoid repetition of subqueries

Example: give the payment numbers and penalty amount that are not equal to the highest and lowest penalty ever paid by player 44.

Also show this highest and lowest amount in the result.

Without CTE:

```
SELECT paymentno, amount,

(SELECT MIN(amount) FROM penalties WHERE playerno = 44),
(SELECT MAX(amount) FROM penalties WHERE playerno = 44)

FROM penalties
WHERE amount <> (SELECT MIN(amount) FROM penalties WHERE playerno = 44)
  AND amount <> (SELECT MAX(amount) FROM penalties WHERE playerno = 44);
```

CTEs to avoid repetition of subqueries

Example: give the payment numbers and penalty amount that are not equal to the highest and lowest penalty ever paid by player 44.

Also show this highest and lowest amount in the result.

With CTE:

```
with min_max(min_amount, max_amount) as
(select min(amount), max(amount)
 from penalties
 where playerno=44)

select p.paymentno, p.amount, mm.min_amount, mm.max_amount
from penalties p cross join min_max mm
where p.amount <> mm.max_amount and p.amount <> mm.min_amount;
```

CTE’s to avoid repetition of subqueries

Example: generate the numbers 0 to 999

```
with numbers(number) as
(select 0 as number union
 select 1 union
 select 2 union
 select 3 union
 select 4 union
 select 5 union
 select 6 union
 select 7 union
 select 8 union
 select 9)

select (number1.number * 100) + (number2.number * 10) + number3.number
as number
from numbers as number1  cross join numbers as number2
                         cross join numbers as number3
order by number;
```

CTE’s to simplify queries

- (DB Xtreme): give per product class the price of the cheapest product and all products with that price.
- Using a subquery:

```sql
SELECT class, unitprice,
(
    SELECT TOP 1 productid
    FROM product
    WHERE productclassid = class AND price = unitprice
)
FROM
(
    SELECT productclassid, MIN(price)
    FROM product AS p
    GROUP BY productclassid
) AS pcmin(class, unitprice);
```

Disadvantage: top 1 is necessary in case several products have that price.
As a consequence, only one product per class can be shown .

CTE’s to simplify queries

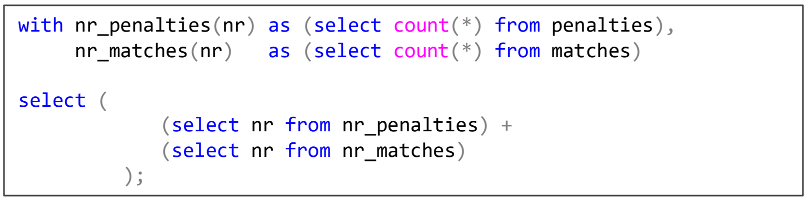
- (DB Xtreme): give per product class the price of the cheapest product and all products with that price.
- Solution: with CTE:

```sql
WITH pcmin(class, unitprice)
     AS (SELECT productclassid, MIN(price)
         FROM product AS p
         GROUP BY productclassid)

SELECT class, unitprice, productid
FROM product AS p
JOIN pcmin AS pc ON p.ProductClassID = pc.class
WHERE p.price = pc.unitprice;
```

Now we get all products with that price.

CTE’s with > 1 WITH-component

Example: what is the total number of rows in both the penalties and the matches table (DB Tennis)?



```
with nr_penalties(nr) as (select count(*) from penalties),
     nr_matches(nr)   as (select count(*) from matches)

select (
            (select nr from nr_penalties) +
            (select nr from nr_matches)
         );
```

Recursive SELECTs

- 'Recursive' 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 SELECTs

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;
```

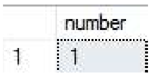
Characteristics of recursive use of WITH:

- 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.

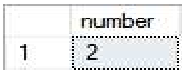
```
select 1
```



	number
1	1

2. Execute all recursive expressoins. The numbers table, that got a value of 1 in step 1, is used.

```
select number + 1
from numbers
where number < 5
```

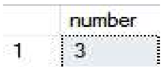


	number
1	2

This row is added to the numbers table.

Recursive SELECTs: how does it work?

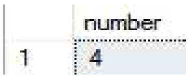
3. Now the recursion starts: the 2nd expression is re-executed, giving as result:



	number
1	3

Remark: not all rows added in all previous steps are processed, but only those rows (1 row in this example), that were added in the previous step (step 2).

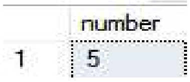
4. Since step 3 also gave a result, the recursive expression is executed again, producing as intermediate result:



	number
1	4

Recursive SELECTs: how does it work?

5. And this happens again:



	number
1	5

6. 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 SELECTs :
max number of recursions = 100

Example: give the numbers from 1 to 999 (cf. CTE without recursion)

```
with numbers(number) as
      (select 1
       union all
       select number + 1
       from numbers
       where number < 999)

select * from numbers;
```

The maximum recursion 100 has been exhausted before statement completion.

Recursive SELECTs:
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);
```

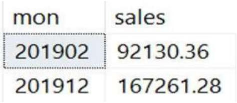
Maxrecursion is MS SQL Server specific.

Application: generate missing months

DB Xtreme: sales per month in year 2019:

```
select year(orderdate)*100 + month(orderdate) mon, sum(orderamount) as sales
from orders o
where year(orderdate) = 2019
group by year(orderdate)*100 + month(orderdate);
```

Problem: not all months occur:

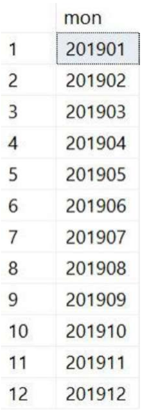


mon	sales
201902	92130.36
201912	167261.28

Application: generate missing months

Solution : generate all months with CTE…

```
with months as
       (select 201901 as mon
        union all
        select mon+1
        from months
        where mon < 201912)

select * from months;
```

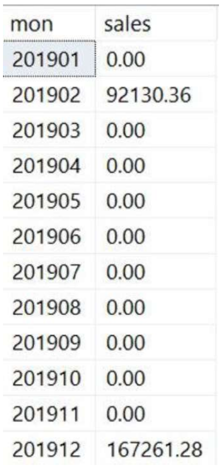


	mon
1	201901
2	201902
3	201903
4	201904
5	201905
6	201906
7	201907
8	201908
9	201909
10	201910
11	201911
12	201912

Application: generate missing months

Solution: … and combine with outer join

```
with months(mon) as
      (select 201901
       union all
       select mon + 1 from months
       where mon < 201912),

ord(mon,amount) as
(select year(orderdate)*100 + month(orderdate),
sum(orderamount)
from orders o
where year(orderdate) = 2019
group by year(orderdate)*100 + month(orderdate))

select m.mon, isnull(amount,0) sales
from months m
left join ord o on m.mon=o.mon
```

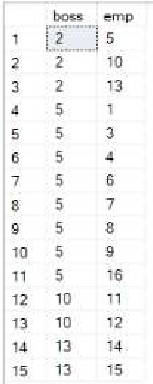


mon	sales
201901	0.00
201902	92130.36
201903	0.00
201904	0.00
201905	0.00
201906	0.00
201907	0.00
201908	0.00
201909	0.00
201910	0.00
201911	0.00
201912	167261.28

Recursively traversing a hierarchical structure

DB Xtreme: give all employees who report directly or indirectly to Andrew Fuller (employeeid=2)

```
with bosses (boss, emp)
as
      (select supervisorid, employeeid
       from employee
       where supervisorid = 2
      union all
       select e.supervisorid, e.employeeid
       from employee e join bosses b on e.supervisorid = b.emp)

select * from bosses
order by boss, emp;
```



	boss	emp
1	2	5
2	2	10
3	2	13
4	5	1
5	5	3
6	5	4
7	5	6
8	5	7
9	5	8
10	5	9
11	5	16
12	10	11
13	10	12
14	13	14
15	13	15

Recursively traversing a hierarchical structure

DB Xtreme: give all employees who report directly or indirectly to Andrew Fuller (employeeid=2)

- the 1st step returns all employees that reports directly to Andrew Fuller
- Step 2 adds the 2de "layer" : who reports to someone who reports to A. Fuller
- Etc.

Recursively traversing a hierarchical structure

DB Xtreme: give the complete hierarchy of the company, including the names of the employees.
Draw the organization chart.

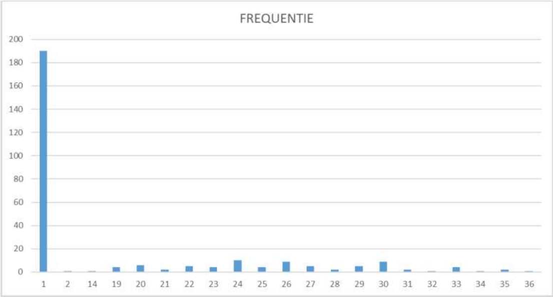
```sql
with bosses (symbol, boss, emp,name, path)
as
      (select convert(varchar(max),'-----'), supervisorid, employeeid,
       lastname + ' ' + firstname, convert(varchar(max), employeeid)
       from employee
       where supervisorid is null
      union all
       select symbol + '-----', e.supervisorid, e.employeeid, lastname
       + ' ‘ +     firstname, b.path + '.' + convert(varchar(max), e.employeeid)
       from employee e
       join bosses b on e.supervisorid = b.emp)

select * from bosses
order by path;
```

Some exercises

Database Xtreme:

1. Rewrite the “monthly gross margin” example from the subqueries chapter, using common table expressions.

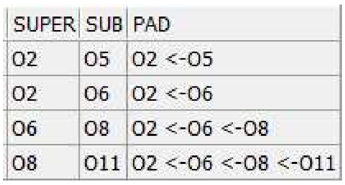
2. Make a histogram of the number of orders per customer, so show how many times each number occurs.
   E.g. in the graph below: 190 customers placed 1 order, 1 customer placed 2 orders, 1 customer placed 14 orders, etc.



Some exercises

Execute the script parts.sql in database Xtreme:

3. 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 below:



SUPER	SUB	PAD
O2	O5	O2 <-O5
O2	O6	O2 <-O6
O6	O8	O2 <-O6 <-O8
O8	O11	O2 <-O6 <-O8 <-O11

Database Tennis:

4. Delete all matches with at least three won sets.

5. Set all won sets to 0 for all players living in Stratford.

WINDOW FUNCTIONS

Window functions: business case

- Often business managers want to compare current sales to previous sales
- Previous sales can be:
  - sales during previous month
  - average sales during last three months
  - last year’s sales until current date (year-to-date)
- Window functions offer a solution to these kind of problems in a single, efficient SQL query
- Introduced in SQL: 2003

OVER clause

- Results of a SELECT are partitioned
- Numbering, ordering and aggregate functions per partition
- The OVER clauses creates partitions and ordering
- The partition behaves as a window that shifts over the data
- The OVER clause can be used with standard aggregate functions (sum, avg, …) or specific window functions (rank, lag,…)

Example: running total

- db xtreme: give orderid, orderdate, orderamount and running total (YTD) of the orderamount.
  Initialize the total for each new year.
- Using a correlated subquery this is very inefficient as for each line the complete sum is recalculated (see chapter about subqueries).

```
SELECT orderid, orderdate, orderamount,
    (select sum(orderamount)
     from orders
     where year(orderdate) = year(o.orderdate)
           and orderid <= o.orderid) YTD
FROM orders o
order by orderid;
```

Example: running total (II)

- The over clause makes the query
  - much simpler
  - far more efficient
- The sum is repeated for each partition
- YTD = year to date

```
SELECT orderid, orderdate, orderamount,
        sum(orderamount) over
        (partition by year(o.orderdate) order by o.orderid) YTD
FROM orders o
order by orderid;
```

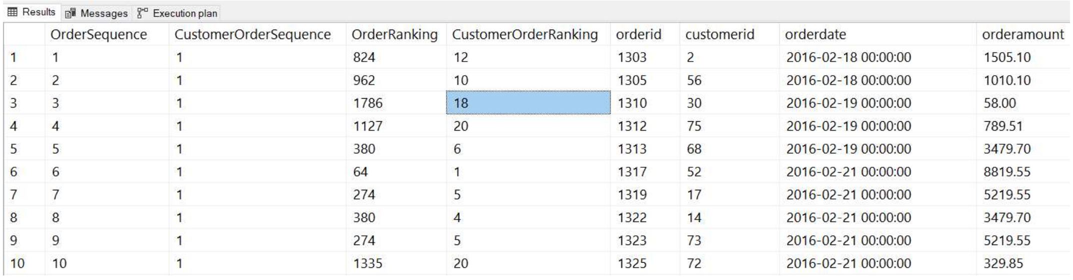
Window functions: row_number(), rank()

- Partition is optional, order by is mandatory
- row_number(): running sequence number, no duplicates occur in same partition
- rank(): running "rank" in partition, duplicates can occur: 1, 2, 3, 3, **5**
- dense_rank(): no gaps in ranking → 1, 2, 3, 3, **4**

```
select
row_number() over (order by o.orderdate, o.orderid) as OrderSequence,
row_number() over (partition by o.customerid order by o.orderdate, o.orderid) as CustomerOrderSequence,
rank() over (order by o.orderamount desc) as OrderRanking,
rank() over (partition by o.customerid order by o.orderamount desc) as CustomerOrderRanking,
o.orderid, o.customerid, o.orderdate, o.orderamount
from orders o
order by o.orderdate, orderid;
```

Window functions: row_number(), rank() (II)

- Result of previous query:



	OrderSequence	CustomerOrderSequence	OrderRanking	CustomerOrderRanking	orderid	customerid	orderdate	orderamount
1	1	1	824	12	1303	2	2016-02-18 00:00:00	1505.10
2	2	1	962	10	1305	56	2016-02-18 00:00:00	1010.10
3	3	1	1786	18	1310	30	2016-02-19 00:00:00	58.00
4	4	1	1127	20	1312	75	2016-02-19 00:00:00	789.51
5	5	1	380	6	1313	68	2016-02-19 00:00:00	3479.70
6	6	1	64	1	1317	52	2016-02-21 00:00:00	8819.55
7	7	1	274	5	1319	17	2016-02-21 00:00:00	5219.55
8	8	1	380	4	1322	14	2016-02-21 00:00:00	3479.70
9	9	1	274	5	1323	73	2016-02-21 00:00:00	5219.55
10	10	1	1335	20	1325	72	2016-02-21 00:00:00	329.85

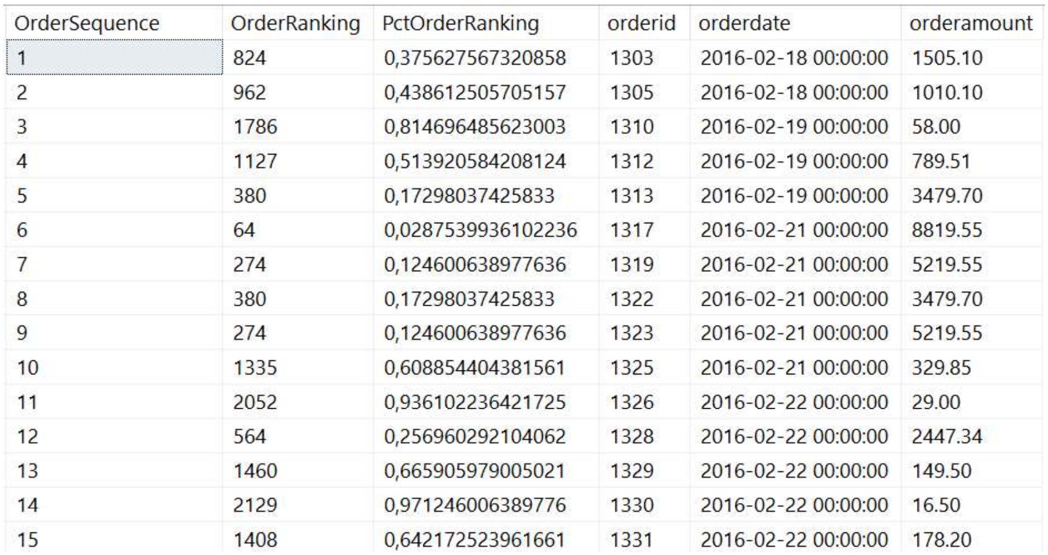
- CustomerOrderRanking = 18 means:
  - The current order is the 18th biggest order for the current customer (customerid = 30)

Window functions: percent_rank()

- percent_rank() shows the ranking on a scale from 0 - 1

```
select
row_number() over (order by o.orderdate, o.orderid) as OrderSequence,
rank() over (order by o.orderamount desc) as OrderRanking,
percent_rank() over (order by o.orderamount desc) as PctOrderRanking,
o.orderid, o.orderdate, o.orderamount
from orders o
order by o.orderdate, orderid;
```

percent_rank(): result of previous query



OrderSequence	OrderRanking	PctOrderRanking	orderid	orderdate	orderamount
1	824	0,375627567320858	1303	2016-02-18 00:00:00	1505.10
2	962	0,438612505705157	1305	2016-02-18 00:00:00	1010.10
3	1786	0,814696485623003	1310	2016-02-19 00:00:00	58.00
4	1127	0,513920584208124	1312	2016-02-19 00:00:00	789.51
5	380	0,17298037425833	1313	2016-02-19 00:00:00	3479.70
6	64	0,0287539936102236	1317	2016-02-21 00:00:00	8819.55
7	274	0,124600638977636	1319	2016-02-21 00:00:00	5219.55
8	380	0,17298037425833	1322	2016-02-21 00:00:00	3479.70
9	274	0,124600638977636	1323	2016-02-21 00:00:00	5219.55
10	1335	0,608854404381561	1325	2016-02-21 00:00:00	329.85
11	2052	0,936102236421725	1326	2016-02-22 00:00:00	29.00
12	564	0,256960292104062	1328	2016-02-22 00:00:00	2447.34
13	1460	0,665905979005021	1329	2016-02-22 00:00:00	149.50
14	2129	0,971246006389776	1330	2016-02-22 00:00:00	16.50
15	1408	0,642172523961661	1331	2016-02-22 00:00:00	178.20

Window functions: moving aggregate (1/7)

- Real meaning of window functions:
  apply to a window that shifts over the result set
- Previous examples work with default window: start of resultset to current row
- *Query 'Running total'* could also have been written as:

```
select orderid, orderdate, orderamount,
    sum(orderamount) over
    (partition by year(o.orderdate) order by o.orderid
    range between unbounded preceding and current row) YTD
from orders o
order by orderid;
```

Window functions: moving aggregate (2/7)

- With range you have three valid options:

```
• range between unbounded preceding and current row
• range between current row and unbounded following
• range between unbounded preceding and unbounded following
```

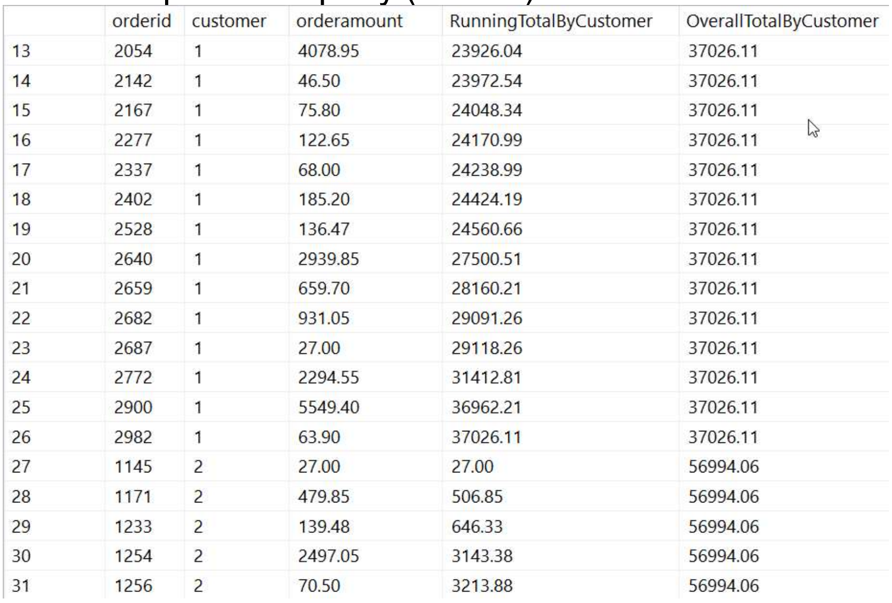
Window functions: moving aggregate (3/7)

Example: show running total and overall total by customer

```
select o.orderid, o.customerid, o.orderamount,
sum(o.orderamount) over (partition by o.customerid order by o.orderid,o.customerid
range between unbounded preceding and current row) as RunningTotalByCustomer, -- running total
sum(o.orderamount) over (partition by o.customerid order by o.orderid   -- order by is mandatory
range between unbounded preceding and unbounded following) as OverallTotalByCustomer
from orders o
order by o.customerid;
```

Window functions: moving aggregate (4/7)

- Result of previous query (extract)



	orderid	customer	orderamount	RunningTotalByCustomer	OverallTotalByCustomer
13	2054	1	4078.95	23926.04	37026.11
14	2142	1	46.50	23972.54	37026.11
15	2167	1	75.80	24048.34	37026.11
16	2277	1	122.65	24170.99	37026.11
17	2337	1	68.00	24238.99	37026.11
18	2402	1	185.20	24424.19	37026.11
19	2528	1	136.47	24560.66	37026.11
20	2640	1	2939.85	27500.51	37026.11
21	2659	1	659.70	28160.21	37026.11
22	2682	1	931.05	29091.26	37026.11
23	2687	1	27.00	29118.26	37026.11
24	2772	1	2294.55	31412.81	37026.11
25	2900	1	5549.40	36962.21	37026.11
26	2982	1	63.90	37026.11	37026.11
27	1145	2	27.00	27.00	56994.06
28	1171	2	479.85	506.85	56994.06
29	1233	2	139.48	646.33	56994.06
30	1254	2	2497.05	3143.38	56994.06
31	1256	2	70.50	3213.88	56994.06

Window functions: moving aggregate (5/7)

- When you use RANGE, the current row is compared to other rows and grouped based on the ORDER BY predicate.
- This is not always desirable; you might actually want a physical offset.
- In this scenario, you would specify ROWS instead of RANGE.
  This gives you three options in addition to the three options enumerated previously:

  - `rows between N preceding and current row`
  - `rows between current row and N following`
  - `rows between N preceding and N following`

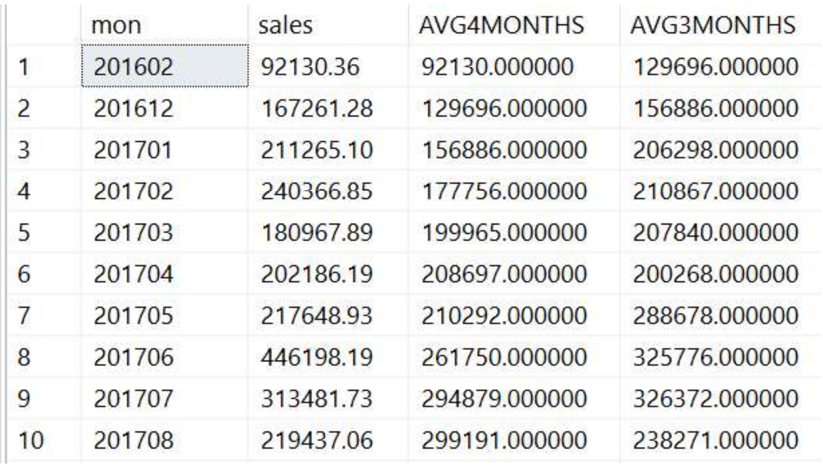
Window functions: moving aggregate (6/7)

- Example: show moving average of monthly sales for
  1. three preceding months and current month
  2. preceding, current and next month
- We use a CTE to calculate the monthly sales

```
with monthlysales as
(select year(orderdate)*100 + month(orderdate) MON, sum(o.orderamount) SALES
from Orders o
group by year(orderdate)*100 + month(orderdate))

select mon, sales,
round(avg(sales) over (order by mon rows between 3 preceding and current row),0) AVG4MONTHS,
round(avg(sales) over (order by mon rows between 1 preceding and 1 following),0) AVG3MONTHS
from monthlysales
order by 1;
```

Window functions: moving aggregate (7/7)

- Result of previous query (extract)



	mon	sales	AVG4MONTHS	AVG3MONTHS
1	201602	92130.36	92130.000000	129696.000000
2	201612	167261.28	129696.000000	156886.000000
3	201701	211265.10	156886.000000	206298.000000
4	201702	240366.85	177756.000000	210867.000000
5	201703	180967.89	199965.000000	207840.000000
6	201704	202186.19	208697.000000	200268.000000
7	201705	217648.93	210292.000000	288678.000000
8	201706	446198.19	261750.000000	325776.000000
9	201707	313481.73	294879.000000	326372.000000
10	201708	219437.06	299191.000000	238271.000000

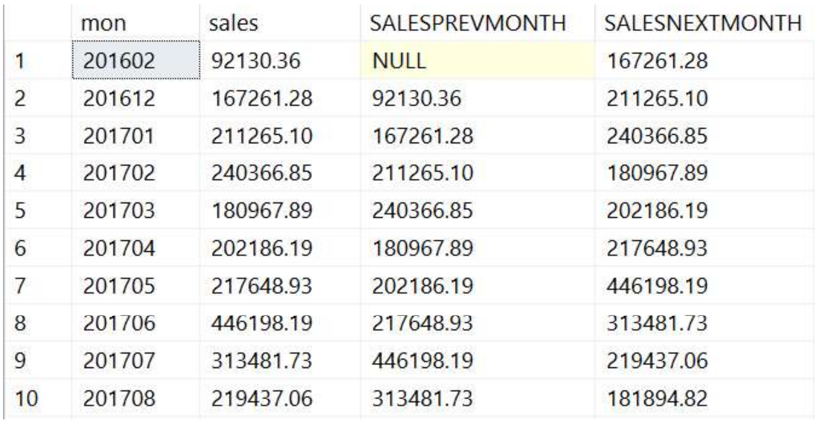
Window functions: LAG and LEAD (1/2)

- Windows functions LAG and LEAD refer to previous and next line respectively
- Example: show monthly sales for previous and next month

```
with monthlysales as
(select year(orderdate)*100 + month(orderdate) MON, sum(o.orderamount) SALES
from orders o
group by year(orderdate)*100 + month(orderdate))

select mon, sales,
lag(sales) over (order by mon) SALESPREVMONTH,
lead(sales) over (order by mon) SALESNEXTMONTH
from monthlysales
order by 1;
```

Window functions: LAG and LEAD (2/2)

- Result of previous query (extract)



	mon	sales	SALESPREVMONTH	SALESNEXTMONTH
1	201602	92130.36	NULL	167261.28
2	201612	167261.28	92130.36	211265.10
3	201701	211265.10	167261.28	240366.85
4	201702	240366.85	211265.10	180967.89
5	201703	180967.89	240366.85	202186.19
6	201704	202186.19	180967.89	217648.93
7	201705	217648.93	202186.19	446198.19
8	201706	446198.19	217648.93	313481.73
9	201707	313481.73	446198.19	219437.06
10	201708	219437.06	313481.73	181894.82

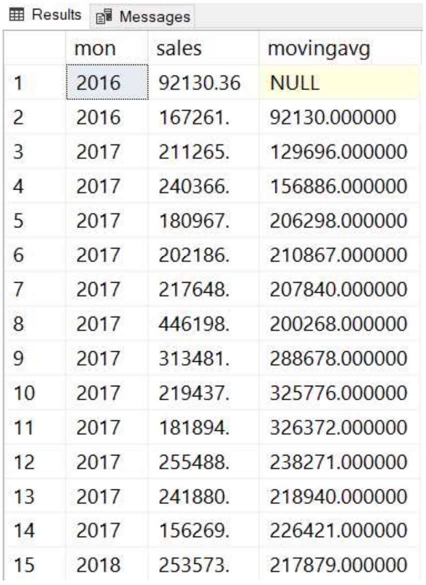
Exercises

Db xtreme

1. Compare the monthly sales to the moving average of the last three months. Show month, sales and moving average, see next slide.
2. Show for each month the percentual growth (or decline) as opposed to the previous month.
   Show month, sales and growth-%, see next slide.

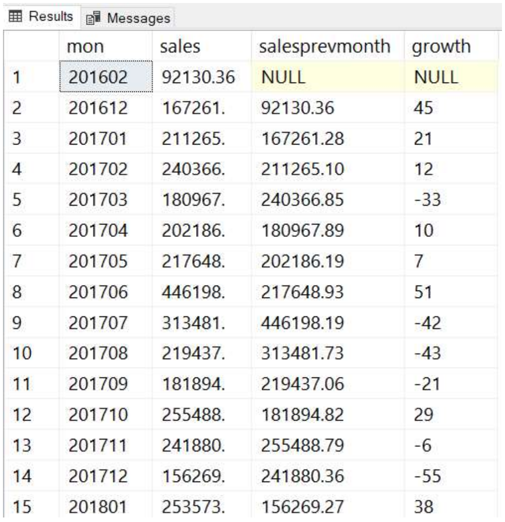
Exercises

Ex. 1: Sample resultset



	mon	sales	movingavg
1	2016	92130.36	NULL
2	2016	167261.	92130.000000
3	2017	211265.	129696.000000
4	2017	240366.	156886.000000
5	2017	180967.	206298.000000
6	2017	202186.	210867.000000
7	2017	217648.	207840.000000
8	2017	446198.	200268.000000
9	2017	313481.	288678.000000
10	2017	219437.	325776.000000
11	2017	181894.	326372.000000
12	2017	255488.	238271.000000
13	2017	241880.	218940.000000
14	2017	156269.	226421.000000
15	2018	253573.	217879.000000

Ex. 2 : Sample resultset



	mon	sales	salesprevmonth	growth
1	201602	92130.36	NULL	NULL
2	201612	167261.	92130.36	45
3	201701	211265.	167261.28	21
4	201702	240366.	211265.10	12
5	201703	180967.	240366.85	-33
6	201704	202186.	180967.89	10
7	201705	217648.	202186.19	7
8	201706	446198.	217648.93	51
9	201707	313481.	446198.19	-42
10	201708	219437.	313481.73	-43
11	201709	181894.	219437.06	-21
12	201710	255488.	181894.82	29
13	201711	241880.	255488.79	-6
14	201712	156269.	241880.36	-55
15	201801	253573.	156269.27	38

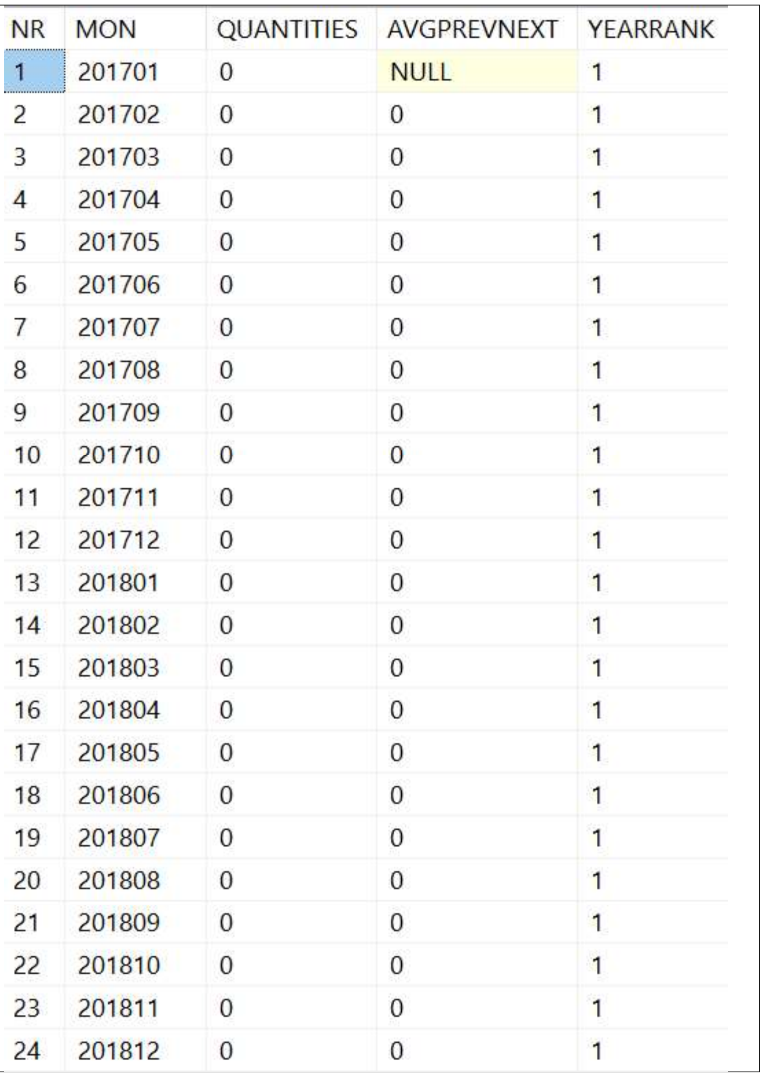
Exercises

Db xtreme

3. Show for each month (january-december) of the years 2017-2019 the total sold quantities and the average of the sold quantities in the previous and the next month.
   Also add a row number and show the rank (based on sold quantities) of each month in the current year.

Exercises

Ex. 3: sample resultset:



NR	MON	QUANTITIES	AVGPREVNEXT	YEARRANK
1	201701	0	NULL	1
2	201702	0	0	1
3	201703	0	0	1
4	201704	0	0	1
5	201705	0	0	1
6	201706	0	0	1
7	201707	0	0	1
8	201708	0	0	1
9	201709	0	0	1
10	201710	0	0	1
11	201711	0	0	1
12	201712	0	0	1
13	201801	0	0	1
14	201802	0	0	1
15	201803	0	0	1
16	201804	0	0	1
17	201805	0	0	1
18	201806	0	0	1
19	201807	0	0	1
20	201808	0	0	1
21	201809	0	0	1
22	201810	0	0	1
23	201811	0	0	1
24	201812	0	0	1