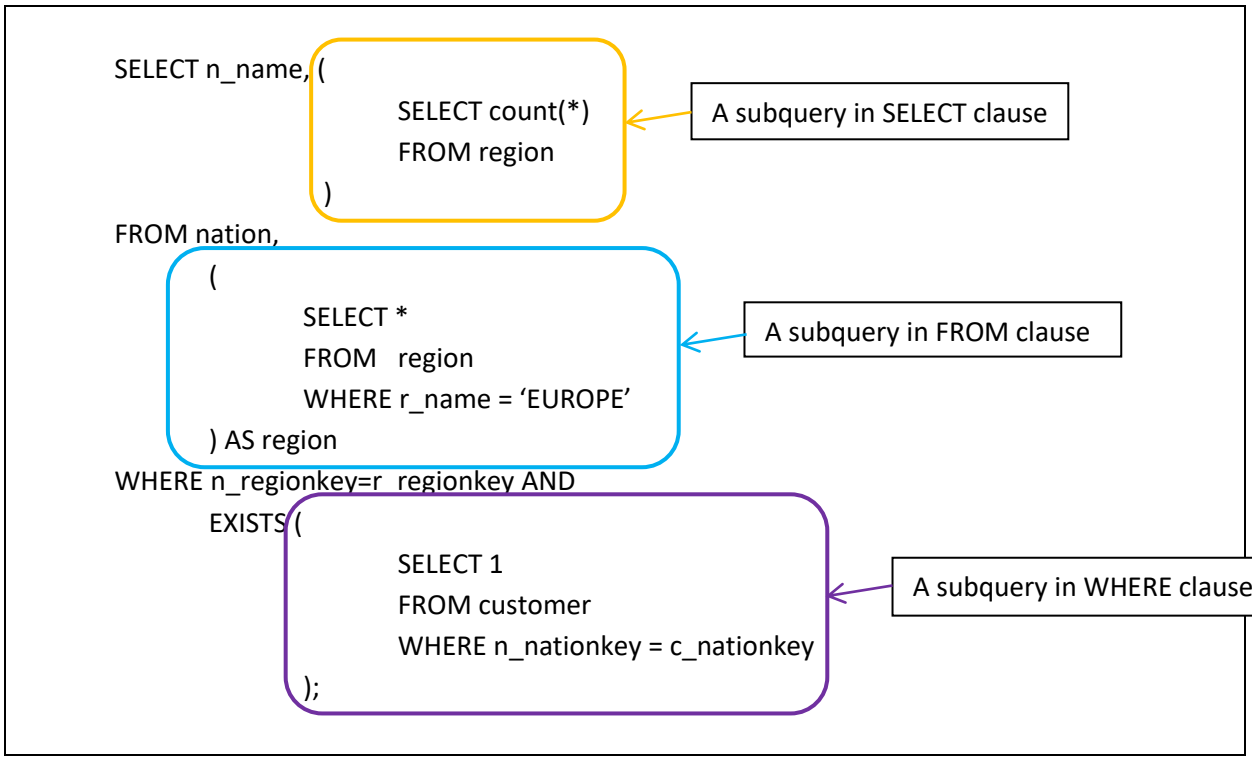
CORRELATED SUBQUERIES AND DECORRELATION

Contents

1. Introduction to subqeries, and decorrelation of correlated subqueries

Firstly we are going to define what a **subquery** is. A subquery is basically a query inside another query (sometimes called „**inner query**"), and the output of a subquery is used in the **outer query** (the query which contains a subquery).

A subquery can appear in almost any part of an SQL query, like in the SELECT clause, in the FROM clause, and in the WHERE clause. Example of a query containing a subquery in all of these three SQL statement clauses is given:



A so called **scalar subquery** is a suquery which produces just a single value output, and can be used as a single value (such a query is the query which is in the above example circled in orange).

Another type of a subquery is so called **set-value subquery**, which is a subquery which produces set of values (not a single value). Example of such a query is the blue query from the example above.

By the SQL standard, if you use a subquery which produces a new relation (i.e. table, or set of values), you must give it a name (like it was done in the case of the blue subquery with the construction "AS region").

Now we will define what **correlated subqueries** are and how do they affect the execution time of a query. A correlated subquery is **a subquery which depends on something (some column, or table) that was not defined in that subquery, but rather in the outer query**. Writing such queries may be convenient for the people who write the query, but there correlated subqueries are one of the main reasons why queries do not finish at all (or at least on time). The reason for this is that **most of systems execute the correlated subquery for every row of the outer query**, making the execution

time rise quadratic. This problem can be fixed by converting the correlated subquery into a subquery which is not correlated, and this process is called **decorrelation**. Some systems do decorrelation by themselves, but of the systems don't, and even the ones who do perform decorrelation can't perform the decorrelation process for any correlated subqueries. But, because we get such a huge improvement in execution time by performing decorrelation on a correlated subquery, and because we can perform the decorrelation process by hand, we should always do it.

An example of a correlated subquery is given below:

```
SELECT avg(l1.l_extendedprice)
FROM lineitem l1
WHERE l1.l_extendedprice = (
                                    SELECT min(l2.l_extendedprice)
                                    FROM lineitem l2
                                    WHERE l1.l_orderkey = l2.l_orderkey
                            );
```

2. Examples of decorrelation of correlated queries

Example 1:

correlated:

```
SELECT
        sum(l1.l_extendedprice)/7.0 AS avg_yearly
FROM
        lineitem l1,
        part p
WHERE
        p.p_partkey = l1.l_partkey AND
        p.p_brand = 'Brand#23' AND
        p.p_container = 'MED BOX' AND
        l1.l_quantity < (
                        SELECT 0.2*avg(l2.l_quantity)
                        FROM lineitem l2
                        WHERE l2.l_partkey = p.p_partkey
                       );
```

uncorrelated, (1 way):

To perform decorrelation of a correlated query, where in a “WHERE” clause of the outer query we have a correlated subquery which has one condition in its “WHERE” clause that uses a reference to a column defined in the outer query, we need to follow the steps:

1. Make a subquery in the “FROM” clause of the outer query
2. In the newly made subquery, in the “FROM” clause put the same tables as the ones specified in the “FROM” clause of the correlated subquery
3. In the newly made subquery, in the “SELECT” clause put all the columns from the “SELECT” clause of the correlated subquery, and as well put the column used in the “WHERE” clause of the correlated subquery which is not referring to anything defined outside the correlated subquery
4. In the newly made subquery make a “GROUP BY” clause, and put in it all of the columns which were at the end of step 3 listed in the “SELECT” clause of the newly made subquery
5. By the SQL standard you are obliged to assign this newly made subquery a name, so add it by using “AS name” syntax after the closing brackets of the newly made subquery
6. The “SELECT” of the outer query should stay as it is
7. Delete the correlated subquery from the “WHERE” clause of the outer query, as well as the condition in which the correlated subqery participated in
8. Add a condition in the “WHERE” clause of the outer query connecting the newly made subqery (from the “FROM” clause), and the outer query, by adding condition of equality among columns of these two for which the correlation happened in the first place
9. Add a condition in the “WHERE” clause of the outer query which resembles the condition for which a correlated subqery existed in the first place

```
SELECT
        sum(l1.l_extendedprice)/7.0 AS avg_yearly
FROM
        lineitem l1,
        part p,
        (
                SELECT
                        0.2*avg(l2.l_quantity) AS yearavg,
                        l2.l_partkey
                FROM
                        lineitem l2
                GROUP BY
                        l2.l_partkey
        ) AS uncor
WHERE
        p.p_partkey = l1.l_partkey AND
        p.p_brand = 'Brand#23' AND
        p.p_container = 'MED BOX' AND
        uncor.l_partkey = l1.l_partkey AND
        l1.l_quantity < uncor.yearavg;
```

uncorrelated, [2 way, (slower than way 1)]:

```
SELECT
        sum(l1.l_extendedprice)/7.0 AS avg_yearly
FROM
        lineitem l1,
        part p,
        (
                SELECT
                        0.2*avg(l2.l_quantity) AS yearavg,
                        domain_set.l_partkey
                FROM
                        lineitem l2,
                        (
                                SELECT DISTINCT
                                        l3.l_partkey
                                FROM
                                        lineitem l3
                        ) AS domain_set
                WHERE
                        domain_set.l_partkey = l2.l_partkey
                GROUP BY
                        domain_set.l_partkey
        ) AS uncor
WHERE
        p.p_partkey = l1.l_partkey AND
        p.p_brand = 'Brand#23' AND
        p.p_container = 'MED BOX' AND
        l1.l_partkey = uncor.l_partkey AND
        l1.l_quantity < uncor.yearavg;
```

Example 2:

correlated:

```
SELECT sum(l1.l_extendedprice)
FROM lineitem l1
WHERE l1.l_extendedprice > (
                                    SELECT avg(l2.l_extendedprice)
                                    FROM lineitem l2
                                    WHERE l2.l_orderkey = l1.l_orderkey);
```

---

uncorrelated, (1 way):

```
SELECT sum(l1.l_extendedprice)
FROM
        lineitem l1,
        (
                SELECT
                        avg(l2.l_extendedprice) AS avgex,
                        l2.l_orderkey
                FROM
                        lineitem l2
                GROUP BY
                        l2.l_orderkey
        ) AS uncor
WHERE
        l1.l_orderkey = uncor.l_orderkey AND
        l1.l_extendedprice > uncor.avgex
```

uncorrelated, [2 way, (slower than way 1)]:

```
SELECT sum(l1.l_extendedprice)
FROM
        lineitem l1,
        (
                SELECT
                        avg(l2.l_extendedprice) AS avgprc,
                        domain_set.l_orderkey
                FROM
                        lineitem l2,
                                (
                                        SELECT DISTINCT
                                                l3.l_orderkey
                                        FROM
                                                lineitem l3
                        ) AS domain_set
                WHERE
                        domain_set.l_orderkey = l2.l_orderkey
                GROUP BY
                        domain_set.l_orderkey
        ) AS decor
WHERE
        l1.l_orderkey = decor.l_orderkey AND
        l1.l_extendedprice > decor.avgprc;
```

Example 3:

correlated:

```
SELECT o1.o_orderkey
FROM orders o1
WHERE o1.o_totalprice < (
                                SELECT avg(o2.o_totalprice )
                                FROM orders o2
                                WHERE o2.o_shippriority = o1.o_shippriority OR
                                        o2.o_orderstatus = o1.o_orderstatus
                        );
```

uncorrelated:

```
SELECT o1.o_orderkey
FROM
        orders o1,
        (
                SELECT
                        avg(o2. o_totalprice) AS avgprc,
                        domain_set.o_shippriority,
                        domain_set.o_orderstatus
                FROM
                        orders o2,
                        (
                                SELECT DISTINCT
                                        o3.o_shippriority,
                                        o3.o_orderstatus
                                FROM
                                        orders o3
                        ) AS domain_set
                WHERE
                        domain_set.o_shippriority = o2.o_shippriority OR
                        domain_set.o_orderstatus = o2.o_orderstatus
                GROUP BY
                        domain_set.o_shippriority,
                        domain_set.o_orderstatus
        ) AS uncor
WHERE
        o1.o_shippriority = uncor.o_shippriority AND
        o1.o_orderstatus = uncor.o_orderstatus AND
        o1.o_totalprice < uncor.avgprc;
```

WITH RECURSIVE statement (recursive common table expression-CTE)

Contents

1. Introduction to WITH RECURSIVE statement

Using WITH RECURSIVE statement is not really using recursion, but rather iteration (in professors opinion, and he is right).

Firstly the structure of a WITH RECURSIVE statement is going to be introduced:

```
WITH RECURSIVE name (list_or_input_parameters) AS (
                                                    A NON-RECURSIVE TERM
                                                        UNION ALL
                                                    A RECURSIVE TERM)
SELECT list_of_columns
FROM table
WHERE specify_which_entries_from_the_statement_output;
```

The first want to discuss what is the difference between A NON-RECURSIVE TERM and A RECURSIVE TERM. These are also called the first and second queries of the "WITH RECURSIVE" statement. A NON-RECURSIVE TERM is a query which gives an output in each iteration which is the resulting output of the previous iteration of the statement, while A RECURSIVE TERM is the term which is a query containing a call to the same "WITH RECURSIVE" statement as the one in which it is located, and it contains the terminating condition in its "WHERE" clause, which if it is not specified carefully, could lead to an infinite loop.

If we on the other hand thing of a "WITH RECURSIVE" statement as a loop, we can think of it as the following. When a recursive CTE query runs, the first query (the NON-RECURSIVE TERM) generates one or more beginning rows which are added to the result set. Then, the second query is run and the resulting rows of this query are added to the result set. This continues so that the second query is run against all the rows from the last iteration, and the new resulting rows are added to the result set. The procedure ends when no more rows are returned by from the second query.

The algorithm look as following:

```
working_table = evaluate_non_recursive_term()
output(working_table)

while(working_table != empty)
{
        working_table = evaluate_recursive_term(working_table)

        output(working_table)
}
```

1.2 Examples of use of WITH RECURSIVE statement

Example 1.1:

```
WITH RECURSIVE r(i) AS (
                        SELECT 1 AS i
                                UNION ALL
                        SELECT i+1
                        FROM r
                        WHERE r.i < 5
                        )
SELECT *
FROM r;
```

A NON-RECURSIVE TERM

You can look at it as input values parameters of a loop will have.

A RECURSIVE TERM

It contains the name of the WITH RECURSIVE statement in the „FROM“ clause, and a terminating condition in the „WHERE“ clause.

name of WITH RESURSIVE statement in „FROM“ clause

termination condition in the „WHERE“ clause

Notice that there is no comma, nor semicolon at the end of the WITH RECURSIVE statement

```
neumann=# with recursive r(i) as (select 1 union all select i+1 from r where r.i<5) select * from r;
 i
---
 1
 2
 3
 4
 5
(5 rows)
```

Example 1.2:

```
WITH RECURSIVE r(i) AS (
                        SELECT 1 AS i
                                UNION ALL
                        SELECT i+1
                        FROM r
                        WHERE r.i < 10)
SELECT *
FROM r;
```

```
neumann=# with recursive r(i) as (select 1 union all select i+1 from r where r.i<10) select * from r;
 i
----
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
(10 rows)
```

Example 2:

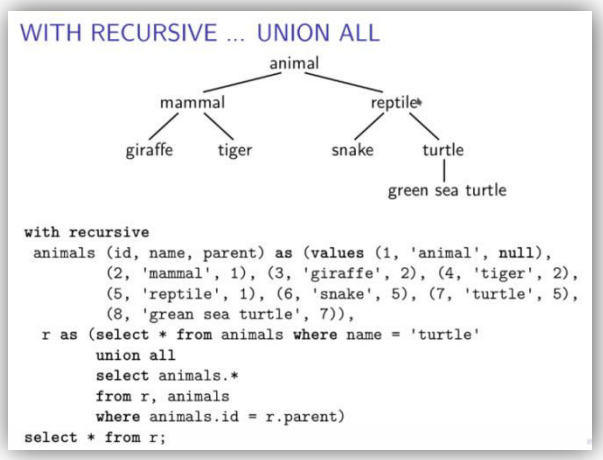
Using "WITH RCURSIVE" statement for traversing a tree-like structure differs a little bit from standard use of "WITH RECURSIVE", so you should look at these two 'types' of uses of "WITH RECURSIVE" statements differently.

With the following code we are just presenting the content of the table „animals":

```
animals(id, name, parent) AS (
        VLAUES (1, 'animal', null),
                (2, 'mammal', 1),
                (3, 'giraffe', 2),
                (4, 'tiger', 2),
                (5, 'reptile', 1),
                (6, 'snake', 5),
                (7, 'turtle', 5),
                (8, 'green sea turtle', 7))
SELECT *
FROM animals;
```

```
neumann-#  animals (id, name, parent) as (values (1, 'animal', null),
neumann(#          (2, 'mammal', 1), (3, 'giraffe', 2), (4, 'tiger', 2),
neumann(#          (5, 'reptile', 1), (6, 'snake', 5), (7, 'turtle', 5),
neumann(#          (8, 'grean sea turtle', 7)) select * from animals;
 id |       name       | parent
----+------------------+--------
  1 | animal           |
  2 | mammal           |      1
  3 | giraffe          |      2
  4 | tiger            |      2
  5 | reptile          |      1
  6 | snake            |      5
  7 | turtle           |      5
  8 | grean sea turtle |      7
(8 rows)
```

The following picture is a from FDE lecture "Using a Database", slide number 20, which shows in a form of a tree how the entries of the above table are connected with one another.



As well, from the code on the screenshot above you see that it is unnecessary to write an additional "d" in the "FROM" clause in the second query of the "WITH RECURSIVE" statement, as professor did in the following examples.

Example 2.1:

In this query (which has the WITH RECURSIVE statement) we just copied the table “animals” we outputted above (in “Example 2”).
In this example we are outputting all the animals from the tree which are in the hierarchy **below** a “**turtle**”.

```
WITH RECURSIVE animals(id, name, parent) AS (
                                        VLAUES (1, 'animal', null),
                                               (2, 'mammal', 1),
                                               (3, 'giraffe', 2),
                                               (4, 'tiger', 2),
                                               (5, 'reptile', 1),
                                               (6, 'snake', 5),
                                               (7, 'turtle', 5),
                                               (8, 'green sea turtle', 7)),
                d AS (
                        SELECT a1.*
                        FROM animals a1
                        WHERE a1.name='turtle'
                                UNION ALL
                        SELECT a2.*
                        FROM animals a2, d
                        WHERE a2.parent=d.id)
SELECT *
FROM d;
```

```
neumann=# with recursive
 animals (id, name, parent) as (values (1, 'animal', null),
        (2, 'mammal', 1), (3, 'giraffe', 2), (4, 'tiger', 2),
        (5, 'reptile', 1), (6, 'snake', 5), (7, 'turtle', 5),
        (8, 'grean sea turtle', 7)), d as (select * from animals where name='turtle' union all select a.* from animals a, d d where a.parent=d
.id) select * from d;
 id |       name       | parent
----+------------------+--------
  7 | turtle           |      5
  8 | grean sea turtle |      7
(2 rows)
```

Example 2.2:

Same as example 2.1, but instead of looking all the animals which are in the hierarchy below a „turtle“ in the tree, we are looking at all animals which are in the hierarchy **below** a „**reptile**“.

```
WITH RECURSIVE animals(id, name, parent) AS (
                                        VLAUES (1, 'animal', null),
                                               (2, 'mammal', 1),
                                               (3, 'giraffe', 2),
                                               (4, 'tiger', 2),
                                               (5, 'reptile', 1),
                                               (6, 'snake', 5),
                                               (7, 'turtle', 5),
                                               (8, 'green sea turtle', 7)),
                d AS (
                        SELECT a1.*
                        FROM animals a1
                        WHERE a1.name='reptile'
                                UNION ALL
                        SELECT a2.*
                        FROM animals a2, d
                        WHERE a2.parent=d.id)
SELECT *
FROM d;
```

```
neumann=# with recursive
 animals (id, name, parent) as (values (1, 'animal', null),
        (2, 'mammal', 1), (3, 'giraffe', 2), (4, 'tiger', 2),
        (5, 'reptile', 1), (6, 'snake', 5), (7, 'turtle', 5),
        (8, 'grean sea turtle', 7)), d as (select * from animals where name='reptile' union all select a.* from animals a, d d where a.parent=
d.id) select * from d;
 id |       name       | parent
----+------------------+--------
  5 | reptile          |      1
  7 | turtle           |      5
  6 | snake            |      5
  8 | grean sea turtle |      7
(4 rows)
```

“SELECT a2.*” specifies that we select all the columns of table “a2”, which is equivalent as if we wrote “SELECT a2.id, a2.name, a2.parent”

Example 2.3:

Same as “Example 2.1”, but instead of looking all the animals which are in the hierarchy below a „turtle“ in the tree, we are looking at all animals which are in the hierarchy **above** a „**turtle**“.

```
WITH RECURSIVE animals(id, name, parent) AS (
                                        VLAUES (1, 'animal', null),
                                               (2, 'mammal', 1),
                                               (3, 'giraffe', 2),
                                               (4, 'tiger', 2),
                                               (5, 'reptile', 1),
                                               (6, 'snake', 5),
                                               (7, 'turtle', 5),
                                               (8, 'green sea turtle', 7)),
                d AS (
                        SELECT a1.*
                        FROM animals a1
                        WHERE a1.name='turtle'
                                UNION ALL
                        SELECT a2.*
                        FROM animals a2, d
                        WHERE a2.id=d.parent)
SELECT *
FROM d;
```

```
neumann=# with recursive
 animals (id, name, parent) as (values (1, 'animal', null),
         (2, 'mammal', 1), (3, 'giraffe', 2), (4, 'tiger', 2),
         (5, 'reptile', 1), (6, 'snake', 5), (7, 'turtle', 5),
         (8, 'grean sea turtle', 7)), d as (select * from animals where name='turtle' union all select a.* from animals a, d d where a.id=d.par
ent) select * from d;
 id |  name   | parent
----+---------+--------
  7 | turtle  |      5
  5 | reptile |      1
  1 | animal  |
(3 rows)
```

Example 2.4:

This example shows that you could use more than one constraint in the “WHERE” clause of the second query (of the two queries a “WITH RECURSIVE” statement consists of).

```
WITH RECURSIVE animals(id, name, parent) AS (
                                        VLAUES (1, 'animal', null),
                                               (2, 'mammal', 1),
                                               (3, 'giraffe', 2),
                                               (4, 'tiger', 2),
                                               (5, 'reptile', 1),
                                               (6, 'snake', 5),
                                               (7, 'turtle', 5),
                                               (8, 'green sea turtle', 7)),
            d AS (
                    SELECT a1.*
                    FROM animals a1
                    WHERE a1.name='turtle'
                            UNION ALL
                    SELECT a2.*
                    FROM animals a2, d
                    WHERE a2.parent=d.id AND a2.name LIKE ‘%t%’)
SELECT *
FROM d;
```

```
neumann=# with recursive
 animals (id, name, parent) as (values (1, 'animal', null),
         (2, 'mammal', 1), (3, 'giraffe', 2), (4, 'tiger', 2),
         (5, 'reptile', 1), (6, 'snake', 5), (7, 'turtle', 5),
         (8, 'grean sea turtle', 7)), d as (select * from animals where name='reptile' union all select a.* from animals a, d d where a.parent=
d.id and a.name like '%t%') select * from d;
 id |       name       | parent
----+------------------+--------
  5 | reptile          |      1
  7 | turtle           |      5
  8 | grean sea turtle |      7
(3 rows)
```

Example 3.1:

[Exercise from lecture “Using a Database”, slide 21]

Compute 10!, using WITH RECURSIVE statement (a recursive CTE).

```
WITH RECURSIVE factorial(n, value) AS (
                                        SELECT 1 AS n, 1 AS value
                                                UNION ALL
                                        SELECT n+1, (n+1)*value
                                        FROM factorial
                                        WHERE n<10)
SELECT value
FROM factorial
WHERE n=10;
```

```
neumann=# with recursive f(n, value) as (select 1 as n, 1 as value union all select n+1, (n+1)*value from f where n<10) select value from f whe
re n=10;
  value
---------
 3628800
(1 row)
```

Example 3.2:

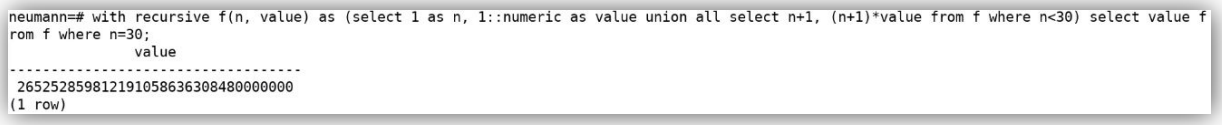
```
WITH RECURSIVE factorial (n, value) AS (
                                        SELECT 1 AS n, 1::bigint AS value
                                                UNION ALL
                                        SELECT n+1,  (n+1)*value
                                        FROM factorial
                                        WHERE n<15)
SELECT value
FROM factorial
WHERE n=15;
```

```
neumann=# with recursive f(n, value) as (select 1 as n, 1::bigint as value union all select n+1, (n+1)*value from f where n<15) select value fr
om f where n=15;
     value
---------------
 1307674368000
(1 row)
```

You can see that because of the quick growth of the factorial function that in this example (example 3.2) we had to write “1::bigint AS value” (changing the data type of the column) as the value would quickly become too big to be stored in an integer type.

Example 3.3:

```
WITH RECURSIVE factorial (n, value) AS (
                                        SELECT 1 AS n, 1::numeric AS value
                                                UNION ALL
                                        SELECT n+1, (n+1)*value
                                        FROM factorial
                                        WHERE n<30)
SELECT value
FROM factorial
WHERE n=30;
```



```
neumann=# with recursive f(n, value) as (select 1 as n, 1::numeric as value union all select n+1, (n+1)*value from f where n<30) select value f
rom f where n=30;
              value
---------------------------------
 265252859812191058636308480000000
(1 row)
```

With picking the appropriate data type you can compute pretty large factorial numbers. In this example we compute factorial 30!, and use type “numeric” to be able to represent this big number.

Example 4.1:

[Exercise from lecture “Using a Database”, slide 21]

Compute the first 20 Fibonacci numbers ($F_1=1$, $F_2=1$, $F_n=F_{n-1}+F_{n-2}$):

```
WITH RECURSIVE fibonacci(n, value, previous_value) AS (
                                        SELECT 1 AS n, 1 AS value, 0 AS previous_value
                                                UNION ALL
                                        SELECT n+1, value+previous_value, value
                                        FROM fibonacci
                                        WHERE n<20)
SELECT *
FROM fibonacci;
```

```
neumann=# with recursive fib(n, value, previousvalue) as (select 1 as n, 1 as value, 0 previousvalue union all select n+1, value+previousvalue,
 value from fib where n<20) select * from fib;
 n  | value | previousvalue
----+-------+---------------
  1 |     1 |             0
  2 |     1 |             1
  3 |     2 |             1
  4 |     3 |             2
  5 |     5 |             3
  6 |     8 |             5
  7 |    13 |             8
  8 |    21 |            13
  9 |    34 |            21
 10 |    55 |            34
 11 |    89 |            55
 12 |   144 |            89
 13 |   233 |           144
 14 |   377 |           233
 15 |   610 |           377
 16 |   987 |           610
 17 |  1597 |           987
 18 |  2584 |          1597
 19 |  4181 |          2584
 20 |  6765 |          4181
(20 rows)
```

Example 4.2:

```
WITH RECURSIVE fibonacci(n, value, previous_value) AS (
                                        (VALUES(1, 1, 0))
                                                UNION ALL
                                        SELECT n+1, value+previous_value, value
                                        FROM fibonacci
                                        WHERE n<20)
SELECT n, value
FROM fibonacci;
```

```
neumann=# with recursive fib(n, value, previousvalue) as ((values(1,1,0)) union all select n+1, value+previousvalue, value from fib where n<20)
 select n, value from fib;
 n  | value
----+-------
  1 |     1
  2 |     1
  3 |     2
  4 |     3
  5 |     5
  6 |     8
  7 |    13
  8 |    21
  9 |    34
 10 |    55
 11 |    89
 12 |   144
 13 |   233
 14 |   377
 15 |   610
 16 |   987
 17 |  1597
 18 |  2584
 19 |  4181
 20 |  6765
(20 rows)
```

As seen in this example, we can construct explicitly a single row (or “tuple”) with help of “VALUES” statement. Using “(VALUES(1, 1, 0))” is basically the same as constructing a single row using “SELECT 1 AS n, 1 AS value, 0 AS previous_value”, where we specify 1 for first parameter of the “WITH RECURSIVE” statement (which is “n”), 1 for the second parameter (which is “value”), and 0 for the third paameter (which is “previous_value”).

Example 4.3:

```
WITH RECURSIVE fibonacci(n, value, previous_value) AS (
                                        (VALUES(1, 1, 0), (2, 1, 1))
                                                UNION ALL
                                        SELECT n+1,  value+previous_value, value
                                        FROM fibonacci
                                        WHERE n<20)
SELECT n, value
FROM fibonacci;
```

```
neumann=# with recursive fib(n, value, previousvalue) as ((values(1,1,0),(2,1,1)) union all select n+1, value+previousvalue, value from fib whe
re n<20) select n, value from fib;
```

n	value
1	1
2	1
2	1
3	2
3	2
4	3
4	3
5	5
5	5
6	8
6	8
7	13
7	13
8	21
8	21
9	34
9	34
10	55
10	55
11	89
11	89
12	144
12	144
13	233
13	233
14	377
14	377
15	610
15	610
16	987
16	987
17	1597
17	1597
18	2584

We can see that if we construct the “WITH RECURSION” statement like above, output will contain duplicates. This is due to the fact that in each iteration we output the “workingTable” and, it contains two values. We now have two values in the “workingTable” in each iteration, which leads to a situation where the last outputting row in an iteration and the first outputted row in the next iteration are the same.

In general it is a little bit silly to start with two rows, because this may make the problem even more difficult, but if you have some recursive formula which does require multiple starting values, you can do it like in this example.

Example 4.4:

```
WITH RECURSIVE fibonacci(n, value, previous_value) AS (
                                        (VALUES(1, 1, 0), (2, 1, 1))
                                                UNION ALL
                                        SELECT n+1, value+previous_value, value
                                        FROM fibonacci
                                        WHERE n BETWEEN 2 AND 19)
SELECT n, value
FROM fibonacci;
```

```
neumann=# with recursive fib(n, value, previousvalue) as ((values(1,1,0),(2,1,1)) union all select n+1, value+previousvalue, value from fib whe
re n between 2 and 19) select n, value from fib;
```

```
 n  | value
----+-------
  1 |     1
  2 |     1
  3 |     2
  4 |     3
  5 |     5
  6 |     8
  7 |    13
  8 |    21
  9 |    34
 10 |    55
 11 |    89
 12 |   144
 13 |   233
 14 |   377
 15 |   610
 16 |   987
 17 |  1597
 18 |  2584
 19 |  4181
 20 |  6765
(20 rows)
```

In the step of the algorithm where we expand our table, we ignore the first element. Because we explicitly gave the first two values, we no longer need to expand the first value, because it has already been expanded explicitly, so it can be ignored.

2. WITH RECURSIVE statement with UNION statement

Instead of using "UNION ALL" statement to make a connection between two queries in the "WITH RECURSIVE" clause, we are going to use "UNION" for that purpose. Although **"UNION" statement** is **not standardised** in contrast to "UNION ALL" statement, **PostgreSQL supports it**. It is very useful because it **eliminates duplicates** when running the query, meaning that **we never output an element twice**.

A great advantage of usage of "UNION" statement instead of "UNION ALL" statement is the fact that **it will never be stuck in an infinite loop caused by a graph which has a cyclic structure inside it**. This means that if we allow duplicate outputs on a graph with a cyclic structure inside it, and try to perform "WITH RECURSIVE" statement with "UNION ALL" between the two queries (the two queries that "WITH RECURSIVE" statement consists of), we will end up in infinite loop (or at least in a loop until our system recognises that the execution time it too long and terminates the queries itself).

The structure of a "WITH RECURSIVE" statement with "UNION" is of same as when the recursive CTE with "UNION ALL" statement was used, except that between two queries there is a "UNION" statement, that is:

```
WITH RECURSIVE name(list_or_input_parameters) AS (
                                                A NON-RECURSIVE TERM
                                                    UNION
                                                A RECURSIVE TERM)
SELECT list_of_columns
FROM table
WHERE specify_which_entries_from_the_statement_output;
```

The algorithm looks as following:

```
working_table = unique( evaluate_non_recursive_term() )
result = working_table

while(working_table != empty)
{
        working_table = unique( evaluate_recursive_term(working_table) ) / result

        result = result  UNION working_table
}

output(result)
```

2.1 Examples of use of WITH RECURSIVE statement with UNION

Example 1:

```
WITH RECURSIVE
        friends(a, b) AS (VALUES ('Alice', 'Bob'),
                                 ('Alice', 'Carol'),
                                 ('Carol', 'Grace'),
                                 ('Carol', 'Chuck'),
                                 ('Carol', 'Grace'),
                                 ('Chuck', Anne),
                                 ('Bob', 'Dan'),
                                 ('Dan', 'Anne'),
                                 ('Eve', 'Adam')),
        friendship(name, friend) AS (
                                 SELECT a, b,
                                 FROM friends
                                        UNION ALL
                                 SELECT b, a
                                 FROM friends)
```

We defined two tables, “friends” and “friendship”.

Example 1.1:

```
SELECT *
FROM friends;
```

```
neumann=# with recursive
neumann-#  friends (a, b) as (values ('Alice', 'Bob'), ('Alice', 'Carol'),
neumann(#    ('Carol', 'Grace'), ('Carol', 'Chuck'), ('Chuck', 'Grace'),
neumann(#    ('Chuck','Anne'),('Bob','Dan'),('Dan','Anne'),('Eve','Adam')),
neumann-#  friendship (name, friend) as -- friendship is symmetric
neumann-#     (select a, b from friends union all select b, a from friends) select * from friends;
   a   |   b
-------+-------
 Alice | Bob
 Alice | Carol
 Carol | Grace
 Carol | Chuck
 Chuck | Grace
 Chuck | Anne
 Bob   | Dan
 Dan   | Anne
 Eve   | Adam
(9 rows)
```

We firstly observe the content of the table “friends”.

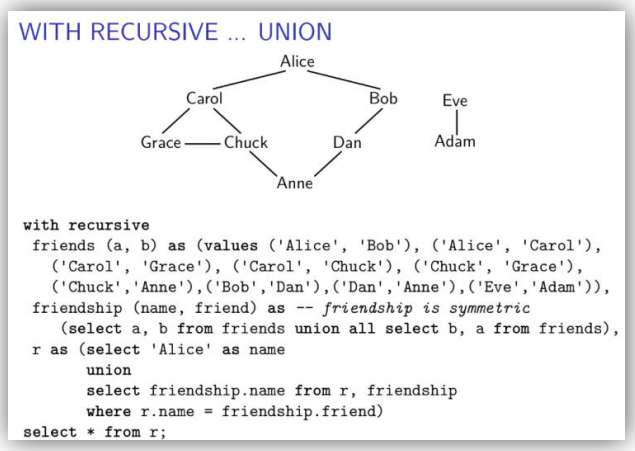
We secondly observe the content of the table “friendship”.

```
SELECT *
FROM friendship;
```

```
neumann=# with recursive
 friends (a, b) as (values ('Alice', 'Bob'), ('Alice', 'Carol'),
   ('Carol', 'Grace'), ('Carol', 'Chuck'), ('Chuck', 'Grace'),
   ('Chuck','Anne'),('Bob','Dan'),('Dan','Anne'),('Eve','Adam')),
 friendship (name, friend) as -- friendship is symmetric
    (select a, b from friends union all select b, a from friends) select * from friendship;
 name  | friend
-------+--------
 Alice | Bob
 Alice | Carol
 Carol | Grace
 Carol | Chuck
 Chuck | Grace
 Chuck | Anne
 Bob   | Dan
 Dan   | Anne
 Eve   | Adam
 Bob   | Alice
 Carol | Alice
 Grace | Carol
 Chuck | Carol
 Grace | Chuck
 Anne  | Chuck
 Dan   | Bob
 Anne  | Dan
 Adam  | Eve
(18 rows)
```

We see that table “friendship” is the same as the table “friends”, except that in this table we have a connection between two entries (or people) from both directions, meaning that for an example in table “friends” we have a row (or edge/connection, if you think about the problem in a way as if it was presented with a graph) which outputs “Alice” in the first column, and “Bob” in the second column, while on the other hand, in table “friendship” we have two output row which are contain the same two people, but in a different columns, i.e. “Alice | Bob” in in one output row, and “Bob | Alice” in the second output row (you can see these three rows from the two tables in the two screenshots above, circled in orange). In other words, table “friendship” is symmetric.

The following picture is a from FDE lecture “Using a Database”, slide number 23, which shows in a form of a fully connected graph entries of the friendship are connected with one another.



Example 1.2:

Output all people who are friends with “Alice”, where friends of “Alice” are all people connected to the “Alice” or some of her friends.

```
WITH RECURSIVE
        friends(a, b) AS (VALUES (‘Alice’, ‘Bob’),
                                 (‘Alice’, ‘Carol’),
                                 (‘Carol’, ‘Grace’),
                                 (‘Carol’, ‘Chuck’),
                                (‘Carol’, ‘Grace’),
                                 (‘Chuck’, Anne),
                                 (‘Bob’, ‘Dan’),
                                 (‘Dan’, ‘Anne’),
                                 (‘Eve’, ‘Adam’)),
        friendship(name, friend) AS (
                                 SELECT a, b,
                                 FROM friends
                                        UNION ALL
                                 SELECT b, a
                                 FROM friends),
        freindsofalice AS (
                        SELECT ‘Alice’ as name
                                UNION
                        SELECT friend
                        FROM friendship, friendofalice
                        WHERE friendship.name= friendofalice.name)
SELECT *
FROM friendofalice;
```

```
neumann=# with recursive
 friends (a, b) as (values ('Alice', 'Bob'), ('Alice', 'Carol'),
   ('Carol', 'Grace'), ('Carol', 'Chuck'), ('Chuck', 'Grace'),
   ('Chuck','Anne'),('Bob','Dan'),('Dan','Anne'),('Eve','Adam')),
 friendship (name, friend) as -- friendship is symmetric
    (select a, b from friends union all select b, a from friends), friendsofalice as (select 'Alice' as name union select friend from friendshi
p, friendsofalice where friendship.name=friendsofalice.name) select * from friendsofalice;
 name
-------
 Alice
 Bob
 Carol
 Grace
 Chuck
 Dan
 Anne
(7 rows)
```

Example 1.3:

Output all people who are friends with “Dan”, where friends of “Dan” are all people connected to the “Dan” or some of his friends.

```
WITH RECURSIVE
        friends(a, b) AS (VALUES (‘Alice’, ‘Bob’),
                                 (‘Alice’, ‘Carol’),
                                 (‘Carol’, ‘Grace’),
                                 (‘Carol’, ‘Chuck’),
                                 (‘Carol’, ‘Grace’),
                                 (‘Chuck’, Anne),
                                 (‘Bob’, ‘Dan’),
                                 (‘Dan’, ‘Anne’),
                                 (‘Eve’, ‘Adam’)),
        friendship(name, friend) AS (
                                 SELECT a, b,
                                 FROM friends
                                        UNION ALL
                                 SELECT b, a
                                 FROM friends),
        freindsofalice AS (
                        SELECT ‘Dan’ as name
                                UNION
                        SELECT friend
                        FROM friendship, friendofalice
                        WHERE friendship.name= friendofalice.name)
SELECT *
FROM friendofalice;
```

```
neumann=# with recursive
 friends (a, b) as (values ('Alice', 'Bob'), ('Alice', 'Carol'),
   ('Carol', 'Grace'), ('Carol', 'Chuck'), ('Chuck', 'Grace'),
   ('Chuck','Anne'),('Bob','Dan'),('Dan','Anne'),('Eve','Adam')),
 friendship (name, friend) as -- friendship is symmetric
    (select a, b from friends union all select b, a from friends), friendsofalice as (select 'Dan' as name union select friend from friendship,
 friendsofalice where friendship.name=friendsofalice.name) select * from friendsofalice;
 name
-------
 Dan
 Anne
 Bob
 Alice
 Chuck
 Carol
 Grace
(7 rows)
```

In the query above, in contrast to the query from example 1.1, we just changed the person in the NON-RECURSIVE TERM (the first query of the WITH RECURSIVE statement) from “Alice” to “Dan”. Although we did this, we did not change the table name from “friendsofalice” and it may look not appropriate for looking at friends of someone else, but it is conceptually the same (the table name is not important).

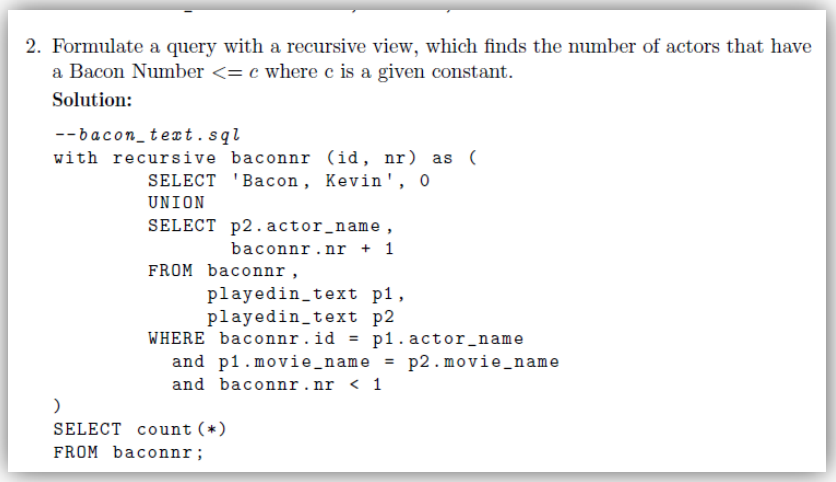
We get the same result, but only in different order (for example, we start with “Dan” in this example, as he is specified in the query as the starting point and he is outputted first), and that is because all the people which are connected to “Alice” are connected to “Dan” as well, and it doesn’t matter where (or better said from whom) you start, you will always get everybody who are in the same part of a fully connected graph.

Example 2:

[Exercise sheet 7, section “Formulating the Query”, exercise 2]

“Formulate a query with a recursive view, which finds the number of actors that have a Bacon Number <= c where c is a given constant.”

```
WITH RECURSIVE baconnr(id, nr) AS (
                        SELECT ‘Bacon, Kevin’, 0
                                UNION
                        SELECT p2.actor_name, baconnr.nr+1
                        FROM baconnr, playedin_text p1, playedin_text p2
                        WHERE baconnr.id=p1.actor_name AND
                                p1.movie_name=p2.movie_name AND
                                baconnr.nr<1)
SELECT count(*)
FROM baconnr;
```



2. Formulate a query with a recursive view, which finds the number of actors that have a Bacon Number <= c where c is a given constant.

Solution:

```
--bacon_text.sql
with recursive baconnr (id, nr) as (
        SELECT 'Bacon, Kevin', 0
        UNION
        SELECT p2.actor_name,
               baconnr.nr + 1
        FROM baconnr,
             playedin_text p1,
             playedin_text p2
        WHERE baconnr.id = p1.actor_name
          and p1.movie_name = p2.movie_name
          and baconnr.nr < 1
)
SELECT count(*)
FROM baconnr;
```

Example 3:

[Exercise sheet 7, section “Tweaking the Query”, exercise 2]

“Is there any mean to reduce the size of intermediate results and thus speed up the query evaluation? (Hint: Try to remove duplicates after every join. With this technique, the query should finish within reasonable time for c = 2.)”

```
WITH RECURSIVE baconnr(id, nr) AS (
                    SELECT ‘Bacon, Kevin’, 0
                        UNION ALL
                    SELECT DISTINCT p2.actor_name, movies.nr+1
                    FROM (
                        SELECT DISTINCT p1.movie_name, baconnr.nr
                        FROM baconnr, playedin_text p1
                        WHERE baconnr.avtor_name=p1.actor_name
                        ) AS movies,
                    WHERE movies.movie_name=p2.movie_name AND movies.nr<5
                    )
SELECT count(DISTINCT actor_name)
FROM baconnr;
```

Solution:

```
-- bacon_unique_text.sql
WITH recursive baconnr (actor_name, nr) as (
        SELECT 'Bacon, Kevin', 0
UNION ALL
        SELECT distinct p2.actor_name,
           movies.nr + 1
        FROM (SELECT distinct p1.movie_name,
                     baconnr.nr
          FROM baconnr,
               playedin_text p1
          WHERE baconnr.actor_name = p1.actor_name
          ) movies,
          playedin_text p2
        WHERE movies.movie_name = p2.movie_name
      and movies.nr < 5
)
SELECT count(distinct actor_name)
FROM baconnr;
```

WINDOW FUNCTIONS, AND ORDERED-SET FUNCTIONS

Contents

1. WINDOW FUNCTIONS

1.1 Introduction to window functions

Syntax for window functions:

```
SELECT columns_which_will_be_outputted, window_function() OVER (
                                        PARTITION BY some_column
                                        ORDER BY some_column
                                        RANGE/ROWS BETWEEN value1 preceding AND value2 following)
FROM table;
```

Firstly, instead of columns_which_will_be_outputted we write some column names from the table. Secondly, instead of window_function() we write one window function, some of which are:

1) sum(column)
2) max(column)
3) min(column)
4) avg(column)
5) cout(column)
6) rank()
7) dense_rank()
8) row_number()
9) ntile(integer_number)
10) percent_rank()
11) cume_dist()
12) lag(column, integer_number, double_number)
13) lead(column, integer_number, double_number)

The row "PARTITION BY some_column" specifies the so called **"PARTITION BY" clause**, which is the clause based on which division of rows (which are inside table) into partitions is done. To be more specific, based on values in the column some_column, the partitioning is performed. If there is no "PARTITION BY" clause specified inside a window function, then all the rows belong to one big partition.

The row "ORDER BY some_column" specifies the so called **"ORDER BY" clause**, which is the clause based on which the ordering or row inside one partition is going to be performed. To be more specific, the ordering depends on the values inside column some_column.

The row "RANGE/ROWS BETWEEN value1 preceding AND value2 following" specifies the so called **"FRAMING" clause**, and it specifies how much of rows from the same partition influence the value in the in the current row. It can start with either "**ROWS**" or "**RANGE**", where the difference is that when "ROWS" is used we literally look value1 rows which come before the current row (inside the same partition) and value2 rows (inside the same partition) which come after the current row. On the other hand, when we use "RANGE", we look at "value1 - 1" rows which come before the current

row (in the same partition), and "value2 - 1" rows which come after the current row (inside the same partition). Additionally, there are some predefined values which can be used instead of specifying value1 and value2 with integer numbers, which allow you to look at all the rows which (in the same partition) come before the current row, and all the rows which come after the current row (in the same partition). The predefined value which allows you to look at all the rows which (in the same partition) come before the current row is **unbounded proceeding**, while the value which allows you to look at all the rows which (in the same partition) come after the current row is **unbounded following**. If we specify the "FRAMING" clause as: "RANGE BETWEEN unbounded preceding AND unbounded following", then this would mean that the current row in a partition (and each of the rows inside a partition) is influenced by all the others rows inside the same partition.

1. The data is divided into partitions by the „PARTITION BY" clause
2. The rows inside each partition are sorted according to the "ORDER BY" clause
3. The used window function as a result adds one more column to the output, after it is executed on a row of the partition.

1.2 lecture 11 (FDE), examples, window functions

```
SELECT o_custkey, o_orderdate, sum(o_totalprice) OVER (
                                    PARTITION BY o_custkey
                                    ORDER BY o_orderdate
                                    RANGE BETWEEN unbounded preceding AND current row)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, sum(o_totalprice) over (partition by o_custkey order by o_orderdate range between unbounded preceding
and current row) from orders;
```

o_custkey	o_orderdate	sum	
1	1992-04-19	74602.81	
1	1992-08-22	197679.65	= 74602.81 + current_price
1	1996-06-29	263157.70	= 197679.65 + current_price = 74602.81+ price_row_2+current_price
1	1996-07-01	437803.64	
1	1996-12-09	491851.90	
1	1997-03-23	587762.91	
2	1992-04-05	167016.61	
2	1994-05-21	270314.29	
2	1994-08-28	286809.62	
2	1994-12-24	319892.45	
2	1995-03-10	541289.80	
2	1996-08-05	715581.21	
2	1997-02-22	1028273.43	
4	1992-04-26	311722.87	
4	1992-09-20	494678.88	
4	1993-10-04	582996.07	
4	1994-06-10	623343.55	
4	1995-05-06	728278.20	
4	1995-11-01	1005771.24	
4	1996-01-03	1051999.18	
4	1996-01-06	1277293.44	
4	1996-06-03	1421264.98	
4	1996-06-06	1656886.81	
4	1996-08-02	1971558.63	
4	1996-08-11	1979890.96	
4	1996-11-29	2132820.76	
4	1997-03-07	2208749.25	
4	1997-06-12	2209880.45	
4	1997-07-12	2227818.86	
4	1997-09-07	2299664.12	
4	1997-10-03	2390817.09	
4	1997-11-23	2466918.04	
4	1998-05-16	2648536.79	
5	1993-06-27	324835.83	

partition 1

partition 2

partition 3

When specifying the range as:
„**RANGE BETWEEN unbounded preceding AND current row**“
this will mean that a row inside a partition is influenced by all the rows which come **before** it in a partition.
So for an example the first row in the first partition would not be influenced by any other row, as there are no rows which come before it. On the other hand, the second row in the partition would be influenced only by the rows which come before it, and there is only one row which comes before it, so only the first row would influence it.

```
SELECT o_custkey , o_orderdate, sum(o_totalprice) OVER (
                                        PARTITION BY o_custkey
                                        ORDER BY o_orderdate
                                        RANGE BETWEEN current row AND unbounded following)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, sum(o_totalprice) over (partition by o_custkey order by o_orderdate range between current row and unbo
unded following) from orders;
```

o_custkey	o_orderdate	sum	
1	1992-04-19	587762.91	6
1	1992-08-22	513160.10	5
1	1996-06-29	390083.26	4
1	1996-07-01	324605.21	3
1	1996-12-09	149959.27	2
1	1997-03-23	95911.01	1
2	1992-04-05	1028273.43	
2	1994-05-21	861256.82	
2	1994-08-28	757959.14	
2	1994-12-24	741463.81	
2	1995-03-10	708380.98	
2	1996-08-05	486983.63	
2	1997-02-22	312692.22	
4	1992-04-26	2648536.79	
4	1992-09-20	2336813.92	
4	1993-10-04	2153857.91	
4	1994-06-10	2065540.72	
4	1995-05-06	2025193.24	
4	1995-11-01	1920258.59	
4	1996-01-03	1642765.55	
4	1996-01-06	1596537.61	
4	1996-06-03	1371243.35	
4	1996-06-06	1227271.81	
4	1996-08-02	991649.98	
4	1996-08-11	676978.16	
4	1996-11-29	668645.83	
4	1997-03-07	515716.03	
4	1997-06-12	439787.54	
4	1997-07-12	438656.34	
4	1997-09-07	420717.93	
4	1997-10-03	348872.67	
4	1997-11-23	257719.70	
4	1998-05-16	181618.75	
5	1993-06-27	684965.28	

When specifying the range as:
„**RANGE BETWEEN current row AND unbounded following**“ this will mean that a row inside a partition is influenced by all the rows which come **after** it in a partition.
So for an example the first row in the first partition would be a sum of all values in column „o_totalprice“ which are specific for the rows which come after it inside a partition. On the other hand, the second row in the partition would be influenced by the rows which come after it inside a partition, which are all rows except the first row of the partition.

```
SELECT o_custkey, o_orderdate, sum(o_totalprice) OVER (
                                        PARTITION BY o_custkey
                                        ORDER BY o_orderdate
                                        RANGE BETWEEN unbounded preceding AND unbounded following)
FROM orders;
```

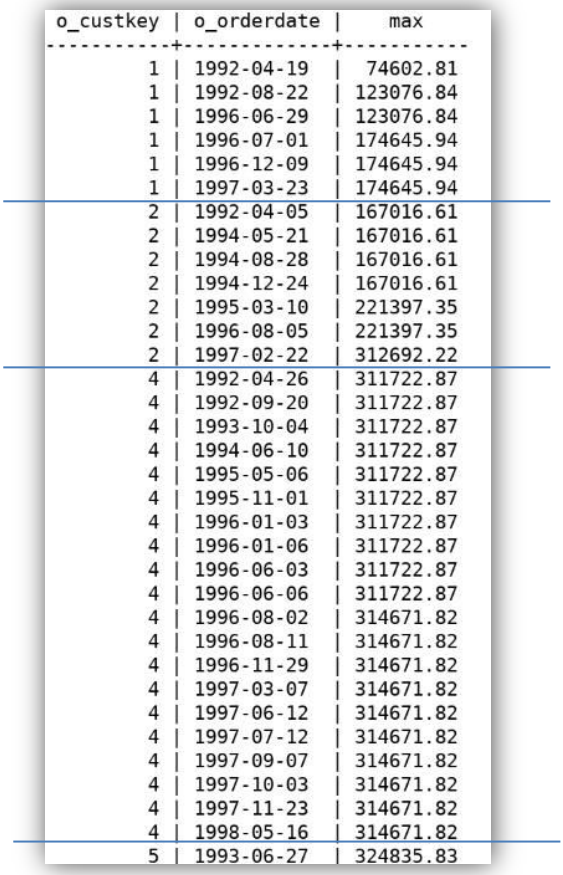
```
neumann=# select o_custkey, o_orderdate, sum(o_totalprice) over (partition by o_custkey order by o_orderdate range between unbounded preceding
and unbounded following) from orders;
```

o_custkey	o_orderdate	sum
1	1992-04-19	587762.91
1	1992-08-22	587762.91
1	1996-06-29	587762.91
1	1996-07-01	587762.91
1	1996-12-09	587762.91
1	1997-03-23	587762.91
2	1992-04-05	1028273.43
2	1994-05-21	1028273.43
2	1994-08-28	1028273.43
2	1994-12-24	1028273.43
2	1995-03-10	1028273.43
2	1996-08-05	1028273.43
2	1997-02-22	1028273.43
4	1992-04-26	2648536.79
4	1992-09-20	2648536.79
4	1993-10-04	2648536.79
4	1994-06-10	2648536.79
4	1995-05-06	2648536.79
4	1995-11-01	2648536.79
4	1996-01-03	2648536.79
4	1996-01-06	2648536.79
4	1996-06-03	2648536.79
4	1996-06-06	2648536.79
4	1996-08-02	2648536.79
4	1996-08-11	2648536.79
4	1996-11-29	2648536.79
4	1997-03-07	2648536.79
4	1997-06-12	2648536.79
4	1997-07-12	2648536.79
4	1997-09-07	2648536.79
4	1997-10-03	2648536.79
4	1997-11-23	2648536.79
4	1998-05-16	2648536.79
5	1993-06-27	684965.28

Outputted values for each row, among rows which are in the same partition, have the same value if you formulate the window function as above. More specifically, if you specify the range as:
„**RANGE BETWEEN unbounded preceding AND unbounded following**“
This will mean that a single row inside a partition will be influenced by all other rows from the same partition.

```
SELECT o_custkey, o_orderdate, max(o_totalprice) OVER (
                                        PARTITION BY o_custkey
                                        ORDER BY o_orderdate
                                        RANGE BETWEEN unbounded preceding AND current row)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, max(o_totalprice) over (partition by o_custkey order by o_orderdate range between unbounded preceding
and current row) from orders;
```



o_custkey	o_orderdate	max
1	1992-04-19	74602.81
1	1992-08-22	123076.84
1	1996-06-29	123076.84
1	1996-07-01	174645.94
1	1996-12-09	174645.94
1	1997-03-23	174645.94
2	1992-04-05	167016.61
2	1994-05-21	167016.61
2	1994-08-28	167016.61
2	1994-12-24	167016.61
2	1995-03-10	221397.35
2	1996-08-05	221397.35
2	1997-02-22	312692.22
4	1992-04-26	311722.87
4	1992-09-20	311722.87
4	1993-10-04	311722.87
4	1994-06-10	311722.87
4	1995-05-06	311722.87
4	1995-11-01	311722.87
4	1996-01-03	311722.87
4	1996-01-06	311722.87
4	1996-06-03	311722.87
4	1996-06-06	311722.87
4	1996-08-02	314671.82
4	1996-08-11	314671.82
4	1996-11-29	314671.82
4	1997-03-07	314671.82
4	1997-06-12	314671.82
4	1997-07-12	314671.82
4	1997-09-07	314671.82
4	1997-10-03	314671.82
4	1997-11-23	314671.82
4	1998-05-16	314671.82
5	1993-06-27	324835.83

In this example we use window function "**max(column)**", and with help of the "FRAMING" clause we try to find the biggest value inside a partition up until some point. Value outputted in the column „max" here represents the maximum value inside a partition which is read up until that moment.

```
SELECT o_custkey, o_orderdate, rank() OVER (
                                PARTITION BY o_custkey
                                ORDER BY o_orderdate)
FROM orders;
```

Notice that there is no “FRAMING” clause.

```
neumann=# select o_custkey, o_orderdate, rank() over (partition by o_custkey order by o_orderdate) from orders;
```

o_custkey	o_orderdate	rank
1	1992-04-19	1
1	1992-08-22	2
1	1996-06-29	3
1	1996-07-01	4
1	1996-12-09	5
1	1997-03-23	6
2	1992-04-05	1
2	1994-05-21	2
2	1994-08-28	3
2	1994-12-24	4
2	1995-03-10	5
2	1996-08-05	6
2	1997-02-22	7
4	1992-04-26	1
4	1992-09-20	2
4	1993-10-04	3
4	1994-06-10	4
4	1995-05-06	5
4	1995-11-01	6
4	1996-01-03	7
4	1996-01-06	8
4	1996-06-03	9
4	1996-06-06	10
4	1996-08-02	11
4	1996-08-11	12
4	1996-11-29	13
4	1997-03-07	14
4	1997-06-12	15
4	1997-07-12	16
4	1997-09-07	17
4	1997-10-03	18
4	1997-11-23	19

By using window function “**rank()**”, we assign each unique row inside a partition a number which is bigger than the previous assigned number (bigger by 1).

```
SELECT n_name, n_regionkey,  rank() OVER (
                                PARTITION BY substr(n_name, 1, 1)
                                ORDER BY n_regionkey)
FROM nation;
```

```
neumann=# select n_name, n_regionkey, rank() over (partition by substr(n_name,1,1) order by n_regionkey) from nation;
          n_name          | n_regionkey | rank
--------------------------+-------------+------
 ALGERIA                  |           0 |    1
 ARGENTINA                |           1 |    2
 BRAZIL                   |           1 |    1
 CANADA                   |           1 |    1
 CHINA                    |           2 |    2
 ETHIOPIA                 |           0 |    1
 EGYPT                    |           4 |    2
 FRANCE                   |           3 |    1
 GERMANY                  |           3 |    1
 INDIA                    |           2 |    1
 INDONESIA                |           2 |    1
 IRAN                     |           4 |    3
 IRAQ                     |           4 |    3
 JAPAN                    |           2 |    1
 JORDAN                   |           4 |    2
 KENYA                    |           0 |    1
 MOROCCO                  |           0 |    1
 MOZAMBIQUE               |           0 |    1
 PERU                     |           1 |    1
 ROMANIA                  |           3 |    1
 RUSSIA                   |           3 |    1
 SAUDI ARABIA             |           4 |    1
 UNITED STATES            |           1 |    1
 UNITED KINGDOM           |           3 |    2
 VIETNAM                  |           2 |    1
(25 rows)
```

When performing operation „**rank()**“ there is a possibility of getting **ties** among multiple entries inside a partition. This happens depending on what is written inside the „ORDER BY“ clause.
In this example the ties which are circled in red happened due to the fact that ordering of the rows inside a partition (which is specified by the “ORDER BY” clause) was done based on the content of the column “n_regionkey”.
For an example if we look at the rows which contain values “INDIA” and “INDONESIA”, we can see that they both have value “2” in column “n_regionkey”, which is the reason why these two rows (which belong to the same partition as partitioning was done based on the first letter of value from column “n_name”) have the same value outputted in column “rank”. It could have happened that we have more than 2 values with the same rank. The next row which belongs to the same partition will be assigned value 3, as there were three rows before it in the partition, no matter if these two ended up having a tie. If we for an example had 6 rows in a same partition which had same values (i.e. there was a tie among them), they would all be assigned value 1, while the following row would be assigned number 7.

```
SELECT n_name, n_regionkey,  dense_rank() OVER (
                                PARTITION BY substr(n_name, 1, 1)
                                ORDER BY n_regionkey)
FROM nation;
```

```
neumann=# select n_name, n_regionkey, dense_rank() over (partition by substr(n_name,1,1) order by n_regionkey) from nation;
          n_name          | n_regionkey | dense_rank
--------------------------+-------------+------------
 ALGERIA                  |           0 |          1
 ARGENTINA                |           1 |          2
 BRAZIL                   |           1 |          1
 CANADA                   |           1 |          1
 CHINA                    |           2 |          2
 ETHIOPIA                 |           0 |          1
 EGYPT                    |           4 |          2
 FRANCE                   |           3 |          1
 GERMANY                  |           3 |          1
 INDIA                    |           2 |          1
 INDONESIA                |           2 |          1
 IRAN                     |           4 |          2
 IRAQ                     |           4 |          2
 JAPAN                    |           2 |          1
 JORDAN                   |           4 |          2
 KENYA                    |           0 |          1
 MOROCCO                  |           0 |          1
 MOZAMBIQUE               |           0 |          1
 PERU                     |           1 |          1
 ROMANIA                  |           3 |          1
 RUSSIA                   |           3 |          1
 SAUDI ARABIA             |           4 |          1
 UNITED STATES            |           1 |          1
 UNITED KINGDOM           |           3 |          2
 VIETNAM                  |           2 |          1
(25 rows)
```

In this example window function „**dense_rank()**“ is used. It does the same thing as the „rank()“ window function, except how it reacts when there is a tie in a partition. In the example above we saw that after we had a tie the number which was assigned to the next row in the partition was not bigger by 1 than the value assigned to the rows which happen to have a tie. On the other hand, this is exactly the case when we use window function „dense_rank()“.
So, if we have six rows in a partition which happen to have a tie based on some criteria which is specified by the “ORDER BY” clause, then all of the six rows will be assigned number 1, while the rows which comes after these six lines in a same partition be assigned number 2 (unlike it would be the case if

```
SELECT n_name, n_regionkey,  row_number() OVER (
                                PARTITION BY substr(n_name, 1, 1)
                                ORDER BY n_regionkey)
FROM nation;
```

```
neumann=# select n_name, n_regionkey, row_number() over (partition by substr(n_name,1,1) order by n_regionkey) from nation;
          n_name           | n_regionkey | row_number
---------------------------+-------------+------------
 ALGERIA                   |           0 |          1
 ARGENTINA                 |           1 |          2
 BRAZIL                    |           1 |          1
 CANADA                    |           1 |          1
 CHINA                     |           2 |          2
 ETHIOPIA                  |           0 |          1
 EGYPT                     |           4 |          2
 FRANCE                    |           3 |          1
 GERMANY                   |           3 |          1
 INDIA                     |           2 |          1
 INDONESIA                 |           2 |          2
 IRAN                      |           4 |          3
 IRAQ                      |           4 |          4
 JAPAN                     |           2 |          1
 JORDAN                    |           4 |          2
 KENYA                     |           0 |          1
 MOROCCO                   |           0 |          1
 MOZAMBIQUE                |           0 |          2
 PERU                      |           1 |          1
 ROMANIA                   |           3 |          1
 RUSSIA                    |           3 |          2
 SAUDI ARABIA              |           4 |          1
 UNITED STATES             |           1 |          1
 UNITED KINGDOM            |           3 |          2
 VIETNAM                   |           2 |          1
(25 rows)
```

In this example window function „**row_number()**“ is used. Unlike both window functions “rank()” and “dense_rank()”, the “row_number()” window function assigns an **unique number to each row inside a partition , even if there is a tie among some rows**. Because it assign different numbers to rows which may have a tie (based on the condition specified in the “ORDER BY” clause), the “row_number()” function **not uniquely specified**. This window function can be very useful because the thing which is in SQL surprisingly difficult is to distinguish two rows which are conceptually different, but have the same value. It is not easy to separate them, unless you are using window function “row_number()”.

```
SELECT n_name, n_regionkey,  ntile(5) OVER (
                                PARTITION BY substr(n_name, 1, 1)
                                ORDER BY n_regionkey)
FROM nation;
```

```
neumann=# select n_name, n_regionkey, ntile(5) over (partition by substr(n_name,1,1) order by n_regionkey) from nation;
          n_name           | n_regionkey | ntile
---------------------------+-------------+-------
 ALGERIA                   |           0 |     1
 ARGENTINA                 |           1 |     2
 BRAZIL                    |           1 |     1
 CANADA                    |           1 |     1
 CHINA                     |           2 |     2
 ETHIOPIA                  |           0 |     1
 EGYPT                     |           4 |     2
 FRANCE                    |           3 |     1
 GERMANY                   |           3 |     1
 INDIA                     |           2 |     1
 INDONESIA                 |           2 |     2
 IRAN                      |           4 |     3
 IRAQ                      |           4 |     4
 JAPAN                     |           2 |     1
 JORDAN                    |           4 |     2
 KENYA                     |           0 |     1
 MOROCCO                   |           0 |     1
 MOZAMBIQUE                |           0 |     2
 PERU                      |           1 |     1
 ROMANIA                   |           3 |     1
 RUSSIA                    |           3 |     2
 SAUDI ARABIA              |           4 |     1
 UNITED STATES             |           1 |     1
 UNITED KINGDOM            |           3 |     2
 VIETNAM                   |           2 |     1
(25 rows)
```

In this example window function „**ntile(**integer_number**)**“ is used. This window function specifies that inside a single partition all the rows are going to be assigned numbers between 1 and the value which is specified in the window function.
So, in this example we used “ntile(5)”, which means that each row inside a single partition is going to be assigned a value between 1 and 5.
Unfortunately, this example does not have a partition which is big enough to

---

```
SELECT n_name, n_regionkey,  ntile(10) OVER (
                                PARTITION BY substr(n_name, 1, 1)
                                ORDER BY n_regionkey)
FROM nation;
```

```
neumann=# select n_name, n_regionkey, ntile(10) over (partition by substr(n_name,1,1) order by n_regionkey) from nation;
          n_name           | n_regionkey | ntile
---------------------------+-------------+-------
 ALGERIA                   |           0 |     1
 ARGENTINA                 |           1 |     2
 BRAZIL                    |           1 |     1
 CANADA                    |           1 |     1
 CHINA                     |           2 |     2
 ETHIOPIA                  |           0 |     1
 EGYPT                     |           4 |     2
 FRANCE                    |           3 |     1
 GERMANY                   |           3 |     1
 INDIA                     |           2 |     1
 INDONESIA                 |           2 |     2
 IRAN                      |           4 |     3
 IRAQ                      |           4 |     4
 JAPAN                     |           2 |     1
 JORDAN                    |           4 |     2
 KENYA                     |           0 |     1
 MOROCCO                   |           0 |     1
 MOZAMBIQUE                |           0 |     2
 PERU                      |           1 |     1
 ROMANIA                   |           3 |     1
 RUSSIA                    |           3 |     2
 SAUDI ARABIA              |           4 |     1
 UNITED STATES             |           1 |     1
 UNITED KINGDOM            |           3 |     2
 VIETNAM                   |           2 |     1
(25 rows)
```

Same as example above, with only difference that we just used „ntile(10)“ instead of „ntile(5)“.

```
SELECT n_name, n_regionkey,  ntile(10) OVER (
                                    ORDER BY n_regionkey)
FROM nation;
```

There is no „PARTITION BY“ clause, meaning that the program will look at all the rows as if the belonged one big partition.

PARTITION 1

```
neumann=# select n_name, n_regionkey, ntile(10) over (order by n_regionkey) from nation;
          n_name           | n_regionkey | ntile
---------------------------+-------------+-------
 ETHIOPIA                  |           0 |     1
 MOZAMBIQUE                |           0 |     1
 MOROCCO                   |           0 |     1
 KENYA                     |           0 |     2
 ALGERIA                   |           0 |     2
 UNITED STATES             |           1 |     2
 ARGENTINA                 |           1 |     3
 BRAZIL                    |           1 |     3
 CANADA                    |           1 |     3
 PERU                      |           1 |     4
 VIETNAM                   |           2 |     4
 INDONESIA                 |           2 |     4
 JAPAN                     |           2 |     5
 CHINA                     |           2 |     5
 INDIA                     |           2 |     5
 ROMANIA                   |           3 |     6
 RUSSIA                    |           3 |     6
 UNITED KINGDOM            |           3 |     7
 GERMANY                   |           3 |     7
 FRANCE                    |           3 |     8
 EGYPT                     |           4 |     8
 IRAQ                      |           4 |     9
 JORDAN                    |           4 |     9
 IRAN                      |           4 |    10
 SAUDI ARABIA              |           4 |    10
(25 rows)
```

In this example, as in previous two examples we use window function „ntile(number)“, except that we now have a case where we have a partition bigger than the number specified in the window function.
When we have a case as in this example where we have 25 wows, and in the window function “ntile(10)” we specified that all rows inside the same partition are going to be assigned values between 1 and 10, then we firstly need to see if we can distribute the values across the rows evenly. This can be concluded by seeing if the division of number of rows and the specified value in the window function gives out remainder zero. If that is not the case, like it is not in this example, then some rows will be grouped in bigger groups, while some will be grouped in smaller groups. In this example, number of rows which is 25 is not divisible by 10 (the value specified in the window function), the system will decide to divide it in 5 groups of 3, and 5 groups of 2.

```
SELECT n_name, n_regionkey,  percent_rank() OVER (
                                ORDER BY n_regionkey)
FROM nation;
```

```
neumann=# select n_name, n_regionkey, percent_rank() over (order by n_regionkey) from nation;
         n_name          | n_regionkey |    percent_rank
-------------------------+-------------+--------------------
 ETHIOPIA                |           0 |                  0
 MOZAMBIQUE              |           0 |                  0
 MOROCCO                 |           0 |                  0
 KENYA                   |           0 |                  0
 ALGERIA                 |           0 |                  0
 UNITED STATES           |           1 | 0.20833333333333334
 ARGENTINA               |           1 | 0.20833333333333334
 BRAZIL                  |           1 | 0.20833333333333334
 CANADA                  |           1 | 0.20833333333333334
 PERU                    |           1 | 0.20833333333333334
 VIETNAM                 |           2 |  0.4166666666666667
 INDONESIA               |           2 |  0.4166666666666667
 JAPAN                   |           2 |  0.4166666666666667
 CHINA                   |           2 |  0.4166666666666667
 INDIA                   |           2 |  0.4166666666666667
 ROMANIA                 |           3 |                0.625
 RUSSIA                  |           3 |                0.625
 UNITED KINGDOM          |           3 |                0.625
 GERMANY                 |           3 |                0.625
 FRANCE                  |           3 |                0.625
 EGYPT                   |           4 |  0.8333333333333334
 IRAQ                    |           4 |  0.8333333333333334
 JORDAN                  |           4 |  0.8333333333333334
 IRAN                    |           4 |  0.8333333333333334
 SAUDI ARABIA            |           4 |  0.8333333333333334
(25 rows)
```

$$\frac{6-1}{25-1} = \frac{5}{24}$$

Again, as there is no „PARTITION BY“ clause, all the rows belong to one big partition.
In this example we use window function “**percent_rank()**”. This window function outputs a value in “percent_rank” column according to the following formula:

$$percent\_rank = \frac{rank - 1}{total\_number\_or\_rows\ -\ 1}$$

where “rank” is the value which would be assigned to a row according to the “rank()” window function.
In this example some rows have the same value in the “percent_rank” column because the column we are ordering on, column “n_regionkey” has values which are not unique. This means that there would be ties if we performed window function “rank()”, which leads to same outputs in the “percent_rank” column, as values in this column depend on the output that window function “rank()” would output.

```
SELECT n_name, n_regionkey,  percent_rank() OVER (
                                ORDER BY n_nationkey)
FROM nation;
```

```
neumann=# select n_name, n_regionkey, percent_rank() over (order by n_nationkey) from nation;
         n_name          | n_regionkey |     percent_rank
-------------------------+-------------+---------------------
 ALGERIA                 |           0 |                   0
 ARGENTINA               |           1 | 0.041666666666666664
 BRAZIL                  |           1 |  0.08333333333333333
 CANADA                  |           1 |                0.125
 EGYPT                   |           4 |  0.16666666666666666
 ETHIOPIA                |           0 | 0.20833333333333334
 FRANCE                  |           3 |                0.25
 GERMANY                 |           3 |  0.2916666666666667
 INDIA                   |           2 |  0.3333333333333333
 INDONESIA               |           2 |               0.375
 IRAN                    |           4 |  0.4166666666666667
 IRAQ                    |           4 |  0.4583333333333333
 JAPAN                   |           2 |                 0.5
 JORDAN                  |           4 |  0.5416666666666666
 KENYA                   |           0 |  0.5833333333333334
 MOROCCO                 |           0 |               0.625
 MOZAMBIQUE              |           0 |  0.6666666666666666
 PERU                    |           1 |  0.7083333333333334
 CHINA                   |           2 |                0.75
 ROMANIA                 |           3 |  0.7916666666666666
 SAUDI ARABIA            |           4 |  0.8333333333333334
 VIETNAM                 |           2 |               0.875
 RUSSIA                  |           3 |  0.9166666666666666
 UNITED KINGDOM          |           3 |  0.9583333333333334
 UNITED STATES           |           1 |                   1
(25 rows)
```

In this example we are using the same widow function as in the example above, the „percent_rank()“ window function, except that we are using a different column according to which we perform ordering of rows inside a partition. This column is „n_nationkey“, and it differs from previously used column „n_regionkey“ in a sense that it contains only unique values. So, by having unique values, this column will lead to different ranks assigned to every row, and consequently different value in the column „percent_rank“ for each row.
The outputted columns in this example may be confusing, as the professor outputted column „n_regionkey“, instead column „“n_nationkey“, according to which the ordering was performed, so don’t confuse yourself by the output that the ordering was done according to the same column as in the previous example, but rather look at the written code.

```
SELECT n_name, n_regionkey,  cume_dist() OVER (
                                    ORDER BY n_regionkey)
FROM nation;
```

```
neumann=# select n_name, n_regionkey, cume_dist() over (order by n_regionkey) from nation;
         n_name          | n_regionkey | cume_dist
-------------------------+-------------+-----------
 ETHIOPIA                |           0 |       0.2
 MOZAMBIQUE              |           0 |       0.2
 MOROCCO                 |           0 |       0.2
 KENYA                   |           0 |       0.2
 ALGERIA                 |           0 |       0.2
 UNITED STATES           |           1 |       0.4
 ARGENTINA               |           1 |       0.4
 BRAZIL                  |           1 |       0.4
 CANADA                  |           1 |       0.4
 PERU                    |           1 |       0.4
 VIETNAM                 |           2 |       0.6
 INDONESIA               |           2 |       0.6
 JAPAN                   |           2 |       0.6
 CHINA                   |           2 |       0.6
 INDIA                   |           2 |       0.6
 ROMANIA                 |           3 |       0.8
 RUSSIA                  |           3 |       0.8
 UNITED KINGDOM          |           3 |       0.8
 GERMANY                 |           3 |       0.8
 FRANCE                  |           3 |       0.8
 EGYPT                   |           4 |         1
 IRAQ                    |           4 |         1
 JORDAN                  |           4 |         1
 IRAN                    |           4 |         1
 SAUDI ARABIA            |           4 |         1
(25 rows)
```

Again, as there is no „PARTITION BY“ clause, all the rows belong to one big partition.
In this example we use window function “**cume_dist()**”. This window function outputs a value in “cume_dist” column according to the following formula:

$$cume\_dist = \frac{number\_of\_preceeding\_rows\_in\_a\_partition}{total\_number\_of\_rows}$$

As when using window function “percent_rank()”, when we use window function “cume_dist()”, the value which would be assigned to a row when using window function “rank()”, plays a role, although in the formula we don’t specifically use the rank assigned to a row. This is the case in this example, where the ordering of rows inside a partition is done based on a column which has non-unique values, which leads to multiple rows being assigned same rank.
So, for an example, for the first row we would not calculate the value in the “cume_dist” column by 1/25, but rather 5/25, as there are 5 rows which would be assigned the same rank.

```
SELECT o_custkey, o_orderdate, o_totalprice – lag(o_totalprice, 1) OVER (
                                                PARTITION BY o_custkey
                                                ORDER BY o_orderdate)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, o_totalprice, o_totalprice - lag(o_totalprice, 1) over (partition by o_custkey order by o_orderdate) f
rom orders;
```

o_custkey	o_orderdate	o_totalprice	?column?
1	1992-04-19	74602.81	NULL
1	1992-08-22	123076.84	48474.03
1	1996-06-29	65478.05	-57598.79
1	1996-07-01	174645.94	109167.89
1	1996-12-09	54048.26	-120597.68
1	1997-03-23	95911.01	41862.75
2	1992-04-05	167016.61	NULL
2	1994-05-21	103297.68	-63718.93
2	1994-08-28	16495.33	-86802.35
2	1994-12-24	33082.83	16587.50
2	1995-03-10	221397.35	188314.52
2	1996-08-05	174291.41	-47105.94
2	1997-02-22	312692.22	138400.81
4	1992-04-26	311722.87	NULL
4	1992-09-20	182956.01	-128766.86
4	1993-10-04	88317.19	-94638.82
4	1994-06-10	40347.48	-47969.71
4	1995-05-06	104934.65	64587.17
4	1995-11-01	277493.04	172558.39
4	1996-01-03	46227.94	-231265.10
4	1996-01-06	225294.26	179066.32
4	1996-06-03	143971.54	-81322.72
4	1996-06-06	235621.83	91650.29
4	1996-08-02	314671.82	79049.99
4	1996-08-11	8332.33	-306339.49
4	1996-11-29	152929.80	144597.47
4	1997-03-07	75928.49	-77001.31
4	1997-06-12	1131.20	-74797.29
4	1997-07-12	17938.41	16807.21
4	1997-09-07	71845.26	53906.85
4	1997-10-03	91152.97	19307.71
4	1997-11-23	76100.95	-15052.02
4	1998-05-16	181618.75	105517.80
5	1993-06-27	324835.83	NULL

As there is no preceding value to the first row of a partition, window function „lag()“ will try to address a row which comes before it which doesn't exist, which is why we will get "NULL" as output (if we use the "lag()" window function to look at just one row behind).

Window function "**lag(**column**,** integer_number**,** double_number**)**" is used to address values from the previous rows of the column specified in the function. How far back (meaning how far a row inside the same partition, which comes before this row, influences the current row) depends on the second argument of the function. The third argument of the window function is optional, and it specifies the default value which is going to be used if there are not enough preceding rows in a partition in order to determine the output of the window function "lag". For an example, if we used zero as the third argument of the "lag" window function, then in the first rows of a partition we would not have "NULL" as output, but rather the same value as the row has in the "o_totalprice" column.

```
SELECT o_custkey, o_orderdate, o_totalprice, o_totalprice – lag(o_totalprice, 1, 0.0) OVER (
                                                    PARTITION BY o_custkey
                                                    ORDER BY o_orderdate)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, o_totalprice, o_totalprice - lag(o_totalprice, 1, 0.0) over (partition by o_custkey order by o_orderda
te) from orders;
```

o_custkey	o_orderdate	o_totalprice	?column?
1	1992-04-19	74602.81	74602.81
1	1992-08-22	123076.84	48474.03
1	1996-06-29	65478.05	-57598.79
1	1996-07-01	174645.94	109167.89
1	1996-12-09	54048.26	-120597.68
1	1997-03-23	95911.01	41862.75
2	1992-04-05	167016.61	167016.61
2	1994-05-21	103297.68	-63718.93
2	1994-08-28	16495.33	-86802.35
2	1994-12-24	33082.83	16587.50
2	1995-03-10	221397.35	188314.52
2	1996-08-05	174291.41	-47105.94
2	1997-02-22	312692.22	138400.81
4	1992-04-26	311722.87	311722.87
4	1992-09-20	182956.01	-128766.86
4	1993-10-04	88317.19	-94638.82
4	1994-06-10	40347.48	-47969.71
4	1995-05-06	104934.65	64587.17
4	1995-11-01	277493.04	172558.39
4	1996-01-03	46227.94	-231265.10
4	1996-01-06	225294.26	179066.32
4	1996-06-03	143971.54	-81322.72
4	1996-06-06	235621.83	91650.29
4	1996-08-02	314671.82	79049.99
4	1996-08-11	8332.33	-306339.49
4	1996-11-29	152929.80	144597.47
4	1997-03-07	75928.49	-77001.31
4	1997-06-12	1131.20	-74797.29
4	1997-07-12	17938.41	16807.21
4	1997-09-07	71845.26	53906.85
4	1997-10-03	91152.97	19307.71
4	1997-11-23	76100.95	-15052.02
4	1998-05-16	181618.75	105517.80
5	1993-06-27	324835.83	324835.83

The error you get if you tried to put „0“ instead of „0.0“ in the „lag“ function as third parameter:

```
neumann=# select o_custkey, o_orderdate, o_totalprice, o_totalprice - lag(o_totalprice, 1, 0) over (partition by o_custkey order by o_orderdate
) from orders;
ERROR:  function lag(numeric, integer, integer) does not exist
LINE 1: ...ustkey, o_orderdate, o_totalprice, o_totalprice - lag(o_tota...
```

1.3 lecture 12 (FDE), examples, window functions

```
SELECT o_custkey, o_orderdate, avg(o_totalprice) OVER (
                                    PARTITION BY o_custkey
                                    ORDER BY o_orderdate
                                    ROWS BETWEEN 1 preceding AND 1 following)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, avg(o_totalprice) over (partition by o_custkey order by o_orderdate rows between 1 preceding and 1 follo
wing) from orders;
```

o_custkey	o_orderdate	avg
1	1992-04-19	98839.825000000000
1	1992-08-22	87719.233333333333
1	1996-06-29	121066.943333333333
1	1996-07-01	98057.416666666667
1	1996-12-09	108201.736666666667
1	1997-03-23	74979.635000000000
2	1992-04-05	135157.145000000000
2	1994-05-21	95603.206666666667
2	1994-08-28	50958.613333333333
2	1994-12-24	90325.170000000000
2	1995-03-10	142923.863333333333
2	1996-08-05	236126.993333333333
2	1997-02-22	243491.815000000000
4	1992-04-26	247339.440000000000
4	1992-09-20	194332.023333333333
4	1993-10-04	103873.560000000000
4	1994-06-10	77866.440000000000
4	1995-05-06	140925.056666666667
4	1995-11-01	142885.210000000000
4	1996-01-03	183005.080000000000
4	1996-01-06	138497.913333333333
4	1996-06-03	201629.210000000000
4	1996-06-06	231421.730000000000
4	1996-08-02	186208.660000000000
4	1996-08-11	158644.650000000000
4	1996-11-29	79063.540000000000
4	1997-03-07	76663.163333333333
4	1997-06-12	31666.033333333333
4	1997-07-12	30304.956666666667
4	1997-09-07	60312.213333333333
4	1997-10-03	79699.726666666667
4	1997-11-23	116290.890000000000
4	1998-05-16	128859.850000000000
5	1993-06-27	293087.245000000000

Computing the average among:

1. value from the previous row,
2. value from the current row,
3. value from the next row

```
SELECT o_custkey, o_orderdate, o_totalprice, avg(o_totalprice) OVER (
                                                    PARTITION BY o_custkey
                                                    ORDER BY o_orderdate
                                                    ROWS BETWEEN 1 preceding AND 1 following)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, o_totalprice, avg(o_totalprice) over (partition by o_custkey order by o_orderdate rows between 1 precedi
ng and 1 following) from orders;
```

o_custkey	o_orderdate	o_totalprice	avg
1	1992-04-19	74602.81	98839.825000000000
1	1992-08-22	123076.84	87719.233333333333
1	1996-06-29	65478.05	121066.943333333333
1	1996-07-01	174645.94	98057.416666666667
1	1996-12-09	54048.26	108201.736666666667
1	1997-03-23	95911.01	74979.635000000000
2	1992-04-05	167016.61	135157.145000000000
2	1994-05-21	103297.68	95603.206666666667
2	1994-08-28	16495.33	50958.613333333333
2	1994-12-24	33082.83	90325.170000000000
2	1995-03-10	221397.35	142923.863333333333
2	1996-08-05	174291.41	236126.993333333333
2	1997-02-22	312692.22	243491.815000000000
4	1992-04-26	311722.87	247339.440000000000
4	1992-09-20	182956.01	194332.023333333333
4	1993-10-04	88317.19	103873.560000000000
4	1994-06-10	40347.48	77866.440000000000
4	1995-05-06	104934.65	140925.056666666667
4	1995-11-01	277493.04	142885.210000000000
4	1996-01-03	46227.94	183005.080000000000
4	1996-01-06	225294.26	138497.913333333333
4	1996-06-03	143971.54	201629.210000000000
4	1996-06-06	235621.83	231421.730000000000
4	1996-08-02	314671.82	186208.660000000000
4	1996-08-11	8332.33	158644.650000000000
4	1996-11-29	152929.80	79063.540000000000
4	1997-03-07	75928.49	76663.163333333333
4	1997-06-12	1131.20	31666.033333333333
4	1997-07-12	17938.41	30304.956666666667
4	1997-09-07	71845.26	60312.213333333333
4	1997-10-03	91152.97	79699.726666666667
4	1997-11-23	76100.95	116290.890000000000
4	1998-05-16	181618.75	128859.850000000000
5	1993-06-27	324835.83	293087.245000000000

Same as previous except column „o_totalprice“ was added to the output to see better the average being computed.

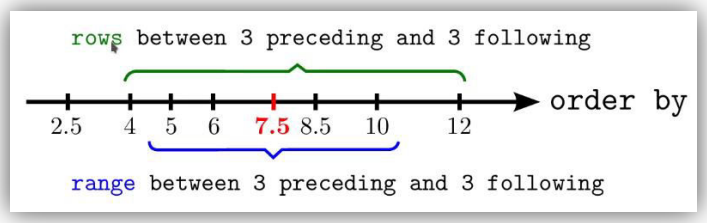
The result in the first row of each partition in the „avg“ column is not average of a sum of three values, but an average of two values, as there is no preceding value in this case ((74602.81+123079.65)/2 = 98839.825, for the first partition, first row), and as well the last value in a partition in the „avg“ column is an average of two values, the current value and the preceding, as there is no following number ((95911.01+5404826)/2 = 74979.635, for the last row in the first partition). All the values in between are an average of three values.

```
SELECT o_custkey, o_orderdate, avg(o_totalprice) OVER (
                                    PARTITION BY o_custkey
                                    ORDER BY o_orderdate
                                    ROWS BETWEEN 2 preceding AND 2 following)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, o_totalprice, avg(o_totalprice) over (partition by o_custkey order by o_orderdate rows between 2 precedi
ng and 2 following) from orders;
```

o_custkey	o_orderdate	o_totalprice	avg
1	1992-04-19	74602.81	87719.233333333333
1	1992-08-22	123076.84	109450.910000000000
1	1996-06-29	65478.05	98370.380000000000
1	1996-07-01	174645.94	102632.020000000000
1	1996-12-09	54048.26	97520.815000000000
1	1997-03-23	95911.01	108201.736666666667
2	1992-04-05	167016.61	95603.206666666667
2	1994-05-21	103297.68	79973.112500000000
2	1994-08-28	16495.33	108257.960000000000
2	1994-12-24	33082.83	109712.920000000000
2	1995-03-10	221397.35	151591.828000000000
2	1996-08-05	174291.41	185365.952500000000
2	1997-02-22	312692.22	236126.993333333333
4	1992-04-26	311722.87	194332.023333333333
4	1992-09-20	182956.01	155835.887500000000
4	1993-10-04	88317.19	145655.640000000000
4	1994-06-10	40347.48	138809.674000000000
4	1995-05-06	104934.65	111464.060000000000
4	1995-11-01	277493.04	138859.474000000000
4	1996-01-03	46227.94	159584.286000000000
4	1996-01-06	225294.26	185721.722000000000
4	1996-06-03	143971.54	193157.478000000000
4	1996-06-06	235621.83	185578.356000000000
4	1996-08-02	314671.82	171105.464000000000
4	1996-08-11	8332.33	157496.854000000000
4	1996-11-29	152929.80	110598.728000000000
4	1997-03-07	75928.49	51252.046000000000
4	1997-06-12	1131.20	63954.632000000000
4	1997-07-12	17938.41	51599.266000000000
4	1997-09-07	71845.26	51633.758000000000
4	1997-10-03	91152.97	87731.268000000000
4	1997-11-23	76100.95	105179.482500000000
4	1998-05-16	181618.75	116290.890000000000
5	1993-06-27	324835.83	223781.723333333333

Notice the use of phrase „ROWS“ (not „RANGE“) in the framing clause of the window function, as these two have a different meaning, which can be seen from the photo:



```
SELECT o_custkey, o_orderdate, o_totalprice, sum(o_totalprice) OVER (
                                                    PARTITION BY o_custkey
                                                    ORDER BY o_orderdate)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, o_totalprice, sum(o_totalprice) over (partition by o_custkey order by o_orderdate) from orders;
```

o_custkey	o_orderdate	o_totalprice	sum
1	1992-04-19	74602.81	74602.81
1	1992-08-22	123076.84	197679.65
1	1996-06-29	65478.05	263157.70
1	1996-07-01	174645.94	437803.64
1	1996-12-09	54048.26	491851.90
1	1997-03-23	95911.01	587762.91
2	1992-04-05	167016.61	167016.61
2	1994-05-21	103297.68	270314.29
2	1994-08-28	16495.33	286809.62
2	1994-12-24	33082.83	319892.45
2	1995-03-10	221397.35	541289.80
2	1996-08-05	174291.41	715581.21
2	1997-02-22	312692.22	1028273.43
4	1992-04-26	311722.87	311722.87
4	1992-09-20	182956.01	494678.88
4	1993-10-04	88317.19	582996.07
4	1994-06-10	40347.48	623343.55
4	1995-05-06	104934.65	728278.20
4	1995-11-01	277493.04	1005771.24
4	1996-01-03	46227.94	1051999.18
4	1996-01-06	225294.26	1277293.44
4	1996-06-03	143971.54	1421264.98
4	1996-06-06	235621.83	1656886.81
4	1996-08-02	314671.82	1971558.63
4	1996-08-11	8332.33	1979890.96
4	1996-11-29	152929.80	2132820.76
4	1997-03-07	75928.49	2208749.25
4	1997-06-12	1131.20	2209880.45
4	1997-07-12	17938.41	2227818.86
4	1997-09-07	71845.26	2299664.12
4	1997-10-03	91152.97	2390817.09
4	1997-11-23	76100.95	2466918.04
4	1998-05-16	181618.75	2648536.79
5	1993-06-27	324835.83	324835.83

No „framing“ clause

```
SELECT o_custkey, o_orderdate, o_totalprice, sum(o_totalprice) OVER (
                                                    PARTITION BY o_custkey)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, o_totalprice, sum(o_totalprice) over (partition by o_custkey) from orders;
```

o_custkey	o_orderdate	o_totalprice	sum
1	1996-06-29	65478.05	587762.91
1	1996-07-01	174645.94	587762.91
1	1997-03-23	95911.01	587762.91
1	1992-08-22	123076.84	587762.91
1	1992-04-19	74602.81	587762.91
1	1996-12-09	54048.26	587762.91
2	1994-12-24	33082.83	1028273.43
2	1992-04-05	167016.61	1028273.43
2	1995-03-10	221397.35	1028273.43
2	1997-02-22	312692.22	1028273.43
2	1994-05-21	103297.68	1028273.43
2	1996-08-05	174291.41	1028273.43
2	1994-08-28	16495.33	1028273.43
4	1997-06-12	1131.20	2648536.79
4	1996-11-29	152929.80	2648536.79
4	1996-01-03	46227.94	2648536.79
4	1997-03-07	75928.49	2648536.79
4	1994-06-10	40347.48	2648536.79
4	1996-08-11	8332.33	2648536.79
4	1996-01-06	225294.26	2648536.79
4	1996-06-06	235621.83	2648536.79
4	1996-08-02	314671.82	2648536.79
4	1995-05-06	104934.65	2648536.79
4	1998-05-16	181618.75	2648536.79
4	1995-11-01	277493.04	2648536.79
4	1993-10-04	88317.19	2648536.79
4	1996-06-03	143971.54	2648536.79
4	1992-04-26	311722.87	2648536.79
4	1997-11-23	76100.95	2648536.79
4	1997-10-03	91152.97	2648536.79
4	1997-07-12	17938.41	2648536.79
4	1992-09-20	182956.01	2648536.79
4	1997-09-07	71845.26	2648536.79
5	1993-06-27	324835.83	684965.28

No „ORDER BY“ clause, and no „FRAMING“ clause („ROWS“ or „RANGE“).

```
SELECT o_custkey, o_orderdate, o_totalprice, sum(o_totalprice) OVER (
                                                PARTITION BY o_custkey
                                                ROWS BETWEEN 1 preceding AND 1 following)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, o_totalprice, sum(o_totalprice) over (partition by o_custkey rows between 1 preceding and 1 following) f
rom orders;
```

o_custkey	o_orderdate	o_totalprice	sum
1	1996-06-29	65478.05	240123.99
1	1996-07-01	174645.94	336035.00
1	1997-03-23	95911.01	393633.79
1	1992-08-22	123076.84	293590.66
1	1992-04-19	74602.81	251727.91
1	1996-12-09	54048.26	128651.07
2	1994-12-24	33082.83	200099.44
2	1992-04-05	167016.61	421496.79
2	1995-03-10	221397.35	701106.18
2	1997-02-22	312692.22	637387.25
2	1994-05-21	103297.68	590281.31
2	1996-08-05	174291.41	294084.42
2	1994-08-28	16495.33	190786.74
4	1997-06-12	1131.20	154061.00
4	1996-11-29	152929.80	200288.94
4	1996-01-03	46227.94	275086.23
4	1997-03-07	75928.49	162503.91
4	1994-06-10	40347.48	124608.30
4	1996-08-11	8332.33	273974.07
4	1996-01-06	225294.26	469248.42
4	1996-06-06	235621.83	775587.91
4	1996-08-02	314671.82	655228.30
4	1995-05-06	104934.65	601225.22
4	1998-05-16	181618.75	564046.44
4	1995-11-01	277493.04	547428.98
4	1993-10-04	88317.19	509781.77
4	1996-06-03	143971.54	544011.60
4	1992-04-26	311722.87	531795.36
4	1997-11-23	76100.95	478976.79
4	1997-10-03	91152.97	185192.33
4	1997-07-12	17938.41	292047.39
4	1992-09-20	182956.01	272739.68
4	1997-09-07	71845.26	254801.27
5	1993-06-27	324835.83	410006.51

No „ORDER BY“ clause, but there is a „FRAMING“ clause („ROWS“ or „RANGE“).

This doesn't make sense, and professors says that in his opinion this should display an error message, as it is not clear what the output of such a window function should be, as there is no specified order (not sorted rows within a partition) according to which the aggregate function of the window function should be applied. It is totally unclear in which order are we going to get output of rows, if „ORDER BY“ clause is not specified. PostgeSQL by default does not output an error (as you can see from the example above), but avoid such constructions of window functions.

```
neumann=# select distinct o_shippriority from orders;
 o_shippriority
----------------
              0
(1 row)

neumann=# select distinct o_orderstatus from orders;
 o_orderstatus
---------------
 P
 O
 F
(3 rows)
```

```
SELECT o_custkey, o_orderdate, o_totalprice, o_orderstatus, sum(o_totalprice) OVER (
                                                    PARTITION BY o_custkey
                                                    ORDER BY o_orderdate
                                                    ROWS BETWEEN
                                                        CASE WHEN o_orderstatus='F'
                                                        THEN 3
                                                        ELSE 1
                                                        END
                                                    preceding AND
                                                        CASE WHEN o_orderstatus='F'
                                                        THEN 3
                                                        ELSE 1
                                                        END
                                                    following)
FROM orders;
```

```
neumann=# select o_custkey, o_orderdate, o_totalprice, o_orderstatus, sum(o_totalprice) over (partition by o_custkey order by o_orderdate rows be
tween case when o_orderstatus='F' then 3 else 1 end preceding and case when o_orderstatus='F' then 3 else 1 end following) from orders;
ERROR:  argument of ROWS must not contain variables
LINE 1: ...stkey order by o_orderdate rows between case when o_ordersta...
                                                             ^
```

Enter a SQL query against a scale-factor 1 TPC-H or the Uni database and retrieve the result set or show the optimized query plan:

```
1  select o_custkey, o_orderdate, o_totalprice, o_orderstatus, sum(o_totalprice) over (partition by o_custkey order by o_orderdate
   rows between case when o_orderstatus='F' then 3 else 1 end preceding and case when o_orderstatus='F' then 3 else 1 end following)
   from orders;
```

(works on HyPer[1])

Query Result

Compilation time: 54.9041 ms
Execution time: 286.114 ms
Result set size: 1500000

Warning! The result set you requested is larger than 1,000 tuples. To prevent web browser crashes, we send at most 1,000 tuples to the client web browser.

Showing 1 to 10 of 1,000 rows Search:

o_custkey	o_orderdate	o_totalprice	o_orderstatus	sum
338	1992-12-08	113366.48	F	371766.28
338	1993-01-18	126340.06	F	639560.53
338	1993-01-26	73893.52	F	820710.13
338	1993-06-05	58166.22	F	878215.30
338	1993-09-19	267794.25	F	818519.72
338	1995-07-22	181149.60	O	506449.02
338	1997-06-17	57505.17	O	292325.67
338	1997-12-12	53670.90	O	222321.43
338	1998-06-08	111145.36	O	301057.42
338	1998-07-30	136241.16	O	247386.52

Previous 1 2 3 4 5 ... 100 Next

PostgreSQL refuses to do complex frames, meaning expressions in the „framing“ clause where there is not a specified number, but some kind of expression which depend on the values in from the tables like in this example, as execution of such queries may lead to $O(n^2)$ complexity. On the other hand, Oracle executes such queries, but they have $O(n^2)$ complexity, which is unacceptable, as we work with large databases. There is a way that such queries can be executed in O(log(n)) complexity, and that is by using **segment trees**. These are trees which are constructed so that aggregate functions from the window-functions can be performed on arbitrary ranges (each output may have a different range).

[1] http://www.hyper-db.de/interface.html#

```
SELECT o_custkey, o_orderdate, sum(o_totalprice) OVER (
                                ORDER BY o_orderdate)
FROM orders;
```

```
neumann=# select o_orderdate, sum(o_totalprice) over (order by o_orderdate) from orders;
```

```
 o_orderdate |       sum
-------------+-----------------
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
 1992-01-01  |     92959447.96
```

```
SELECT o_orderdate, sum(o_totalprice) OVER (
                        ORDER BY o_orderdate)
FROM (SELECT o_orderdate, sum(o_totalprice) AS o_totalprice
      FROM orders
      GROUP BY o_orderdate) AS s
```

```
neumann=# select o_orderdate, sum(o_totalprice) over (order by o_orderdate) from (select o_orderdate, sum(o_totalprice) as o_totalprice from orde
rs group by o_orderdat;
```

o_orderdate	sum
1992-01-01	92959447.96
1992-01-02	183382958.80
1992-01-03	273442665.61
1992-01-04	372511155.40
1992-01-05	472653747.27
1992-01-06	569702309.32
1992-01-07	664397458.59
1992-01-08	756567096.95
1992-01-09	853123678.06
1992-01-10	945732033.37
1992-01-11	1041122903.58
1992-01-12	1137368775.79
1992-01-13	1226559447.38
1992-01-14	1322945252.98
1992-01-15	1413046548.54
1992-01-16	1506841127.49
1992-01-17	1599099588.61
1992-01-18	1692692397.29
1992-01-19	1786952958.43
1992-01-20	1876297816.76
1992-01-21	1974869725.94
1992-01-22	2069200707.14
1992-01-23	2161952567.58
1992-01-24	2257525107.37
1992-01-25	2353857993.11
1992-01-26	2445159796.79
1992-01-27	2540380061.29
1992-01-28	2638571578.23
1992-01-29	2727254641.60
1992-01-30	2820015529.78
1992-01-31	2924475612.31
1992-02-01	3021792141.99
1992-02-02	3115270560.13
1992-02-03	3211733691.51

Exercise from slide 31, FDE, section „Using a Database“:

- For each customer from GERMANY compute the cumulative spending (“sum(o_totalprice)”) by year (“extract(year from o_orderdate)”)?

```
SELECT o_custkey, year, sum(revenue) OVER (
                                    PARTITION BY o_custkey
                                    ORDER BY year)
FROM ( SELECT o_custkey, year, sum(o_totalprice) AS revenue
       FROM ( SELECT o_custkey, extract(year FROM o_orderdate) AS year, o_totalprice
              FROM orders) AS s
       GROUP BY o_custkey, year) AS s
```

```
neumann=# select o_custkey, year, sum(revenue) over (partition by o_custkey order by year) from (select o_custkey, year, sum(o_totalprice) as rev
enue from (select o_custkey, extract(year from o_orderdate) as year, o_totalprice from orders) s group by o_custkey, year) s;
```

o_custkey	year	sum
1	1992	197679.65
1	1996	491851.90
1	1997	587762.91
2	1992	167016.61
2	1994	319892.45
2	1995	541289.80
2	1996	715581.21
2	1997	1028273.43
4	1992	494678.88
4	1993	582996.07
4	1994	623343.55
4	1995	1005771.24
4	1996	2132820.76
4	1997	2466918.04
4	1998	2648536.79
5	1993	324835.83
5	1995	586174.49
5	1996	684965.28
7	1992	757256.81
7	1993	1339738.76
7	1994	1708660.89
7	1995	2227166.77
7	1996	2306271.23
7	1997	2639637.70
7	1998	2957861.16
8	1992	105225.10
8	1994	366327.85
8	1995	775713.90
8	1996	954249.00
8	1997	1116270.81
8	1998	1634983.70
10	1992	557777.71
10	1993	622598.35
10	1995	1306305.98

1.3.1 WITH statement

In this example we will introduce the **"WITH" clause**. It is a clause which has the following syntax:

```
WITH q1 AS (    SELECT ...
                FROM ...),
     q2 AS (    SELECT ...
                FROM q1, ....)
SELECT ...
FROM ...
```

From this we can see that **a single "WITH" clause can introduce multiple queries by separating them with a comma** (i.e. the "WITH" word is not repeated). On the other hand, when there is only one query in a "WITH" clause, there is no comma, nor any character after it, as well at **there is no comma, nor any other character after the last query specified in a "WITH" clause when there is more than one query specified in it**.

Also, we can see from the example above that when we specify multiple queries inside a single "WITH" statement, inside one of the queries we are writing we can refer to any query which is defined above the current one (like in the example above where in query "q2" we refer to the query defined above it in the same "WITH" clause, i.e. query "q1").

Also, **a query from "WITH" can be referred at multiple times in one query**, and in that case it is executed only once, and its result is reused (i.e. it will not be executed multiple times).

The purpose of use of "WITH" clause is **breaking a complex query into smaller parts**.s

```
WITH years AS ( SELECT DISTINCT extract(year FROM o_orderdate) AS year
                FROM orders)
SELECT o_custkey, year, sum(revenue) OVER (
                        PARTITION BY o_custkey
                        ORDER BY year)
FROM ( SELECT o_custkey, year, sum(o_totalprice) AS revenue
       FROM ( SELECT o_custkey, extract(year FROM o_orderdate) AS year, o_totalprice
              FROM orders) AS s
       GROUP BY o_custkey, year) AS s
```

```
neumann=# with years as (select distinct extract(year from o_orderdate) as year from orders) select o_custkey, year, sum(revenue) over (partition
 by o_custkey order by year) from (select o_custkey, year, sum(o_totalprice) as revenue from (select o_custkey, extract(year from o_orderdate) as
_year, o_totalprice from orders union all select c_custkey, year, 0 from customer, years) s group by o_custkey, year) s;
```

o_custkey	year	sum
1	1992	197679.65
1	1993	197679.65
1	1994	197679.65
1	1995	197679.65
1	1996	491851.90
1	1997	587762.91
1	1998	587762.91
2	1992	167016.61
2	1993	167016.61
2	1994	319892.45
2	1995	541289.80
2	1996	715581.21
2	1997	1028273.43
2	1998	1028273.43
3	1992	0
3	1993	0
3	1994	0
3	1995	0
3	1996	0
3	1997	0
3	1998	0
4	1992	494678.88
4	1993	582996.07
4	1994	623343.55
4	1995	1005771.24
4	1996	2132820.76
4	1997	2466918.04
4	1998	2648536.79
5	1992	0
5	1993	324835.83
5	1994	324835.83
5	1995	586174.49
5	1996	684965.28
5	1997	684965.28

You get an entry for each customer in each year, even if they did not make any orders in a year (with help of a dummy variable we added for each year which adds an order of price 0 for each year).

In the case we use „sum“ as the aggregate function, the situation where there is no entries (purchases in this case) for certain rows, it doesn't make a difference if we add dummy entries or not, but in the case we used „avg“ as an aggregate function and refer to rows which are preceding and following to the current row, it important that we have these dummy rows, as they influence the result. An example of this is given on the next two pages.

What would happen if we did not use the version with dummy rows, and instead of calculating the sum like in the previous example we calculated the average, using the „avg“ function:

```
SELECT o_custkey, year, avg(revnue) OVER (
                        PARTITION BY o_custkey
                        ORDER BY year
                        ROWS BETWEEN 1 preceding AND 1 following)
FROM ( SELECT o_custkey, year, sum(o_totalprice) AS revenue
       FROM ( SELECT o_custkey, extract(year FROM o_orderdate) AS year, o_totalprice
              FROM orders) AS s
       GROUP BY o_custkey, year) AS s
```

```
neumann=# select o_custkey, year, avg(revenue) over (partition by o_custkey order by year rows between 1 preceding and 1 following  ) from (selec
t o_custkey, year, sum(o_totalprice) as revenue from (select o_custkey, extract(year from o_orderdate) as year, o_totalprice from orders) s group
 by o_custkey, year) s;
```

o_custkey	year	avg
1	1992	245925.950000000000
1	1996	195920.970000000000
1	1997	195041.630000000000
2	1992	159946.225000000000
2	1994	180429.933333333333
2	1995	182854.866666666667
2	1996	236126.993333333333
2	1997	243491.815000000000
4	1992	291498.035000000000
4	1993	207781.183333333333
4	1994	170364.120000000000
4	1995	516608.230000000000
4	1996	614524.830000000000
4	1997	547588.516666666667
4	1998	257858.015000000000
5	1993	293087.245000000000
5	1995	228321.760000000000
5	1996	180064.725000000000
7	1992	669869.380000000000
7	1993	569553.630000000000
7	1994	489969.986666666667
7	1995	322177.490000000000
7	1996	310325.603333333333
7	1997	243564.796666666667
7	1998	325794.965000000000
8	1992	183163.925000000000
8	1994	258571.300000000000
8	1995	283007.966666666667
8	1996	249980.986666666667
8	1997	286423.266666666667
8	1998	340367.350000000000
10	1992	311299.175000000000
10	1993	435435.326666666667
10	1995	620136.416666666667

In this case average of value in the row of first partition where there is „1992“ in the column „year“, the average value in the „avg“ column is computed as average of values in 1992 (current) and 1996 (following, at least that is how the program sees that), which should not be the case as following to 1992 value should be 1993 value, but the customer to which the first partition refers to did not make any purchases in 1993, 1994, 1995 (so dummy rows should be added, for all years, and then for the missing ones they will be displayed as zero, while these dummy rows will have no effect on rows for which some purchases were made).

```
WITH years AS ( SELECT DISTINCT extract(year FROM o_orderdate) AS year
                FROM orders)
SELECT o_custkey, year, avg(revnue) OVER (
                        PARTITION BY o_custkey
                        ORDER BY year
                        ROWS BETWEEN 1 preceing AND 1 following)
FROM ( SELECT o_custkey, year, sum(o_totalprice) AS revenue
       FROM ( SELECT o_custkey, extract(year FROM o_orderdate) AS year, o_totalprice
              FROM orders
              UNION ALL
              SELECT c_custkey, year, 0
              FROM customer, years) AS s
       GROUP BY o_custkey, year) AS s
```

```
neumann=# with years as (select distinct extract(year from o_orderdate) as year from orders) select o_custkey, year, avg(revenue) over (partition
 by o_custkey order by year rows between 1 preceding and 1 following) from (select o_custkey, year, sum(o_totalprice) as revenue from (select o_c
ustkey, extract(year from o_orderdate) as year, o_totalprice from orders union all select c_custkey, year, 0 from customer, years) s group by o_c
ustkey, year) s;
```

o_custkey	year	avg
1	1992	98839.825000000000
1	1993	65893.216666666667
1	1994	0.00000000000000000000
1	1995	98057.416666666667
1	1996	130027.753333333333
1	1997	130027.753333333333
1	1998	47955.505000000000
2	1992	83508.305000000000
2	1993	106630.816666666667
2	1994	124757.730000000000
2	1995	182854.866666666667
2	1996	236126.993333333333
2	1997	162327.876666666667
2	1998	156346.110000000000
3	1992	0.00000000000000000000
3	1993	0.00000000000000000000
3	1994	0.00000000000000000000
3	1995	0.00000000000000000000
3	1996	0.00000000000000000000
3	1997	0.00000000000000000000
3	1998	0.00000000000000000000
4	1992	291498.035000000000
4	1993	207781.183333333333
4	1994	170364.120000000000
4	1995	516608.230000000000
4	1996	614524.830000000000
4	1997	547588.516666666667
4	1998	257858.015000000000
5	1992	162417.915000000000
5	1993	108278.610000000000
5	1994	195391.496666666667
5	1995	120043.150000000000
5	1996	120043.150000000000
5	1997	32930.263333333333

In contrast to the previous example, in this one we added dummy rows and got a different (the output which would be most likely be wanted instead of the output from the previous example).

2. ORDERED-SET FUNCTIONS

2.1 Introduction to ordered-set functions

Ordered-set Functions are functions which require that a table on which these functions are applied on is sorted.

Syntax for ordered-set functions:

```
SELECT columns_which_will_be_outputted, orderd_set_function() WITHIN GROUP(
                                                    ORDERED BY column)
FROM table
GROUP BY column;
```

Firstly, instead of columns_which_will_be_outputted we write some column names from the table. Secondly, instead of orderd_set_function() we write one ordered-set function, some of which are:

1) mode()
2) percentile_disc(p)
3) percentile_cont(p)

In both ordered-set functions "percentile_disc(p)" and "p ercentile_cont(p)", the p is short for **"probability**", and probability can only be in a value which belongs to range [0, 1].

2.2 lecture 12 (FDE), examples, ordered-set functions

```
neumann=# select count(*) from orders;
  count
---------
 1500000
(1 row)
```

We see that there is an even number of rows.

If we use order-set function „**percentile_cont(p)**“, and we as well have an even number of rows, it is not really clear which value should be taken as the **median value** (middle value), so in case of this ordered-set function interpolation is used to determine

```
neumann=# select percentile_cont(0.5) within group (order by o_totalprice) from orders;
```

```
 percentile_cont
--------------------
 144409.03999999998
(1 row)
```

row[15000/2] value is = 144409.02
row[(15000/2) +1] value is = 144409.06
Interpolation:
(144409.02 + 144409.06) / 2 = 144409.039999998

If we use order-set function „**percentile_disc(p)**“ instead of „percentile_cont(p)“, like above, we get the first value of two if there is an even number of rows.

```
neumann=# select percentile_disc(0.5) within group (order by o_totalprice) from orders;
 percentile_disc
-----------------
       144409.02
(1 row)
```

To demonstrate what is done by ordered-set functions, we will use window-functions to display the “15000/2”-th row, and “(15000/2) +1”-th row:

```
neumann=# select * from (select o_totalprice, row_number() over (order by o_totalprice) as r from orders) s where r between 1500000/2 and 1500000
/2+1;
 o_totalprice |   r
--------------+--------
    144409.02 | 750000
    144409.06 | 750001
(2 rows)
```

This also shows how much would it more difficult be to calculate the median using window-functions, then order-set functions, and in addition we would have to know in advance how many rows are there.

With ordered-set functions:

```
SELECT mode()
        WITHIN GROUP (ORDER BY o_totalprice)
FROM orders;
```

With window-functions:

```
SELECT o_totalprice, count(*) AS c
FROM orders
GROUP BY o_totalprice
ORDER BY c DESC
LIMIT 2;
```

```
neumann=# select mode() within group (order by o_totalprice) from orders;
   mode
----------
 27242.17
(1 row)

neumann=# select o_totalprice, count(*) as c from orders group by o_totalprice order by c desc limit 1;
 o_totalprice | c
--------------+---
     27242.17 | 4
(1 row)

neumann=# select o_totalprice, count(*) as c from orders group by o_totalprice order by c desc limit 2;
 o_totalprice | c
--------------+---
     27242.17 | 4
     43328.69 | 4
(2 rows)
```

Order-set function „**mode()**“ gives **most frequent value**.

We see from the window-function version of the result of this problem, that there was more than one element which occurred the same number of times.

```
SELECT mode()
        WITHIN GROUP (ORDER BY o_orderstatus)
FROM orders;
```

```
neumann=# select mode() within group (order by o_orderstatus) from orders;
 mode
------
 O
(1 row)

neumann=# select o_orderstatus, count(*) as c from orders group by o_orderstatus order by c desc limit 2;
 o_orderstatus |   c
---------------+--------
 O             | 732044
 F             | 729413
(2 rows)
```

In this example the element which occurred most times is unique.

```
SELECT o_custkey, percentile_cont(0.5)
        WITHIN GROUP ( ORDER BY o_totalprice)
FROM orders
GROUP BY o_custkey
```

```
neumann=# select o_custkey, percentile_cont(0.5) within group (order by o_totalprice) from orders group by o_custkey;
```

o_custkey	percentile_cont
1	85256.91
2	167016.61
4	98043.81
5	173254.66999999998
7	186300.05
8	134464.25
10	146730.585
11	198540.63
13	117162.07
14	86391.31999999999
16	166072.71
17	178491.405
19	125732.41500000001
20	132046.91499999998
22	171317.905
23	174664.45
25	170402.38
26	163530.39
28	180371.495
29	145635.04499999998
31	142954.89
32	137908.44
34	215578.16
35	106502.09
37	116665.555
38	175062.385
40	156289.75
41	160224.44
43	183152.11
44	187465.115
46	148670.08500000002
47	164956.3
49	157916.63
50	99363.675

It is also possible to combine order-set functions with „GROUP BY“ clause, like in this example where we calculate the median value of spending of each customer.

```
neumann=# select r_regionkey from region;
 r_regionkey
-------------
           0
           1
           2
           3
           4
(5 rows)

neumann=# select percentile_cont(0.5) within group (order by r_regionkey) from region;
 percentile_cont
-----------------
               2
(1 row)

neumann=# select percentile_cont(0.4) within group (order by r_regionkey) from region;
 percentile_cont
-----------------
             1.6
(1 row)

neumann=# select percentile_cont(0.3) within group (order by r_regionkey) from region;
 percentile_cont
-----------------
             1.2
(1 row)

neumann=# select percentile_cont(0.35) within group (order by r_regionkey) from region;
 percentile_cont
-----------------
             1.4
(1 row)
```

Not only does the order-set function „percentile_cont(p)“perform interpolation when calculating the median value, but for values which depend on the value you specify in the function „percentile_count(p)“ as „p“.

```
neumann=# select percentile_cont(0.95) within group (order by o_orderdate) from orders;
ERROR:  function percentile_cont(numeric, date) does not exist
LINE 1: select percentile_cont(0.95) within group (order by o_orderd...
                ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
neumann=# select percentile_disc(0.95) within group (order by o_orderdate) from orders;
 percentile_disc
-----------------
 1998-04-04
(1 row)
```

Ordered-set function „percentile_cont(p)“ works only on numeric values, as it performs interpolation, so when you give it a non-numeric value, it tries to perform interpolation and gives an error as output. On the other hand, when you try function „percentile_disc(p)“ on a non-numeric value it will be able to work, as it just depends only on numbers of rows.

3. GROUPING SETS, ROLLUP AND CUBE

3.1 Introduction to Grouping sets, Rollup and Cube

We have types of SQL features that we use as a part of the "GROUP BY" clause, which we can use to perform aggregation over multiple dimensions, and these are:

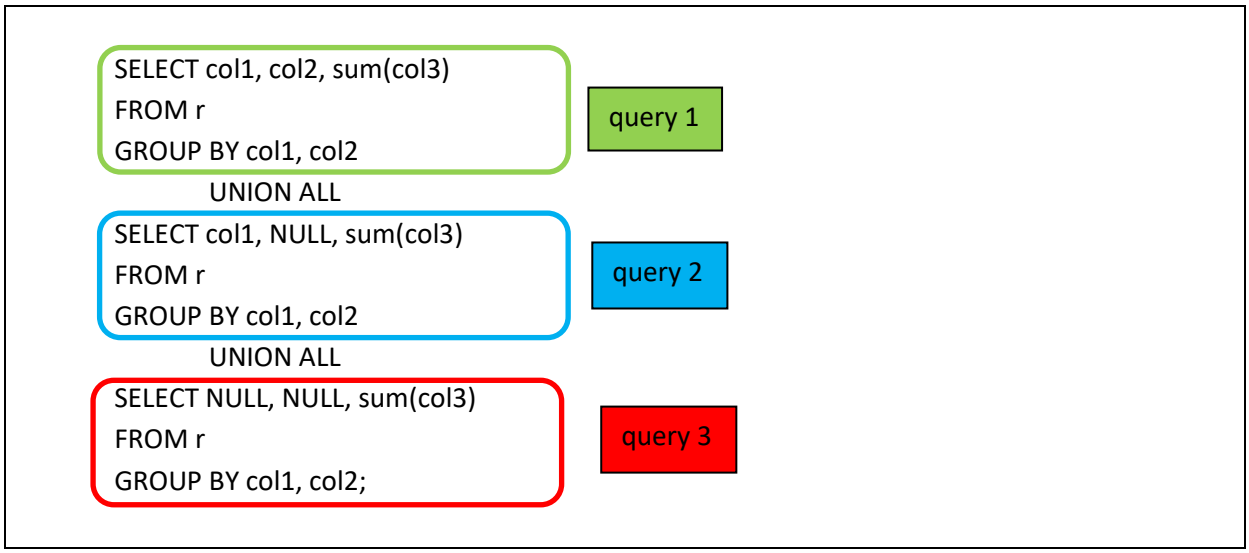
1. GROUPING SETS(sets_of_columns)
2. ROLLUP(list_of_columns)
3. CUBE(list_of_columns)

3.1.1 GROUP BY GROUPING SETS

When using "GROUP BY GROUPING SETS(sets_of_columns)" we can explicitly specify multiple sets of columns according to which grouping is going to be performed. So, for an example we could write

```
GROUP BY GROUPING_SETS( (col1, col2), (col1), () )
```

This would be equivalent to SQL code:



You could have specified different grouping sets like:

```
GROUP BY GROUPING SETS( (col1, col2), (col1), (col2) )
```

or

```
GROUP BY GROUPING SETS( (col1), (col2) )
```

or any other combination of the groupings between columns.

3.1.2 GROUP BY ROLLUP

On the other hand, you could also use “ROLLUP(list_of_columns)” in the “GROUP BY” clause, and it is more popular to use this SQL feature than the “GROUPING SETS(sets_of_columns)”.

You could this SQL feature as:

```
GROUP BY ROLLUP(col1, col2)
```

which is equivalent to “GROUP BY GROUPING_SETS((col1, col2), (col1), ())”.

It is also possible to specify more than two columns in the list_of_columns of the “GROUP BY ROLLUP(list_of_columns)”, like for an example: “GROUP BY ROLLUP(col1, col2, col3)”.

3.1.3 GROUP BY CUBE

When using “GROUP BY CUBE(list_of_columns)”, we actually perform all possible combinations of groupings among columns specified in list_of_columns. So, if we wrote:

```
GROUP BY CUBE(col1, col2)
```

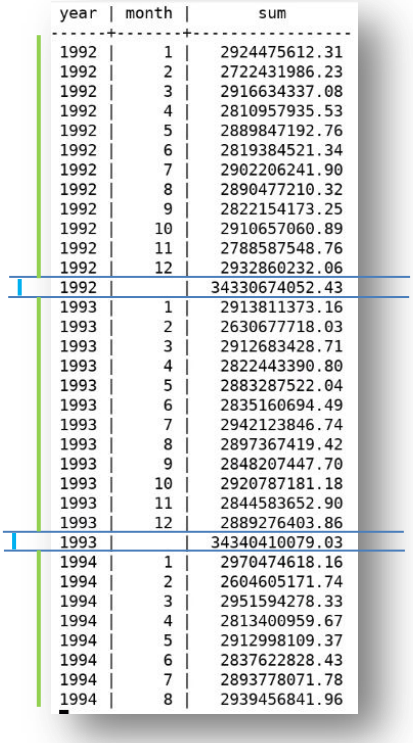
it would be equivalent to writing:

```
GROUP BY GROUPING SETS( (col1, col2), (col1), (col2), () )
```

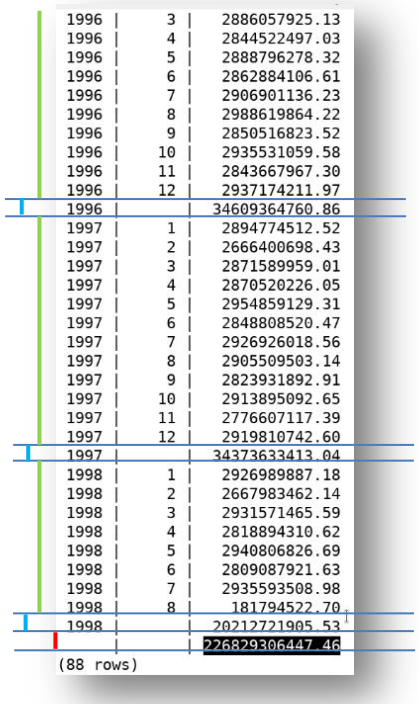
3.2 lecture 12 (FDE), examples, Grouping sets, Rollup and Cube

```
SELECT year, month, sum(o_totalprice)
FROM ( SELECT extract(year FROM o_orderdate) AS year, extract (month FROM o_orderdate) AS month, o_totalprice
        FROM orders) AS s
GROUP BY ROLLUP(year, month)
ORDER BY year, month;
```

```
neumann=# select year, month, sum(o_totalprice) from (select extract(year from o_orderdate) as year, extract(month from o_orderdate) as month, o_
totalprice from orders) s group by rollup (year, month) order by year, month;
```



year	month	sum
1992	1	2924475612.31
1992	2	2722431986.23
1992	3	2916634337.08
1992	4	2810957935.53
1992	5	2889847192.76
1992	6	2819384521.34
1992	7	2902206241.90
1992	8	2890477210.32
1992	9	2822154173.25
1992	10	2910657060.89
1992	11	2788587548.76
1992	12	2932860232.06
1992		34330674052.43
1993	1	2913811373.16
1993	2	2630677718.03
1993	3	2912683428.71
1993	4	2822443390.80
1993	5	2883287522.04
1993	6	2835160694.49
1993	7	2942123846.74
1993	8	2897367419.42
1993	9	2848207447.70
1993	10	2920787181.18
1993	11	2844583652.90
1993	12	2889276403.86
1993		34340410079.03
1994	1	2970474618.16
1994	2	2604605171.74
1994	3	2951594278.33
1994	4	2813400959.67
1994	5	2912998109.37
1994	6	2837622828.43
1994	7	2893778071.78
1994	8	2939456841.96



1996	3	2886057925.13
1996	4	2844522497.03
1996	5	2888796278.32
1996	6	2862884106.61
1996	7	2906901136.23
1996	8	2988619864.22
1996	9	2850516823.52
1996	10	2935531059.58
1996	11	2843667967.30
1996	12	2937174211.97
1996		34609364760.86
1997	1	2894774512.52
1997	2	2666400698.43
1997	3	2871589959.01
1997	4	2870520226.05
1997	5	2954859129.31
1997	6	2848808520.47
1997	7	2926926018.56
1997	8	2905509503.14
1997	9	2823931892.91
1997	10	2913895092.65
1997	11	2776607117.39
1997	12	2919810742.60
1997		34373633413.04
1998	1	2926989887.18
1998	2	2667983462.14
1998	3	2931571465.59
1998	4	2818894310.62
1998	5	2940806826.69
1998	6	2809087921.63
1998	7	2935593508.98
1998	8	181794522.70
1998		20212721905.53
		226829306447.46

(88 rows)

• • •

Another row is added to the output (for each year) where there is no value (NULL technically speaking) in the column „month“, but there is a value in the column „sum“, which specifies the sum over all the months in a year. Also, the last row of the output gives a sum of all values in all years. This is called „**level of detail aggregation**“, we aggregate it over a month level, and over a year level, and over everything.

As mentioned earlier, the reason why we get such an output is because “GROUP BY ROLLUP(list_of_columns)” can be written as an union between three queries. To bring closer to the reader what is the result of which of these three queries three colors were used beside each row in the photos of the output above, the same colors used to denote each of the three queries in the 3.1.1 section of this document.

```
SELECT year, month, sum(o_totalprice)
FROM ( SELECT extract(year FROM o_orderdate) AS year, extract (quarter FROM o_orderdate) AS quarter, o_totalprice
        FROM orders) AS s
GROUP BY ROLLUP(year, quarter)
ORDER BY year, quarter;
```

(This query is a little bit changed in comparison to the down original one, written by the professor. The thing that is changed is that I put the column to be named „quarter“ instead of „month“, as professor forgot to do this, but the output is the same)

```
neumann=# select year, month, sum(o_totalprice) from (select extract(year from o_orderdate) as year, extract(quarter from o_orderdate) as month,
o_totalprice from orders) s group by rollup (year, month) order by year, month;
```

```
 1992 |     1 |   8563541935.62
 1992 |     2 |   8520189649.63
 1992 |     3 |   8614837625.47
 1992 |     4 |   8632104841.71
 1992 |       |  34330674052.43
 1993 |     1 |   8457172519.90
 1993 |     2 |   8540891607.33
 1993 |     3 |   8687698713.86
 1993 |     4 |   8654647237.94
 1993 |       |  34340410079.03
 1994 |     1 |   8526674068.23
 1994 |     2 |   8564021897.47
 1994 |     3 |   8662412198.04
 1994 |     4 |   8663260889.23
 1994 |       |  34416369052.97
 1995 |     1 |   8545777729.15
 1995 |     2 |   8607870947.61
 1995 |     3 |   8714236202.49
 1995 |     4 |   8678248304.35
 1995 |       |  34546133183.60
 1996 |     1 |   8550750816.08
 1996 |     2 |   8596202881.96
 1996 |     3 |   8746037823.97
 1996 |     4 |   8716373238.85
 1996 |       |  34609364760.86
 1997 |     1 |   8432765169.96
 1997 |     2 |   8674187875.83
 1997 |     3 |   8656367414.61
 1997 |     4 |   8610312952.64
 1997 |       |  34373633413.04
 1998 |     1 |   8526544814.91
 1998 |     2 |   8568789058.94
 1998 |     3 |   3117388031.68
 1998 |       |  20212721905.53
      |       | 226829306447.46
(35 rows)
```

•••

The same as previous example, just a year is divided into quarters instead of months, so that all the rows of the output can fit into the screen.

```
SELECT year, month, sum(o_totalprice)
FROM ( SELECT extract(year FROM o_orderdate) AS year, extract (month FROM o_orderdate) AS month, o_totalprice
        FROM orders) AS s
GROUP BY GROUPING SETS( (year, month), (year), (month) )
ORDER BY year, month;
```

```
neumann=# select year, month, sum(o_totalprice) from (select extract(year from o_orderdate) as year, extract(month from o_orderdate) as month, o_
totalprice from orders) s group by grouping sets ((year, month), (year), (month)) order by year, month;
```

year	month	sum
1992	1	2924475612.31
1992	2	2722431986.23
1992	3	2916634337.08
1992	4	2810957935.53
1992	5	2889847192.76
1992	6	2819384521.34
1992	7	2902206241.90
1992	8	2890477210.32
1992	9	2822154173.25
1992	10	2910657060.89
1992	11	2788587548.76
1992	12	2932860232.06
1992		34330674052.43
1993	1	2913811373.16
1993	2	2630677718.03
1993	3	2912683428.71
1993	4	2822443390.80
1993	5	2883287522.04
1993	6	2835160694.49
1993	7	2942123846.74
1993	8	2897367419.42
1993	9	2848207447.70
1993	10	2920787181.18
1993	11	2844583652.90
1993	12	2889276403.86
1993		34340410079.03
1994	1	2970474618.16
1994	2	2604605171.74
1994	3	2951594278.33
1994	4	2813400959.67
1994	5	2912998109.37
1994	6	2837622828.43
1994	7	2893778071.78
1994	8	2939456841.96
1996		34609364760.86
1997	1	2894774512.52
1997	2	2666400698.43
1997	3	2871589959.01
1997	4	2870520226.05
1997	5	2954859129.31
1997	6	2848808520.47
1997	7	2926926018.56
1997	8	2905509503.14
1997	9	2823931892.91
1997	10	2913895092.65
1997	11	2776607117.39
1997	12	2919810742.60
1997		34373633413.04
1998	1	2926989887.18
1998	2	2667983462.14
1998	3	2931571465.59
1998	4	2818894310.62
1998	5	2940806826.69
1998	6	2809087921.63
1998	7	2935593508.98
1998	8	181794522.70
1998		20212721905.53
	1	20480581098.99
	2	18742466233.97
	3	20380179720.89
	4	19834036099.40
	5	20368996890.96
	6	19869120928.41
	7	20458080715.17
	8	17749180192.98
	9	16991717101.97
	10	17494109915.63
	11	16945316176.59
	12	17515521372.50

(99 rows)

• • •

```
SELECT year, month, sum(o_totalprice)
FROM ( SELECT extract(year FROM o_orderdate) AS year, extract (month FROM o_orderdate) AS month, o_totalprice
        FROM orders) AS s
GROUP BY GROUPING SETS( (year, month), (year), (month), () )
ORDER BY year, month;
```

```
neumann=# select year, month, sum(o_totalprice) from (select extract(year from o_orderdate) as year, extract(month from o_orderdate) as month, o_
totalprice from orders) s group by grouping sets ((year, month), (year), (month), ()) order by year, month;
```

```
 1997 |     2 |   2666400698.43
 1997 |     3 |   2871589959.01
 1997 |     4 |   2870520226.05
 1997 |     5 |   2954859129.31
 1997 |     6 |   2848808520.47
 1997 |     7 |   2926926018.56
 1997 |     8 |   2905509503.14
 1997 |     9 |   2823931892.91
 1997 |    10 |   2913895092.65
 1997 |    11 |   2776607117.39
 1997 |    12 |   2919810742.60
 1997 |       |  34373633413.04
 1998 |     1 |   2926989887.18
 1998 |     2 |   2667983462.14
 1998 |     3 |   2931571465.59
 1998 |     4 |   2818894310.62
 1998 |     5 |   2940806826.69
 1998 |     6 |   2809087921.63
 1998 |     7 |   2935593508.98
 1998 |     8 |    181794522.70
 1998 |       |  20212721905.53
      |     1 |  20480581098.99
      |     2 |  18742466233.97
      |     3 |  20380179720.89
      |     4 |  19834036099.40
      |     5 |  20368996890.96
      |     6 |  19869120928.41
      |     7 |  20458080715.17
      |     8 |  17749180192.98
      |     9 |  16991717101.97
      |    10 |  17494109915.63
      |    11 |  16945316176.59
      |    12 |  17515521372.50
      |       | 226829306447.46
(100 rows)
```

•••

By adding in the clause „GROUP BY GROUPING SETS“ value „()“, this is used to refer to all the values (to be summed up in this case)

```
SELECT year, month, sum(o_totalprice)
FROM ( SELECT extract(year FROM o_orderdate) AS year, extract (month FROM o_orderdate) AS month, o_totalprice
        FROM orders) AS s
GROUP BY CUBE( year, month )
ORDER BY year, month;
```

```
neumann=# select year, month, sum(o_totalprice) from (select extract(year from o_orderdate) as year, extract(month from o_orderdate) as month, o_
totalprice from orders) s group by cube (year, month) order by year, month;
```

year	month	sum
1992	1	2924475612.31
1992	2	2722431986.23
1992	3	2916634337.08
1992	4	2810957935.53
1992	5	2889847192.76
1992	6	2819384521.34
1992	7	2902206241.90
1992	8	2890477210.32
1992	9	2822154173.25
1992	10	2910657060.89
1992	11	2788587548.76
1992	12	2932860232.06
1992		34330674052.43
1993	1	2913811373.16
1993	2	2630677718.03
1993	3	2912683428.71
1993	4	2822443390.80
1993	5	2883287522.04
1993	6	2835160694.49
1993	7	2942123846.74
1993	8	2897367419.42
1993	9	2848207447.70
1993	10	2920787181.18
1993	11	2844583652.90
1993	12	2889276403.86
1993		34340410079.03
1994	1	2970474618.16
1994	2	2604605171.74
1994	3	2951594278.33
1994	4	2813400959.67
1994	5	2912998109.37
1994	6	2837622828.43
1994	7	2893778071.78
1994	8	2939456841.96
1997	2	2666400698.43
1997	3	2871589959.01
1997	4	2870520226.05
1997	5	2954859129.31
1997	6	2848808520.47
1997	7	2926926018.56
1997	8	2905509503.14
1997	9	2823931892.91
1997	10	2913895092.65
1997	11	2776607117.39
1997	12	2919810742.60
1997		34373633413.04
1998	1	2926989887.18
1998	2	2667983462.14
1998	3	2931571465.59
1998	4	2818894310.62
1998	5	2940806826.69
1998	6	2809087921.63
1998	7	2935593508.98
1998	8	181794522.70
1998		20212721905.53
	1	20480581098.99
	2	18742466233.97
	3	20380179720.89
	4	19834036099.40
	5	20368996890.96
	6	19869120928.41
	7	20458080715.17
	8	17749180192.98
	9	16991717101.97
	10	17494109915.63
	11	16945316176.59
	12	17515521372.50
		226829306447.46

(100 rows)

•••

This query written using „GROUP BY CUBE“ clause means all 2^n combinations of groupings, so it could be written with „GROUP BY GROUPING SETS“ clause like:

```
SELECT year, month, sum(o_totalprice)
FROM ( SELECT extract(year FROM o_orderdate) AS year, extract (month FROM o_orderdate) AS month, o_totalprice
        FROM orders) AS s
GROUP BY GROUPING  SETS( (year, month), (year), (month), () )
ORDER BY year, month;
```

Using „CUBE“ may be very useful sometimes, but it should be used only if a few attributes are included in the clause, as otherwise the output would be huge.

```
SELECT year, quarter, month, sum(o_totalprice)
FROM ( SELECT extract(year FROM o_orderdate) AS year, extract (quarter FROM o_orderdate) AS quarter, extract
                                                    (month  FROM o_orderdate) AS month, o_totalprice
        FROM orders) AS s
GROUP BY ROLLUP(year, quarter, month )
ORDER BY year, month;
```

```
neumann=# select year, quarter, month, sum(o_totalprice) from (select extract(year from o_orderdate) as year, extract(quarter from o_orderdate) a
s quarter, extract(month from o_orderdate) as month, o_totalprice from orders) s group by rollup (year, quarter, month) order by year, quarter, m
onth;
```

year	quarter	month	sum
1992	1	1	2924475612.31
1992	1	2	2722431986.23
1992	1	3	2916634337.08
1992	1		8563541935.62
1992	2	4	2810957935.53
1992	2	5	2889847192.76
1992	2	6	2819384521.34
1992	2		8520189649.63
1992	3	7	2902206241.90
1992	3	8	2890477210.32
1992	3	9	2822154173.25
1992	3		8614837625.47
1992	4	10	2910657060.89
1992	4	11	2788587548.76
1992	4	12	2932860232.06
1992	4		8632104841.71
1992			34330674052.43
1993	1	1	2913811373.16
1993	1	2	2630677718.03
1993	1	3	2912683428.71
1993	1		8457172519.90
1993	2	4	2822443390.80
1993	2	5	2883287522.04
1993	2	6	2835160694.49
1993	2		8540891607.33
1993	3	7	2942123846.74
1993	3	8	2897367419.42
1993	3	9	2848207447.70
1993	3		8687698713.86
1993	4	10	2920787181.18
1993	4	11	2844583652.90
1993	4	12	2889276403.86
1993	4		8654647237.94
1993			34340410079.03
1996	4	11	2843667967.30
1996	4	12	2937174211.97
1996	4		8716373238.85
1996			34609364760.86
1997	1	1	2894774512.52
1997	1	2	2666400698.43
1997	1	3	2871589959.01
1997	1		8432765169.96
1997	2	4	2870520226.05
1997	2	5	2954859129.31
1997	2	6	2848808520.47
1997	2		8674187875.83
1997	3	7	2926926018.56
1997	3	8	2905509503.14
1997	3	9	2823931892.91
1997	3		8656367414.61
1997	4	10	2913895092.65
1997	4	11	2776607117.39
1997	4	12	2919810742.60
1997	4		8610312952.64
1997			34373633413.04
1998	1	1	2926989887.18
1998	1	2	2667983462.14
1998	1	3	2931571465.59
1998	1		8526544814.91
1998	2	4	2818894310.62
1998	2	5	2940806826.69
1998	2	6	2809087921.63
1998	2		8568789058.94
1998	3	7	2935593508.98
1998	3	8	181794522.70
1998	3		3117388031.68
1998			20212721905.53
			226829306447.46

(115 rows)

• • •

If for the same problem as the pervious one we used „CUBE“, instead of „ROLLUP“, we would get the following:

```
SELECT year, quarter, month,  sum(o_totalprice)
FROM ( SELECT extract(year FROM o_orderdate) AS year, extract (quarter FROM o_orderdate) AS quarter, extract
                                                    (month FROM o_orderdate) AS month, o_totalprice
        FROM orders) AS s
GROUP BY CUBE( year, quarter, month )
ORDER BY year, month;
```

```
neumann=# select year, quarter, month, sum(o_totalprice) from (select extract(year from o_orderdate) as year, extract(quarter from o_orderdate) a
s quarter, extract(month from o_orderdate) as month, o_totalprice from orders) s group by cube (year, quarter, month) order by year, quarter, mon
th;
```

```
 1998 |         |     5 |   2940806826.69
 1998 |         |     6 |   2809087921.63
 1998 |         |     7 |   2935593508.98
 1998 |         |     8 |    181794522.70
 1998 |         |       |  20212721905.53
      |       1 |     1 |  20480581098.99
      |       1 |     2 |  18742466233.97
      |       1 |     3 |  20380179720.89
      |       1 |       |  59603227053.85
      |       2 |     4 |  19834036099.40
      |       2 |     5 |  20368996890.96
      |       2 |     6 |  19869120928.41
      |       2 |       |  60072153918.77
      |       3 |     7 |  20458080715.17
      |       3 |     8 |  17749180192.98
      |       3 |     9 |  16991717101.97
      |       3 |       |  55198978010.12
      |       4 |    10 |  17494109915.63
      |       4 |    11 |  16945316176.59
      |       4 |    12 |  17515521372.50
      |       4 |       |  51954947464.72
      |         |     1 |  20480581098.99
      |         |     2 |  18742466233.97
      |         |     3 |  20380179720.89
      |         |     4 |  19834036099.40
      |         |     5 |  20368996890.96
      |         |     6 |  19869120928.41
      |         |     7 |  20458080715.17
      |         |     8 |  17749180192.98
      |         |     9 |  16991717101.97
      |         |    10 |  17494109915.63
      |         |    11 |  16945316176.59
      |         |    12 |  17515521372.50
      |         |       | 226829306447.46
(223 rows)
```

•••

This example is used to show how usage of „CUBE“ may drastically increase the number of output rows. In the above example there were 115 outputted rows (using „ROLLUP“), while using „CUBE“ for the 'same' query we get 223 rows.

Exercise from slide 36, FDE, section „Using a Database“:

- Aggregate revenue (“sum(o_totalprice)”): total, by region (r_name) by name (n_name), example output:

revenue	region	nation
836330704.31	AFRICA	ALGERIA
902849428.98	AFRICA	ETHIOPIA
784205751.27	AFRICA	KENYA
893122668.52	AFRICA	MOROCCO
852278134.31	AFRICA	MOZAMBIQUE
4268786687.39	AFRICA	
...		
21356596030.63		

```
SELECT r_name, n_name, sum(o_totalprice)
FROM ( SELECT r_name, n_name, o_totalprice
        FROM oders, customer, nation, region
        WHERE c_custkey=o_custkey AND c_nationkey=n_nationkey AND r_regionkey=n_regionkey) AS s
GROUP BY ROLLUP(r_name, n_name);
```

or equivalently:

```
SELECT r_name, n_name, sum(o_totalprice)
FROM ( SELECT r_name, n_name, o_totalprice
        FROM oders, customer, nation, region
        WHERE c_custkey=o_custkey AND c_nationkey=n_nationkey AND r_regionkey=n_regionkey) AS s
GROUP BY GROUPING SETS( (r_name, n_name), (r_name), () );
```

```
neumann=# select r_name, n_name, sum(o_totalprice) from (select r_name, n_name, o_totalprice from orders, customer, nation, region where c_custke
y=o_custkey and c_nationkey=n_nationkey and r_regionkey=n_regionkey) s group by rollup(r_name, n_name);
```

r_name	n_name	sum
AFRICA	ALGERIA	9065723966.78
AFRICA	ETHIOPIA	9032642974.38
AFRICA	KENYA	8897163266.18
AFRICA	MOROCCO	8985579085.22
AFRICA	MOZAMBIQUE	9249114609.66
AFRICA		45230223902.22
AMERICA	ARGENTINA	9022490350.05
AMERICA	BRAZIL	9107367126.70
AMERICA	CANADA	9143635385.19
AMERICA	PERU	8946481134.38
AMERICA	UNITED STATES	9086969258.89
AMERICA		45306943255.21
ASIA	CHINA	9161685172.34
ASIA	INDIA	9035791922.09
ASIA	INDONESIA	9300830039.29
ASIA	JAPAN	8993418470.64
ASIA	VIETNAM	9121689438.20
ASIA		45613415042.56
EUROPE	FRANCE	9318715232.78
EUROPE	GERMANY	9039194008.74
EUROPE	ROMANIA	9196280024.51
EUROPE	RUSSIA	9282323186.28
EUROPE	UNITED KINGDOM	8956753007.40
EUROPE		45793265459.71
MIDDLE EAST	EGYPT	8925726169.17
MIDDLE EAST	IRAN	9025552858.09
MIDDLE EAST	IRAQ	8892603095.99
MIDDLE EAST	JORDAN	9229296044.98
MIDDLE EAST	SAUDI ARABIA	8812280619.53
MIDDLE EAST		44885458787.76
		226829306447.46

(31 rows)