5-9 November

Athens Hilton

Athens, Greece

Session: F07

CTEs in DB2 for z/OS:
more powerful than you imagine

IDUG® 2007 Europe

Peter Vanroose

ABIS Training and Consulting

6 November 2007 • 15:30 – 16:30

Platform: DB2 for z/OS

IDUG The Worldwide DB2 User Community

GoFurther

Agenda

- SQL query blocks in DB2: syntax history
- views, nested table expressions, and common table expressions (CTEs)
- *fil rouge:* the iterative querying paradigm
- recursion / iteration
- Case studies:
  - How to get a top-10 list in a readable way?
  - Whose birthday this week?
  - How to find out about “holes” in a table?
  - How to circumvent non-indexable predicates?

SQL nested queries: syntax history

- 1970: relational model:
  - joins; selection & projection; views
- 1983: DB2 v1 for MVS:
  - subqueries (in WHERE clause): IN, EXISTS
  - views (DDL)
- 1994: DB2 v4 for MVS:
  - nested table expressions (in FROM clause)
- 1999: SQL-99 standard defines CTE
- 2004: DB2 v8 for z/OS:
  - common table expressions (“WITH” subclause)
  - “scalar fullselect” query blocks as expressions

Nested table expressions

- "iterative querying" paradigm:
  - create intermediate result table ("view")
  - use it in a next querying iteration
  - etc.
- Example: *which was the most popular product?*

```
SELECT MAX(p.c)
FROM   (SELECT o.product, COUNT(*) AS c
        FROM   (SELECT product
                FROM   orders
                WHERE  orderdate IN ...) AS o
        GROUP BY product
       ) AS p
```

The “subquery” solution:

- Example: *which was the most popular product?*

```
SELECT product, COUNT(*)
FROM   orders
WHERE  orderdate IN ...
GROUP BY product
HAVING COUNT(*) >= ALL ( SELECT COUNT(*)
                         FROM   orders
                         WHERE  orderdate IN ...
                         GROUP BY product       )
```

- Exercise: the “self-join” solution ...

The “view” solution:

- Example: *which was the most popular product?*

```
 CREATE VIEW o(product)    AS SELECT product FROM orders
                              WHERE  orderdate IN ...
;
 CREATE VIEW p(product,c)  AS SELECT product, COUNT(*)
                              FROM   o
                              GROUP BY product
;
 SELECT MAX(c)    |    SELECT product, c
 FROM   p         |    FROM   p
;                 |    WHERE  c = (SELECT MAX(c)
                  |                  FROM   p     )
                  |  ;
```

The CTE solution:

- Example: *which was the most popular product?*

```
WITH o(product)    AS (SELECT product
                       FROM   orders
                       WHERE  orderdate IN ...  )
,
     p(product, c) AS (SELECT product, COUNT(*)
                       FROM   o
                       GROUP BY product         )
SELECT product, c
FROM   p
WHERE  c = (SELECT MAX(c) FROM p)
```

- For DB2, this is equivalent to the NTE formulation

Advantages of using a CTE

- Avoid duplicate code (maintainability!)
- Avoid duplicate operations (performance!)
- Avoid creating a view
- Modularity, readability
- Create a “temp table” based on host variables
- Let the optimizer decide on materialization
- Create a “temp table” through *iteration* (recursion)

No need anymore for NTEs, temp VIEWs, temp TBLs

CTEs and recursion (iteration)

- SQL-99 allows CTEs to refer itself
- DB2 v8 implements only the possibility of *iteration*:

```
WITH v (c1, c2, c3) AS
     ( SELECT ...  -- non-recursive part: initialization
      UNION ALL
       SELECT ...
       FROM   v .. -- recursive part: single iteration
     )
SELECT ... FROM v ...
```

- First SELECT must not refer v; typically: single row
- Second SELECT: *iteration*; “v” means “previous step”

CTEs and recursion: an example

```
WITH v(n) AS
     ( VALUES (1)      -- this also sets the data type
      UNION ALL
       SELECT n+1
       FROM   v
       WHERE  n < 100 -- stop criterion
     )
SELECT * FROM v
```

This CTE is a 100-row “table” with values 1, 2, ... 100.

CTEs and recursion: an example

- DB2 v8 for z/OS does not yet support “VALUES”:

```
WITH v(n) AS
     ( SELECT 1 FROM sysibm.sysdummy1
       UNION ALL
        SELECT n+1
        FROM   v
        WHERE  n < 100 -- stop criterion
     )
SELECT * FROM v
```

This CTE is a 100-row “table” with values 1, 2, ... 100.

CTEs and recursion: SQLCODE 347

- Be prepared for SQL return code +347 !
- SQLCODE=0 only when *guarantee* of non-infinite loop.

```
WITH v(name, id, su_id) AS
     ( SELECT name, id, supervisor_id
       FROM   personnel
       WHERE  dept = ...
      UNION ALL
       SELECT name, id, supervisor_id
       FROM   personnel
       WHERE  id IN (SELECT su_id FROM v)
)
SELECT name FROM v -- names of (indirect) supervisors
```

CTEs and explain

- Recursive CTE:
  - plan_table has line with TABLE_TYPE = 'R'
- Non-recursive CTE:
  - plan_table has line with TABLE_TYPE = 'C'
- Nested table expression or view (non-materialized):
  - plan_table has line with TABLE_TYPE = 'Q'
- Materialization of any of these will be visible:
  - sort: plan_table has METHOD = 3
  - work table: plan_table has TABLE_TYPE = 'W'

Case studies

- Several example problem settings follow:

  1) Give the 10 most popular names (1 table)
  2) Give a ranking of companies for # employees (2 tables)
  3) Find “missing data” (holes) in a table
  4) Use CTEs for performance
  5) Whose birthday next week?
  6) Aggregate concatenation

- Different solutions are compared
  - for readability
  - for performance (==> with EXPLAIN output)
- CTE solution(s) are highlighted:
  - the “iterative querying” paradigm
  - often best performance & readability !

1(a). single table, group by

- *Give 10 most popular first names from birth register*

```
SELECT name, COUNT(*) AS freq
FROM   birth_list
GROUP BY name
ORDER BY freq DESC
FETCH FIRST 10 ROWS ONLY
```

==> ties possibly not shown ...

1(a). single table, group by: *explain*

- plan_table:

```
QBLOCKNO METHOD TNAME      MC ACCESSTYPE INDEXONLY
       1      0 BIRTH_LIST  0 I          Y
       1      3
```

sort for “group by” through index access (index-only)
additional sort for “order by”
“non-standard” solution because of FETCH FIRST
(disadvantage: “random” choice of ties
because exactly 10 rows)

1(b). single table, group by

- *Give 10 most popular first names from birth register*

```
WITH p(n, c) AS ( SELECT name, COUNT(*)
                  FROM   birth_list
                  GROUP BY name
                )
SELECT p1.n, p1.c
FROM   p AS p1
        INNER JOIN
       p AS p2     ON p1.c <= p2.c
GROUP BY p1.n, p1.c
HAVING COUNT(*) <= 10
```

1(b). single table, group by: *explain*

- plan_table:

QBLOCKNO	METHOD	TNAME	ACCESSTYPE	MC	INDEXONLY	CORRELATION_NAME	QBLOCK_TYPE	TABLE_TYPE
2	0	B_LIST	I	0	Y		NCOSUB	T
1	0	P	R			P1	SELECT	C
1	1	P	R			P2	SELECT	C
1	3						SELECT	

two table scans of (potentially large) CTEs

materialized sort for “group by”

all ties shown: possibly less then 10 rows returned

(Exercise: make that “at least 10” ...)

1(c). single table, group by

- *Give 10 most popular first names from birth register*

```
WITH p(n, c)  AS ( SELECT name, COUNT(*)
                   FROM   birth_list
                   GROUP BY name
                 )
 ,   q(n,c,r) AS ( SELECT n, c,
                          RANK() OVER (ORDER BY c DESC)
                   FROM   p
                 )
SELECT n, c   FROM   q
WHERE  r <= 10
ORDER BY r
```

(OLAP functionality: DB2 9 for z/OS, or DB2 for L/U/W)

2(a). two tables, group by

- *Give company with highest number of employees*

```
SELECT name
FROM   companies
WHERE  id IN ( SELECT employer_id
               FROM   persons
               GROUP BY employer_id
               HAVING COUNT(*) >= ALL   -- “max”
                                  (SELECT COUNT(*)
                                   FROM   persons
                                   GROUP BY employer_id
                                  )
             )
```

2(a). two tables, group by: *explain*

- plan_table:

```
QBLOCKNO METHOD TNAME     ACCESSTYPE MC INDEXONLY QBLOCK_TYPE TABLE_TYPE
       3      0 PERSONS   I           0 Y         NCOSUB      T
       2      0 PERSONS   I           0 Y         NCOSUB      T
       2      3                                   NCOSUB
       1      0 COMPANIES R                       SELECT      T
```

persons accessed twice (ix-only), once per subquery
sort of *persons* (for removal of duplicates)
table scan of *companies* (no query rewrite into join)

2(b). two tables, group by

- *Give company with highest number of employees*

```
WITH p(i, c)  AS (SELECT employer_id, COUNT(*)
                  FROM   persons
                  GROUP BY employer_id
                 )
   ,
     q(cmax)  AS (SELECT MAX(c) FROM p)

SELECT name
FROM   companies INNER JOIN p ON p.i = id
                 INNER JOIN q ON q.cmax = c
```

2(b). two tables, group by: *explain*

- plan_table:

QBLOCKNO	METHOD	TNAME	ACCESSTYPE	MC	INDEXONLY	QBLOCK_TYPE	TABLE_TYPE
2	0	PERSONS	I	0	Y	NCOSUB	T
3	0	P	R			TABLEX	C
1	0	Q	R			SELECT	W
1	1	P	R			SELECT	C
1	1	COMPANIES	I	1	N	SELECT	T

persons accessed twice (ix-only): no materialization

materialization of *q*, to perform join

q used as outer table in join

companies accessed (nested loop) through ix on id

2(c). two tables, group by

- *Give company with highest number of employees*

```
WITH p(i, c)  AS (SELECT employer_id, COUNT(*)
                  FROM   persons
                  GROUP BY employer_id
                 )
   ,
     q(cmax)  AS (SELECT MAX(c) FROM p)

SELECT name
FROM   companies
WHERE  id IN (SELECT i FROM p
              WHERE  c = (SELECT cmax FROM q)
             )
```

2(c). two tables, group by: *explain*

- plan_table:

QBLOCKNO	METHOD	TNAME	ACCESSTYPE	MC	INDEXONLY	QBLOCK_TYPE	TABLE_TYPE
3	0	PERSONS	I	0	Y	NCOSUB	T
4	0	P	R			NCOSUB	C
2	0	P	R			NCOSUB	C
2	3					NCOSUB	
1	0	COMPANIES	N	1	N	SELECT	T

q optimized away in qblockno = 4

persons accessed just once (ix-only) in definition of *p*

materialization of *p* (sort for removal of duplicates)

3(a). missing rows ("holes") in table

- *Which days of 2006 had no orders?*

```
WITH days(d) AS ( VALUES (CAST('2006-01-01' AS date))
                  UNION ALL
                   SELECT d + 1 DAY
                   FROM   days
                   WHERE   d < '2006-12-31' )
SELECT d
FROM   days
WHERE  d NOT IN (SELECT date FROM orders)
```

(returns SQLCODE +347)

note the CAST !

3(a). holes in table: *explain*

- plan_table:

QBLOCKNO	METHOD	TNAME	ACCESSTYPE	MC	INDEXONLY	QBLOCK_TYPE	TABLE_TYPE
2	0					UNIONA	
3	0	DUMMY1	R			NCOSUB	T
4	0	DAYS	R			NCOSUB	R
5	0	ORDERS	I	0	N	NCOSUB	T
5	3					NCOSUB	
1	0	DAYS	R			SELECT	C

recursive part of union in qblockno = 4

orders accessed through primary key (cluster) index

sort of *orders* for removal of duplicate dates

qblockno = 3: “sysibm.sysdummy1” used

table scan of *days* needed for “not in”

3(b). holes in table

- *Which days of 2006 had no orders?*

```
WITH days(d) AS ( VALUES (CAST('2006-01-01' AS date))
                  UNION ALL
                   SELECT d + 1 DAY
                   FROM   days
                   WHERE   d < '2006-12-31' )
SELECT d
FROM   days
WHERE  NOT EXISTS (SELECT 1
                    FROM   orders
                    WHERE  date = days.d)
```

3(b). holes in table: *explain*

- plan_table:

QBLOCKNO	METHOD	TNAME	ACCESSTYPE	MC	INDEXONLY	QBLOCK_TYPE	TABLE_TYPE
2	0					UNIONA	
3	0	DUMMY1	R			NCOSUB	T
4	0	DAYS	R			NCOSUB	R
5	0	ORDERS	I	0	N	CORSUB	T
1	0	DAYS	R			SELECT	C

sort avoided, but correlated subquery instead
rest is unchanged

3(c). holes in table

- *Which days of 2006 had no orders?*

```
WITH days(d) AS ( VALUES (CAST('2006-01-01' AS date))
                  UNION ALL
                   SELECT d + 1 DAY
                   FROM   days
                   WHERE   d < '2006-12-31' )
SELECT d
FROM   days  LEFT OUTER JOIN  orders  ON  date = d
WHERE  orders.pk IS NULL
```

3(c). holes in table: *explain*

- plan_table:

QBLOCKNO	METHOD	TNAME	ACCESSTYPE	MC	INDEXONLY	SORTN_JOIN	SORTC	QBLOCK_TYPE	TABLE_TYPE
2	0							UNIONA	
3	0	DUMMY1	R					NCOSUB	T
4	0	DAYS	R					NCOSUB	R
1	2	ORDERS	I	0	N	Y	Y	SELECT	T
1	0	DAYS	R					SELECT	C

chronological sort of *days* & *orders* (merge scan join)

rest unchanged

3(d). holes in table

- *Which non-weekend days of 2006 had no orders?*

```
WITH days(d) AS ( VALUES (CAST('2006-01-01' AS date))
                  UNION ALL
                   SELECT d + 1 DAY
                   FROM   days
                   WHERE   d < '2006-12-31' )
 ,   workdays(d) AS (SELECT d FROM days
                      WHERE  dayofweek(d) IN (2,3,4,5,6))
SELECT d FROM weekdays
WHERE  NOT EXISTS (SELECT 1
                   FROM   orders
                   WHERE  date = workdays.d)
```

3(d). holes in table: *explain*

- plan_table:

QBLOCKNO	METHOD	TNAME	ACCESSTYPE	MC	INDEXONLY	QBLOCK_TYPE	TABLE_TYPE
2	0					UNIONA	
3	0	DUMMY1	R			NCOSUB	T
4	0	DAYS	R			NCOSUB	R
6	0	ORDERS	I	0	N	CORSUB	T
1	0	DAYS	R			SELECT	C

sort avoided (correlated subquery)

workdays table is optimized away

equally performant as without extra condition

4(a). avoid table scan

- *Give all persons with email in domain "abis.be"*

```
SELECT name
FROM   persons
WHERE  email LIKE '%@abis.be %'
```

Assumptions:

- index on *name*
- index on *email*
- no composite index on (*name*, *email*) nor on (*email*, *name*)

4(a). avoid table scan: *explain*

- plan_table:

```
QBLOCKNO METHOD TNAME     ACCESSTYPE MC INDEXONLY QBLOCK_TYPE TABLE_TYPE
       1      0 PERSONS   R                       SELECT      T
```

table scan, since “LIKE '%...' is not indexable

4(b). avoid table scan

- *Give all persons with email in domain "abis.be"*

```
WITH emails(e) AS
     (SELECT email
      FROM   persons
      WHERE  email LIKE '%@abis.be %'
     )
SELECT name
FROM   persons
WHERE  email IN (SELECT e FROM emails)
```

4(b). avoid table scan: *explain*

- plan_table:

```
QBLOCKNO METHOD TNAME     ACCESSTYPE MC INDEXONLY QBLOCK_TYPE TABLE_TYPE
       3      0 PERSONS   I           0 Y         NCOSUB      T
       1      0 PERSONS   N           1 N         SELECT      T
```

non-matching index scan, but index-only,
followed by a matching index scan (with data access)
(CTE optimized away)

4(c). avoid table scan

- *Give all persons with email in domain "abis.be"*

```
WITH emails(e) AS
     (SELECT email
      FROM   persons
      WHERE  email LIKE '%@abis.be %'
     )
SELECT name
FROM   persons
       INNER JOIN
       emails
       ON email = e
```

4(c). avoid table scan: *explain*

- plan_table:

```
QBLOCKNO METHOD TNAME     ACCESSTYPE MC INDEXONLY QBLOCK_TYPE TABLE_TYPE
       2      0 PERSONS   I           0 Y         SELECT      T
       1      0 EMAILS    R                       SELECT      C
       1      1 PERSONS   I           1 N         SELECT      T
```

non-matching index scan, but index-only,
then a join from a (relatively small) CTE as outer table
to the index of the inner table, using a nested loop join

5(a). birthdays this week

- *Who is celebrating one's birthday this week?*

Problem: database has date of birth, not all birthdays

1st attempt; suppose start_d and end_d are this week's “boundaries”:

```
SELECT name
FROM   persons
WHERE  dayofyear(date_birth) BETWEEN
       dayofyear(start_d) AND dayofyear(end_d)
```

5(b). birthdays this week

- *Who is celebrating one's birthday this week?*

Problem: `dayofyear('2004-02-29') = dayofyear('2005-03-01')`

2nd attempt:

```
SELECT name
FROM   persons
WHERE  month(date_birth) || day(date_birth) BETWEEN
       month(start_d)    || day(start_d)    AND
       month(end_d)      || day(start_d)
```

(after the necessary casting, using `DIGITS()`, not `CAST(.. AS CHAR(2))` !)

5(c). birthdays this week

- *Who is celebrating one's birthday this week?*

Problem: what if `start_d = '2007-12-30'` and `end_d = '2008-01-05'` ?

3rd attempt:

```
SELECT name
FROM   persons
WHERE  date_birth – (year(date_birth) – 1000
                     – year(start_d)+year(end_d) ) years
       BETWEEN start_d – (year( st_date) – 1000
                     – year(start_d)+year(end_d) ) years
       AND     end_d   – (year(end_date) – 1000 ) years
```

5(d). birthdays this week

- *Who is celebrating one's birthday this week?*

Still some tweaks ...

4th attempt: finally! ==> Note the “reverse” logic!

```
WITH birthdays(name, d) AS
     (SELECT name, date_birth FROM persons
       UNION ALL
      SELECT name, d + 1 YEAR FROM birthdays
      WHERE  d < end_d)
SELECT name
FROM   birthdays
WHERE  d BETWEEN start_d and end_d
```

6(a). aggregate concatenation

- *List all items per category as a single text field*

```
WITH t(g, list) AS
     ( SELECT g, CAST(RTRIM(i) AS VARCHAR(254))
       FROM   items
      UNION ALL
       SELECT items.g, t.list || ', ' ||RTRIM(items.i)
       FROM   t INNER JOIN items ON t.g = items.g
       WHERE  LOCATE(RTRIM(items.i),t.list) = 0       )
  ,   l(g, len) AS (SELECT g, MAX(LENGTH(list))
                    FROM   t     GROUP BY g            )
SELECT t.g, MIN(t.list)
FROM   t INNER JOIN l ON l.g = t.g
WHERE  len = LENGTH(t.list)
GROUP BY t.g
```

6(b). aggregate concatenation

Problem: group by ==> want to see concatenation of a column
(alternative formulation: “pivot” or rotate a table)

Why is there no `CONCAT()` aggregate function?

==> `COUNT()`, `SUM()`, `MIN()`, `MAX()`, `AVG()` are symmetric!

==> `CONCAT()` would need an (implicit or explicit) ordering clause

Solution when number of rows per group is fixed:

```
SELECT min(v), max(v)
FROM   items
GROUP BY g
```

Impossible with unlimited or variable group size ...

6(c). aggregate concatenation

Example:

Input:

items.g	items.v
10	a
10	b
20	b
20	c
20	d

Expected output:

g	list
10	a, b
20	b, c, d

6(d). aggregate concatenation

Iterative ("recursive") SQL comes to the rescue:

- build up a CTE, `t`, having three columns, as follows:
  - First give it all groups of table items as (empty) rows, i.e.

```
(g,  list, aux)
(10, '',   '')
(20, '',   '')
```

  - Then add to this (though a `UNION ALL`) the join of `t` with `items`
    Result:

```
(10, '', '')   (10, ', a', 'a')   (20, ', b', 'b')
(20, '', '')   (10, ', b', 'b')   (20, ', c', 'c')
                                  (20, ', d', 'd')
```

  - Iterate this last step; make sure to (1) order, (2) avoid looping
    I.e.: rows of items are to be joined only if `t.aux < items.v`

6(e). aggregate concatenation

Hence the following rows are added to `t` in step 3:

```
(10, ', a, b', 'b')
(20, ', b, c', 'c')
(20, ', b, d', 'd')
(20, ', c, d', 'd')
```

Finally (for the small table used here) the last row to be added is

```
(20, ', b, c, d', 'd')
```

With this CTE, execute the query `SELECT g,SUBSTR(list,3) FROM t`
The “substring” function removes the leading “, ”

Need a “`WHERE`” condition to keep, per `g`, only longest strings of `list`

6(f). aggregate concatenation

Putting it all together:

```
WITH t (g, list, aux) AS
( SELECT DISTINCT g, CAST('' AS varchar(255)),
                     CAST(null AS varchar(255))
    FROM items
  UNION ALL
    SELECT t.g, t.list || ', ' || COALESCE(items.v, ''),
           COALESCE(items.v, '')
    FROM   t INNER JOIN items ON t.g = items.g
    WHERE  COALESCE(t.aux, '') < items.v
)
SELECT g, substr(list, 3)   FROM t AS tx
WHERE length(list) = (SELECT max(length(list)) FROM t
                       WHERE  t.f = tx.f                )
```

6(g). aggregate concatenation

- *Alternative solution:*

```
WITH t(g, list) AS
     ( SELECT g, CAST(v AS VARCHAR(254))
       FROM   items
      UNION ALL
       SELECT items.g, t.list || ', ' ||items.v
       FROM   t INNER JOIN items ON t.g = items.g
       WHERE  LOCATE(items.v,t.list) = 0          )
 ,   l(g, len) AS (SELECT g, MAX(LENGTH(list))
                   FROM   t     GROUP BY g        )
SELECT t.g, MIN(t.list)
FROM   t INNER JOIN l ON    l.g = t.g
                        AND l.len = LENGTH(t.list)
GROUP BY t.g
```

7. group by expression

- *Make a per-week sum of a table with per-day values*

```
WITH t(value, nr) AS
     (SELECT value, id/7 -- integer division
      FROM   entries
     )
SELECT SUM(value)
FROM   t
GROUP BY nr
```

8. use “history” table info

- *Which reorg was not followed by a copy (backup)?*

```
WITH data(t,ts,typ) AS
     ( SELECT timestamp,dbname||'.'||tsname,ictype
       FROM   sysibm.syscopy
     )
 ,   reorg(t,ts) AS
     ( SELECT * FROM data  WHERE typ IN ('W','X')
     )
SELECT * FROM reorg
WHERE  NOT EXISTS (SELECT 1 FROM data
                   WHERE  ictype IN ('I','F')
                     AND  t = ( SELECT MIN(t) FROM data
                                WHERE  t > reorg.t     )
```

9. avoid duplicate code

- *Give the cheapest holiday option*

```
WITH p AS (SELECT date, price, code
           FROM   package
           WHERE  duration=7 AND date = ...)
 ,   x AS (SELECT MIN(p.date) AS min_date
           FROM   p
           GROUP BY p.code                  )
SELECT p.code, x.min_date, MIN(p.price)
FROM   p INNER JOIN x
       ON p.code = x.code AND x.min_date = p.date
GROUP BY p.code,x.min_date
```

10. generate a table (function)

- *Give all working days for the next 10 years*

```
WITH holidays(d) AS (...)
  ,   all_days(d) AS (SELECT VALUES(date('2008-01-01'))
                      UNION ALL
                      SELECT d + 1 day FROM all_days
                      WHERE  d < '2017-12-31'          )
SELECT * FROM all_days
EXCEPT
SELECT * FROM holidays
```

(*In "...": combination of recursion and union of several cases:*

- "fixed" dates like January 1, December 25, etc.
- *"variable" dates based on formula for Easter date*)

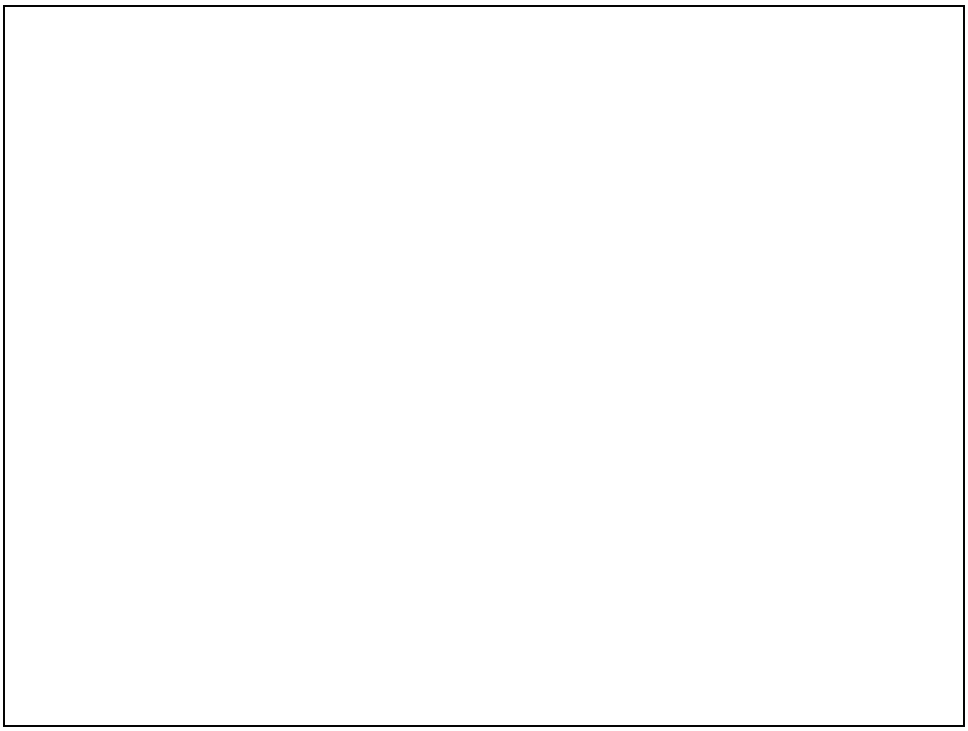
11. avoid multiple table scans

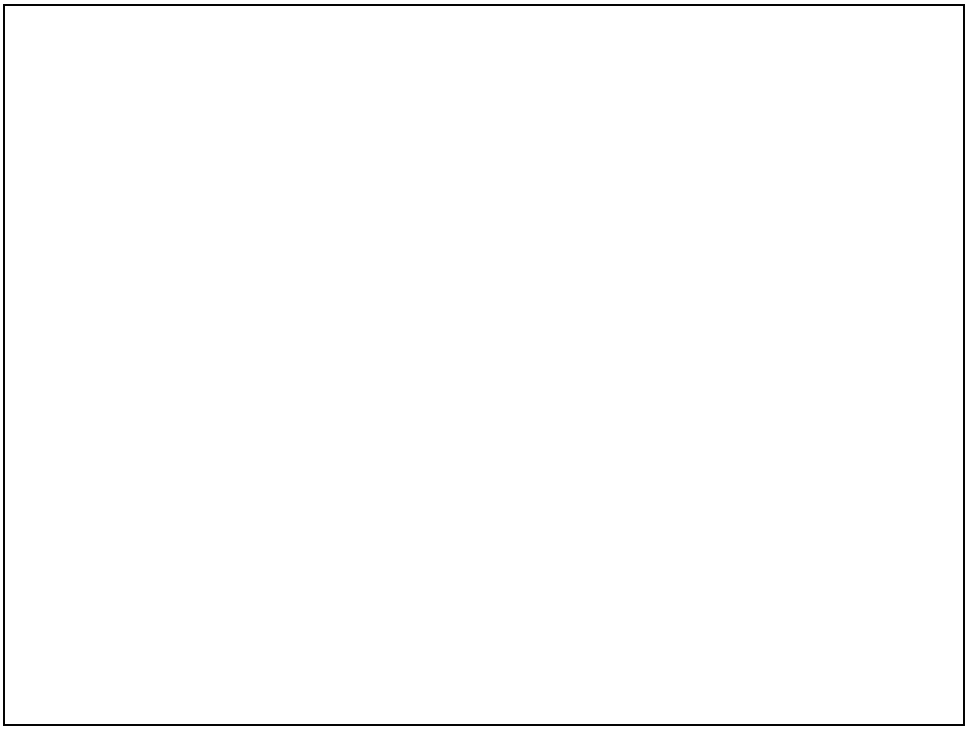
- *Give only the 10 first items per category*

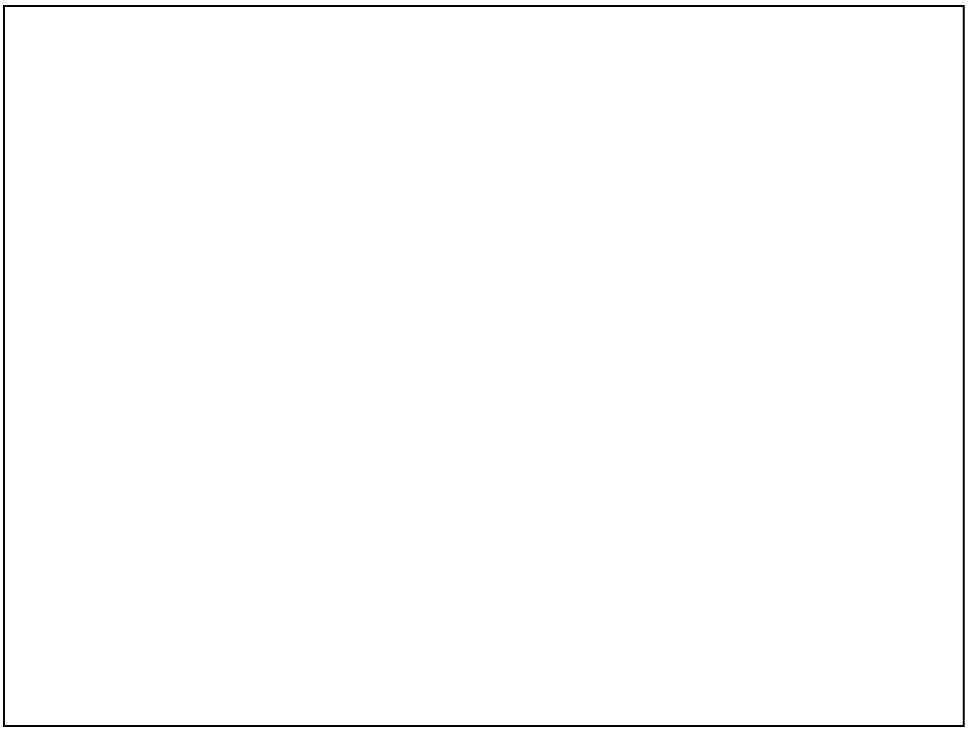
```
WITH i(item, category, r) AS
     (SELECT itemname, cat,
             rank() over (partition by cat)
      FROM   items
     )
SELECT category, item FROM i
WHERE  r <= 10
ORDER BY categ
```

CTEs in DB2 for z/OS: more powerful than you imagine

Peter Vanroose
ABIS Training & Consulting
Leuven, Belgium
http://www.abis.be/
pvanroose@abis.be







Nested queries, or "**query blocks**": SELECT clauses used inside other queries.

The relational model (1970) already has the notion of "iterative" querying:
must use **views** for this purpose.

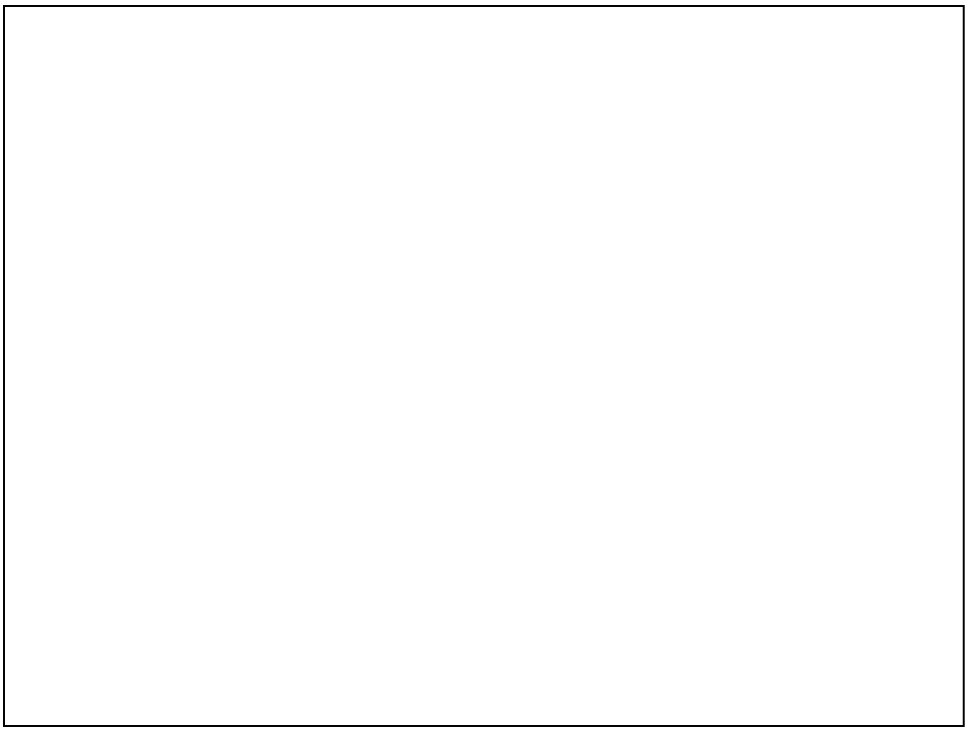
The first versions of DB2 only support views, and **subqueries** (i.e., nested queries in the WHERE subclause, inside an **IN** or **EXISTS**

Since version 4 (1994), **nested table expressions** (i.e., query blocks in the FROM subclause) were introduced: functionally equivalent to a view, but no DDL needed

The SQL-92 standard provides the possibility to use query blocks (at least, if they return just a single cell: **scalar fullselects**) wherever expressions are allowed:
- in SELECT subclause
- as arguments to scalar functions, CASE expressions, ...
- in GROUP BY or ORDER BY

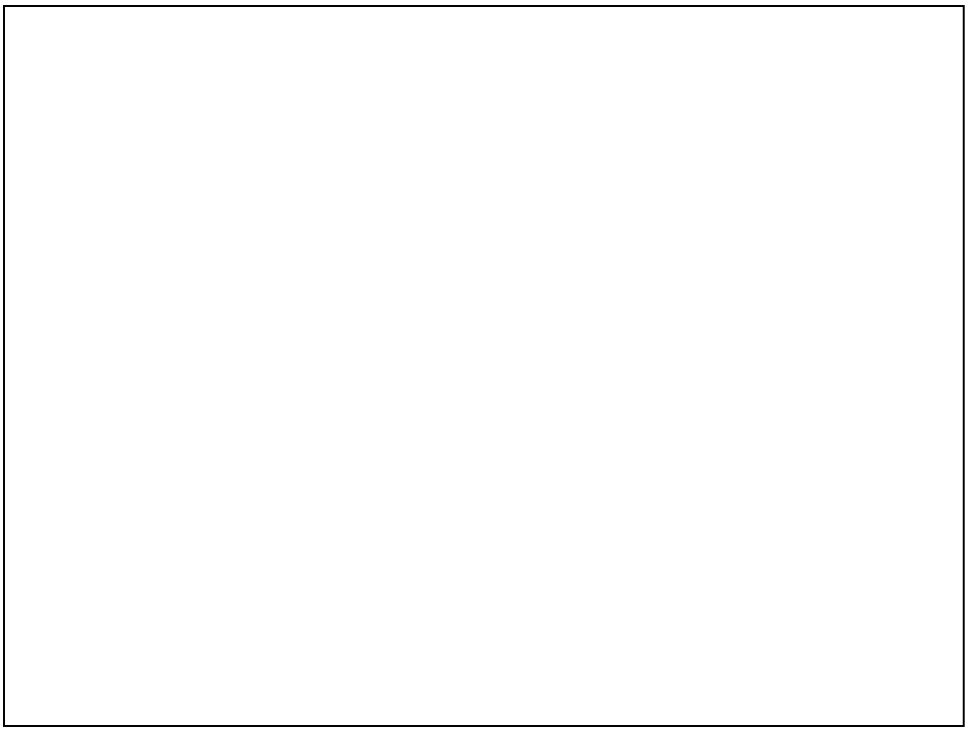
Some of these possibilities were implemented in DB2 v8 for z/OS (and in DB2 v7 for L/U/W)

CTEs, as defined by the SQL-99 standard, were implemented in DB2 v8.



To answer the question, construct (and read) the query inside-out:

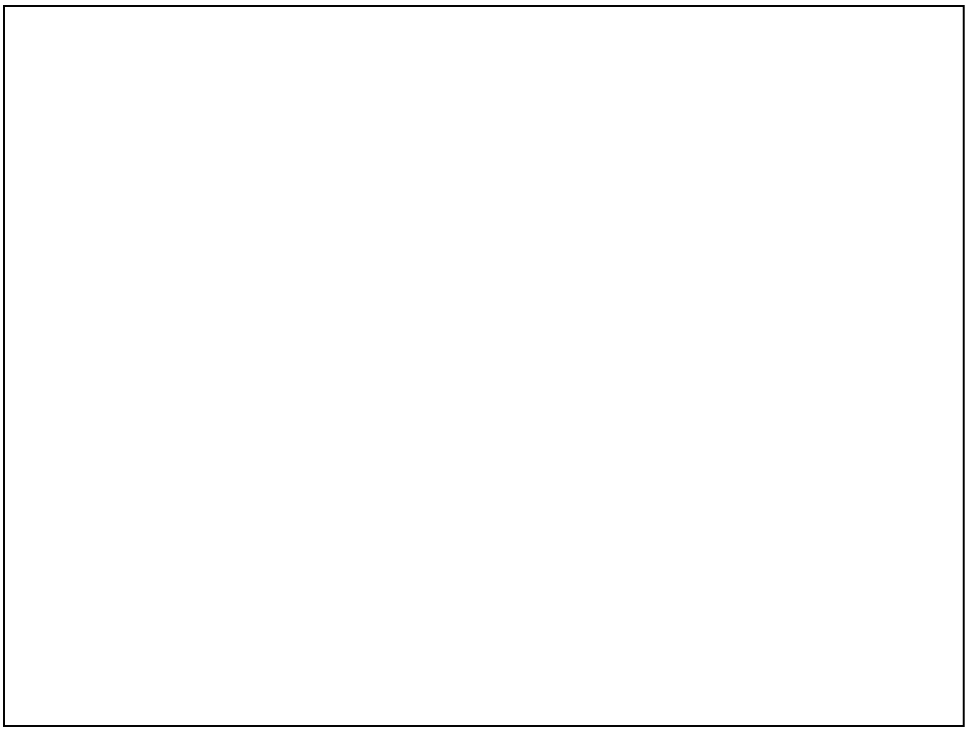
- create a temporary table (actually: a view) of just the products from the orders table, including duplicates, possibly restricted to a certain time period.
- now query this view: group by product name, and count.
- this produces a new view: again, query this one for the largest count
- finally (not on the slide), display the product name for this maximal c



This more “traditional” approach is non-optimal for two reasons:

- maintainability: must repeat the “WHERE orderline IN ...”
- performance: the selection and grouping must be done twice.

Other examples of the “iterative querying” approach would require using JOINs in the “traditional” implementation, where the nested table expression solution is quering individual tables

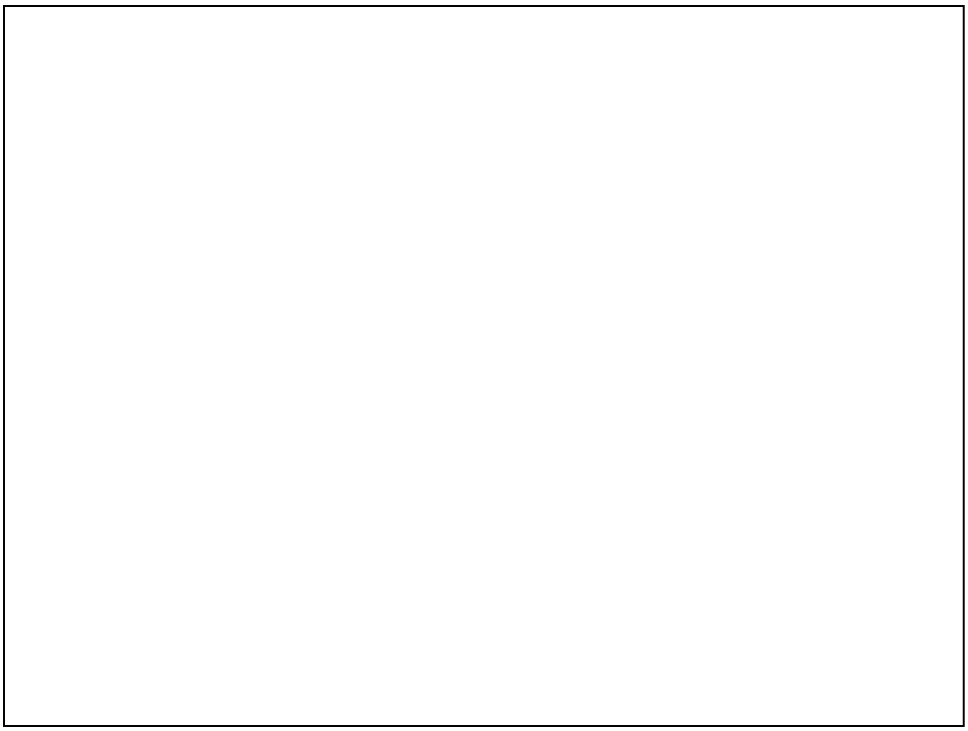


Readability is better than with nested table expressions:

- the “logical” iterative order of constructing temporary tables is also the order of writing down the statements(top-down)
- first the view name, then the definition

Disadvantages:

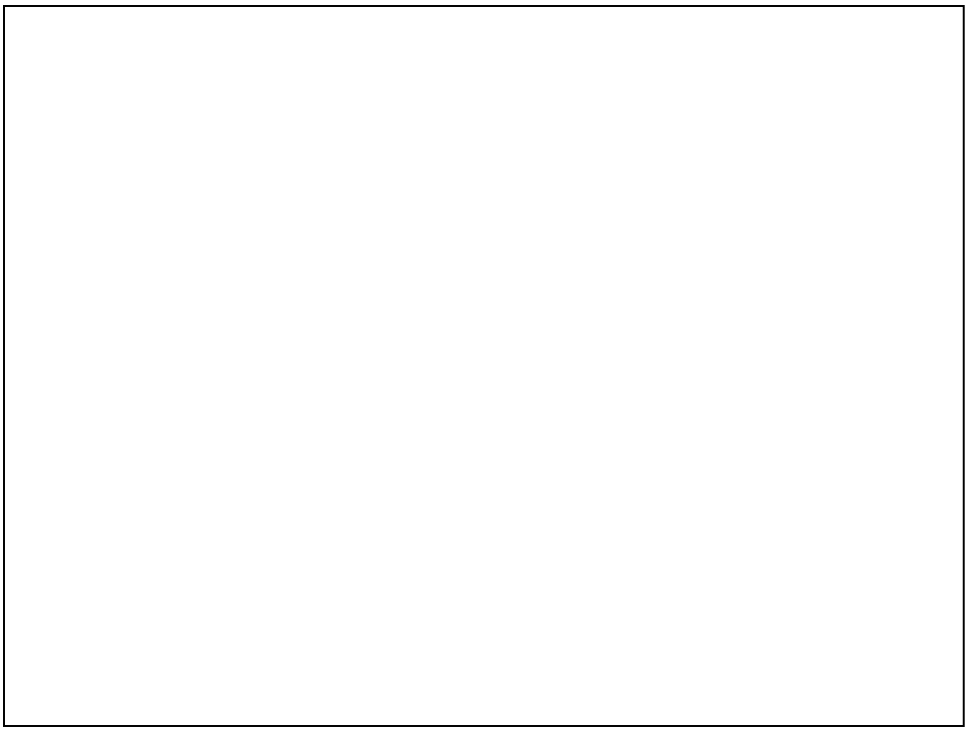
- DDL (visibility, authorization, catalog writes)
- single-use views: not very typical!



Syntactically very similar to the “create view” solution.
Only: no DDL, hence no catalog visibility & catalog write delay.

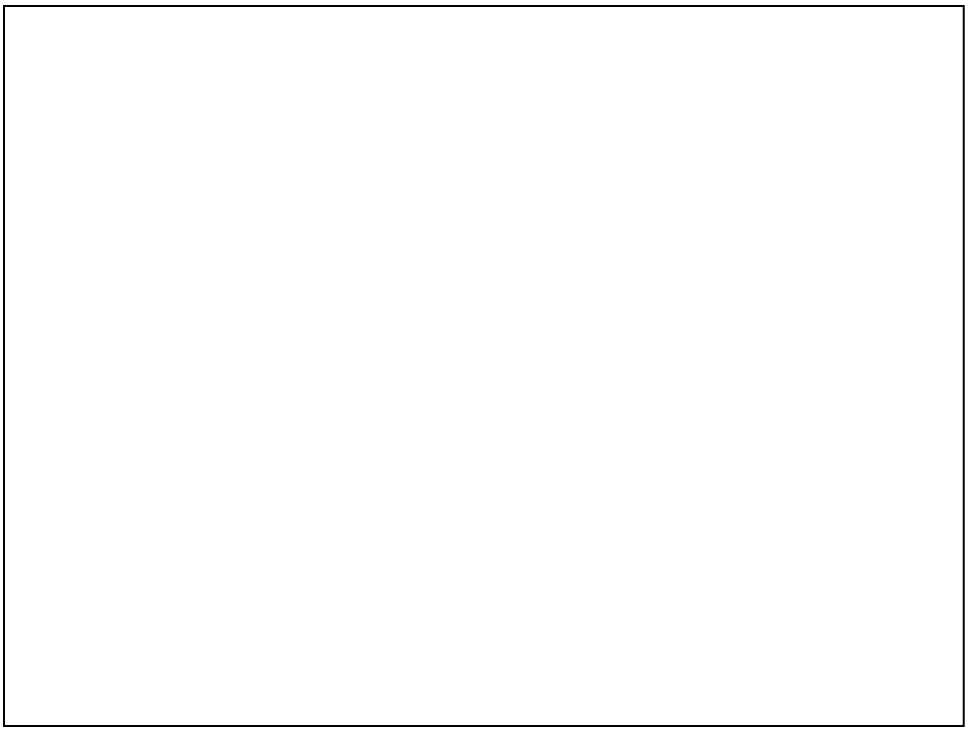
For the optimizer, there is no difference between the three solutions.

Note:
DB2 v8 for z/OS allows CTEs in SELECT, INSERT, and CREATE VIEW.



No reason anymore to use
- temporary views
- temporary tables
- nested table expressions

CTEs replace any of these, without any additional disadvantage



The “alias” name of a CTE comes **before** its implementation.

This opens the syntactic possibility of self-reference
(which a nested table expression or a view does not allow).

DB2 limits this possibility to an **iteration**:

CTE definition with three required ingredients:

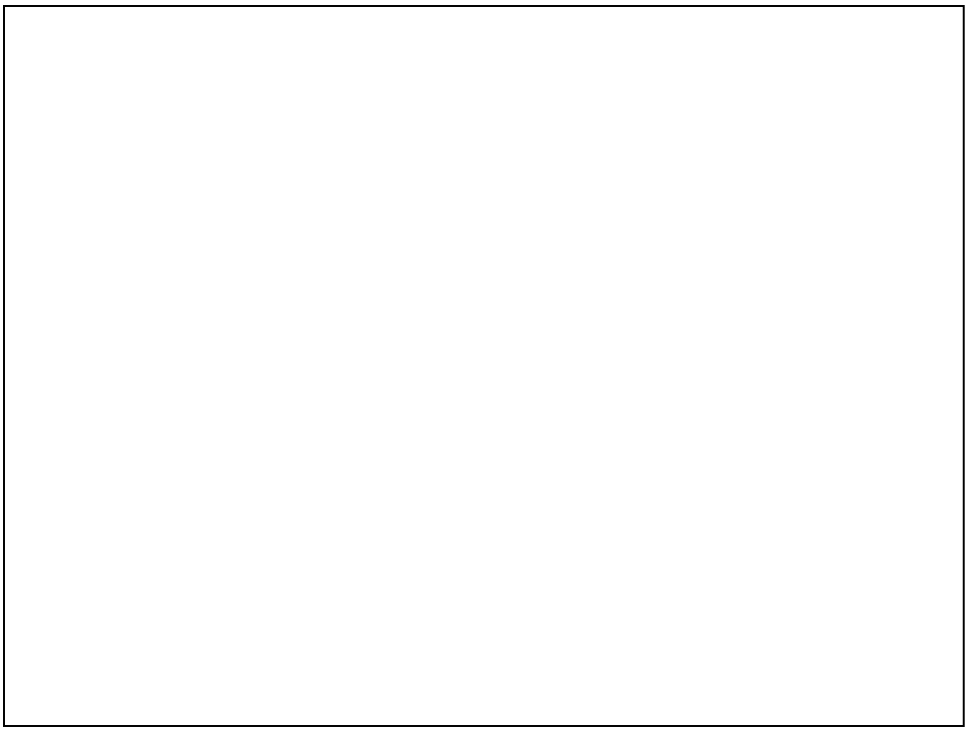
- UNION ALL of two parts

- first part is the iteration “initialization” (without self-reference)

- second part expresses a single iteration step (with self-reference)

The iteration step **must** include a stop criterion to avoid an endless iteration

The reference to v in the iteration step is actually just a reference to the **previous iteration step**, not the table v “up to now”!

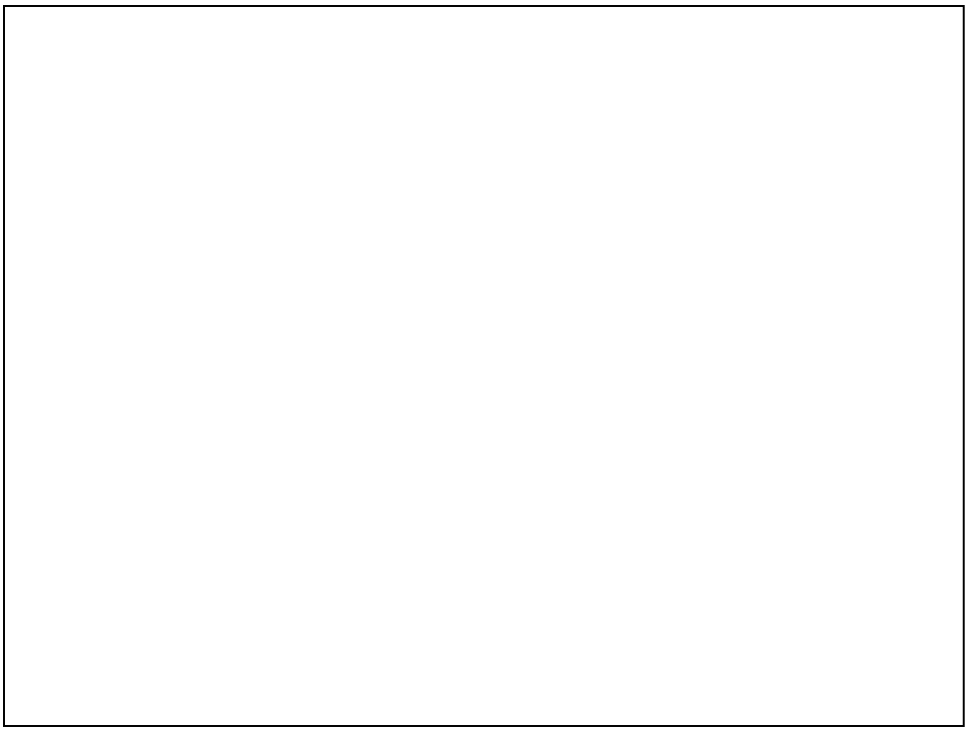


Iteratively create a 1-column temporary table “out of the blue”:
- initial row: the value “1”
- iteration step: add a row with a value “n+1”, i.e., one more than the previous row
- stop criterion: only add “n+1” as long as n < 100.

The resulting table has 100 rows: 1, 2, 3, ..., 100.

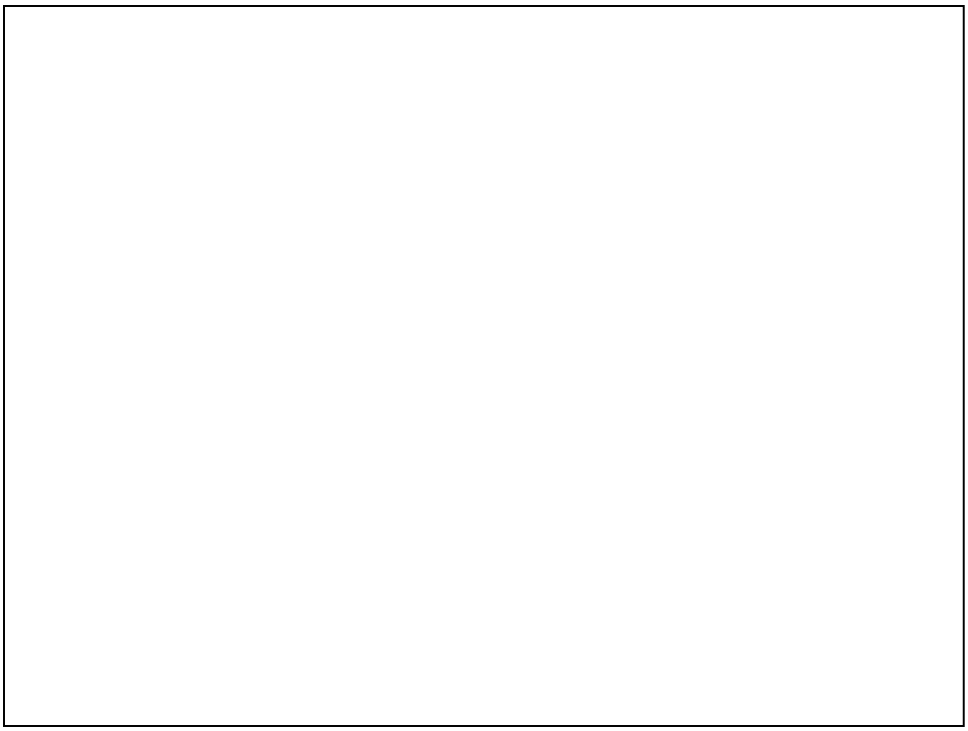
This “table” / “view” may then be used wherever a table or view is allowed:
- in a subsequent CTE definition
- in the main SELECT statement:
  - joined with other tables
  - inside a subquery
  - ...

Other “recursive SQL” implementations (other than DB2) require use of keyword RECURSIVE before CTE name.



VALUES (..., ...) , (...,...) creates an “explicit” table “out of the blue”.
With DB2 for z/OS <= v8, only the limited use with INSERT INTO is provided;
DB2 for L/U/W & DB2 9 for z/OS allow “VALUES” wherever SELECT is allowed.

Workaround: use SYSIBM.SYSDUMMY1, a 1x1 catalog table.



Only in the very limited case of

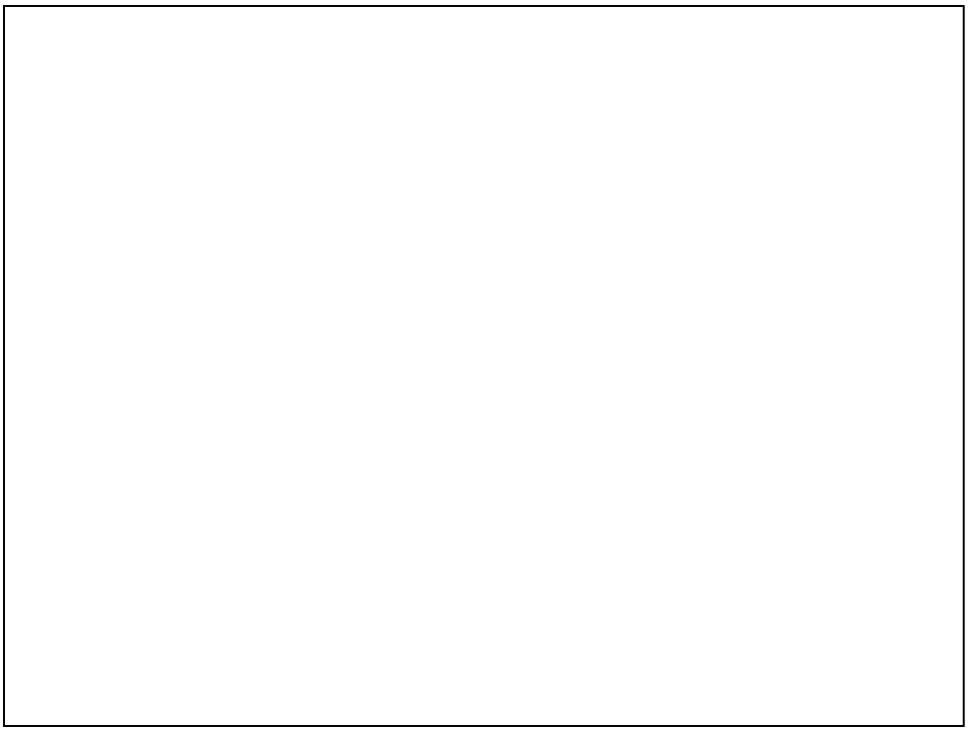
- iteration with “col + 1” for some numeric column of the CTE
- stop criterion with “WHERE col < ...” for that same column

DB2 detects a guaranteed terminating iteration, and returns SQLCODE = 0.

In all other cases, SQLCODE=347 is returned.

Problem:

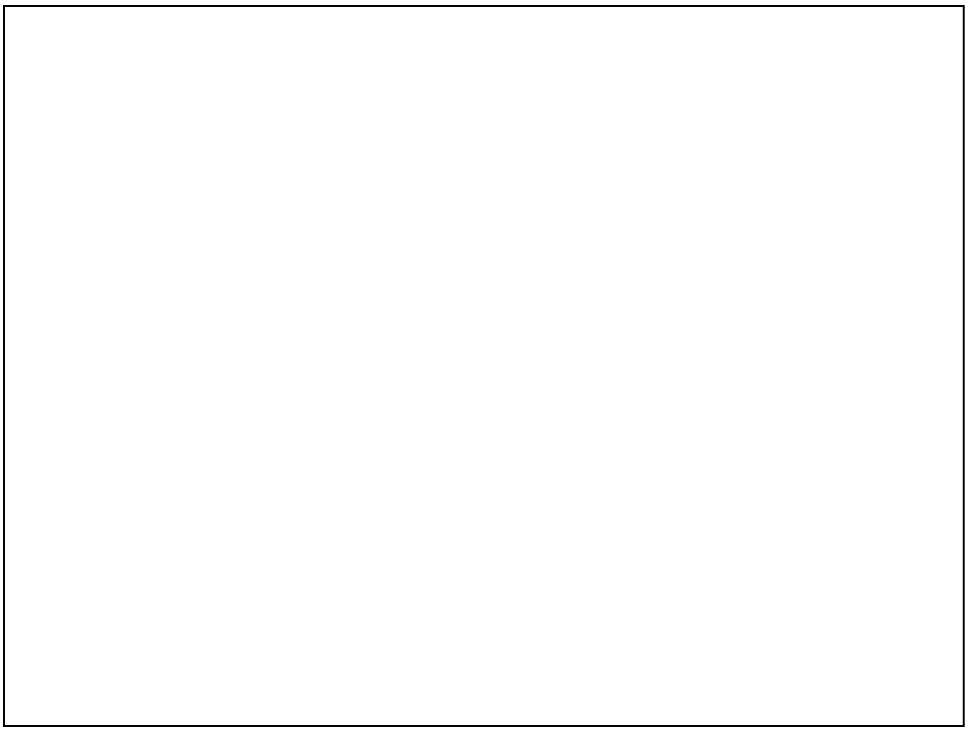
- must catch this value explicitly in embedded SQL
- SQLCODE=347 might indicate a real problem ==> timeout, but CPU consumed



The optimizer can choose to rewrite a query:

- nested table expressions, views, or CTEs may be “optimized away”
- they may require materialization
  - because of sorting
  - because of a UNION
  - because too large for in-memory handling

Any of these choices reflect in the EXPLAIN output in the PLAN_TABLE.



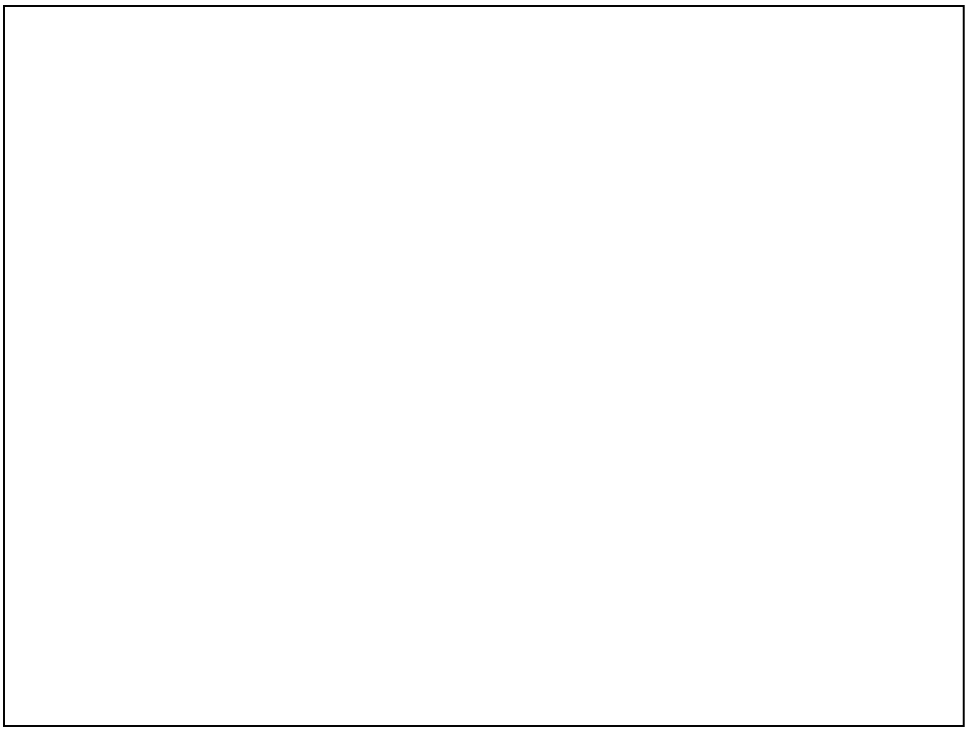
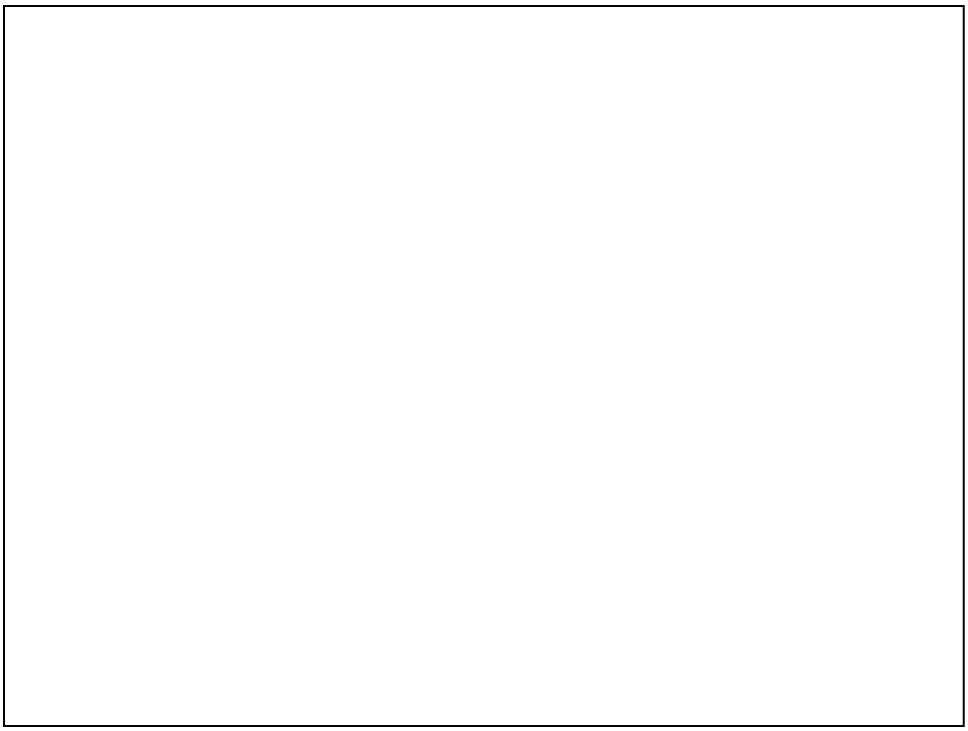
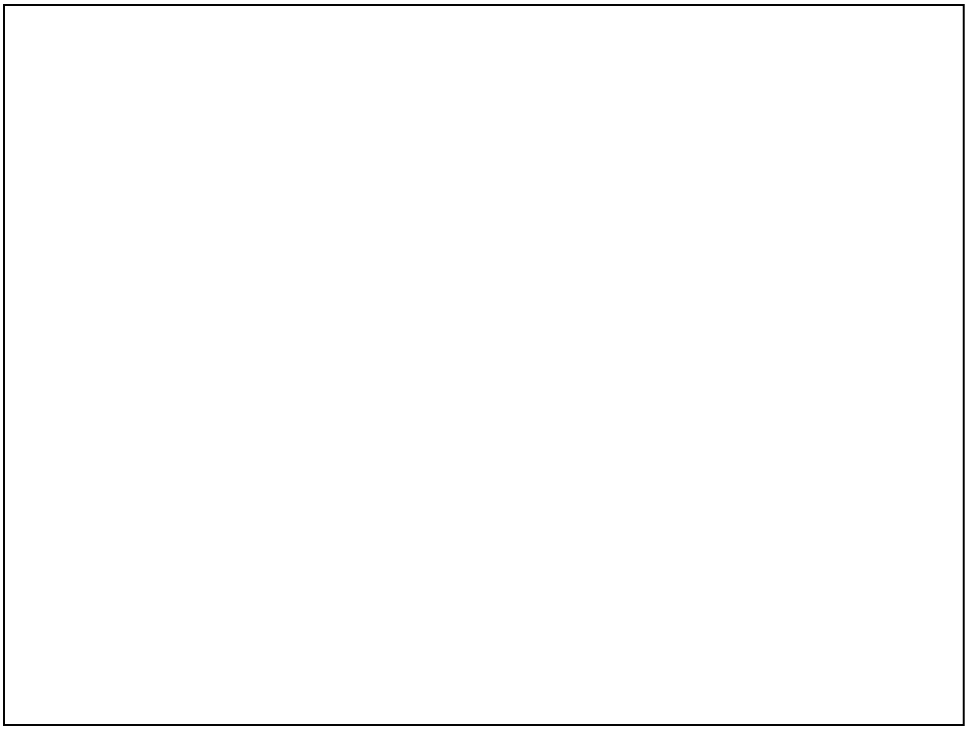


Table “birth_list” contains the first names of all babies born during the last year.
We want to know which was the most popular name.
Or, give a ranking of popular names, limiting this ranking to the top 10 or so.
Using “FETCH FIRST 10 ROWS ONLY” is a DB2-specific solution.
This is often the best choice performancewise, but with two drawbacks:
- not portable outside DB2
- “random” choice of ties, e.g. in case there are two “10th most frequent” names



The EXPLAIN output shows indeed an optimal access path.
When the "`name`" column is indexed, this is even an index-only access.



The second solution adds a generated column, the “ranking number”.

Advantages:

- portable outside DB2
- equal ties are all shown

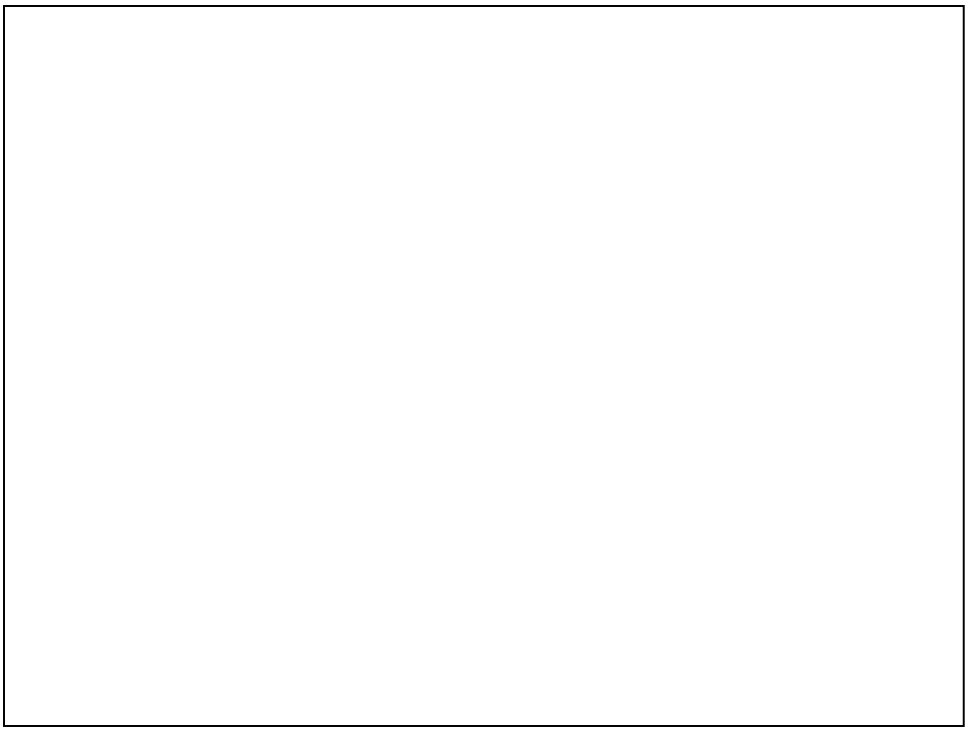
Disadvantage: performance! (see next slide)

Note that the two `COUNT(*)` expressions are counting different things!

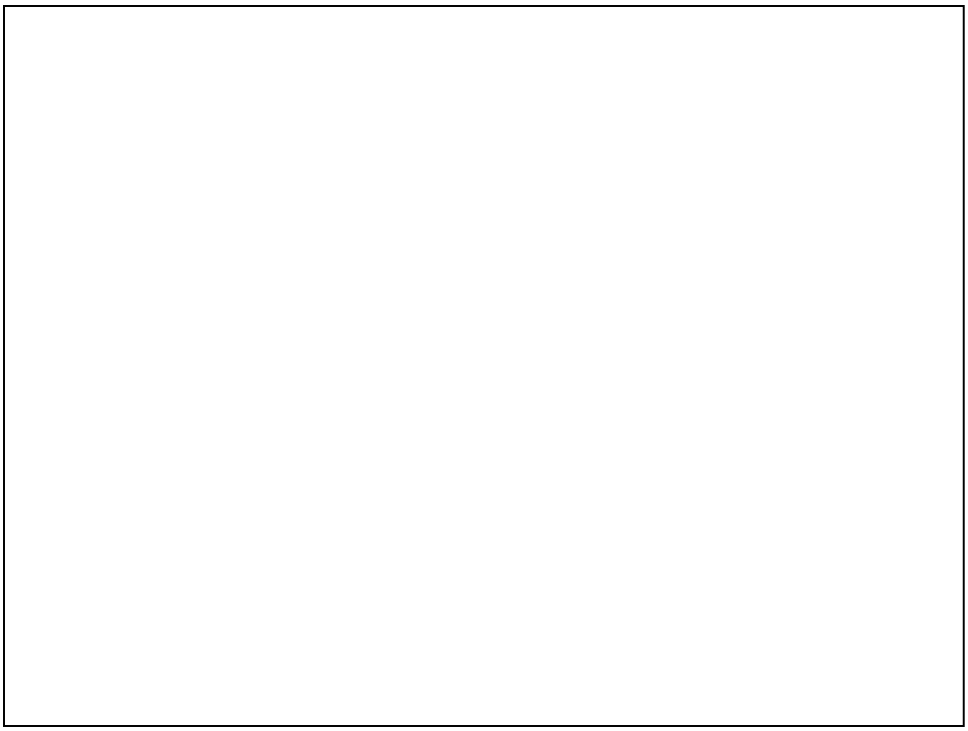
This fact should actually be highlighted by using a second CTE and giving the respective `COUNT(*)` columns informative names.

For example:

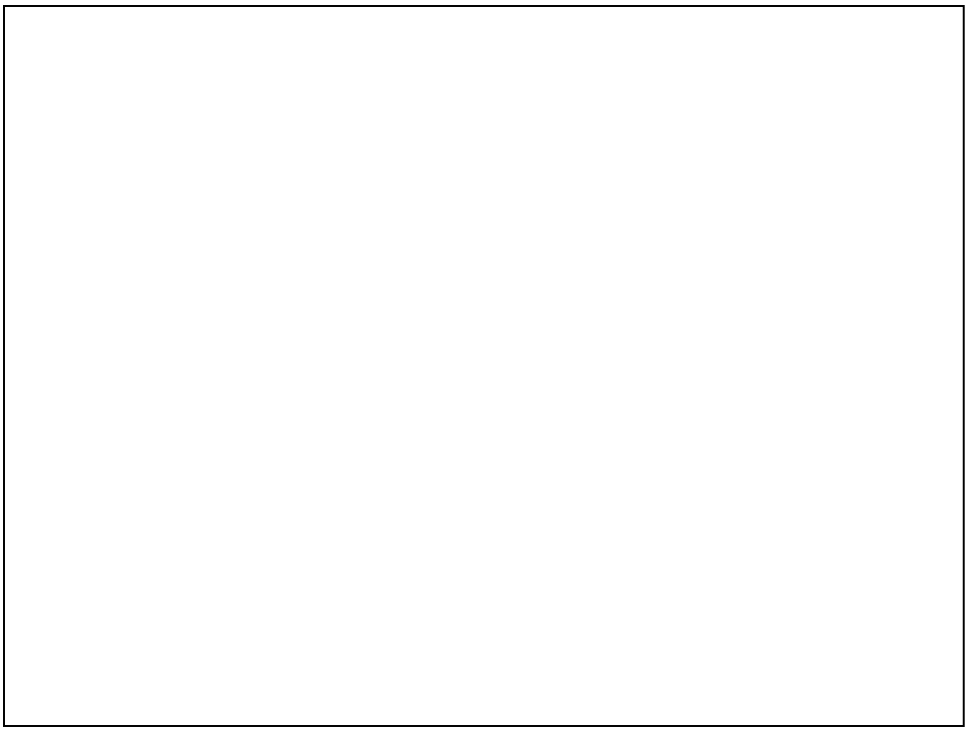
```
WITH p(name, name_count) AS ( SELECT name, COUNT(*)
                              FROM birth_list GROUP BY name)
   , q(name, name_count, ranking) AS
     (SELECT p1.name, p1.name_count, COUNT(*)
      FROM   p AS p1 INNER JOIN p AS p2 ON p1.c <= p2.c
      GROUP BY p1.name, p1.name_count                  )
SELECT name, name_count
FROM   q
WHERE  ranking <= 10
ORDER BY ranking
```



Generating such a “ranking” column requires two table scans of the grouped intermediate result table

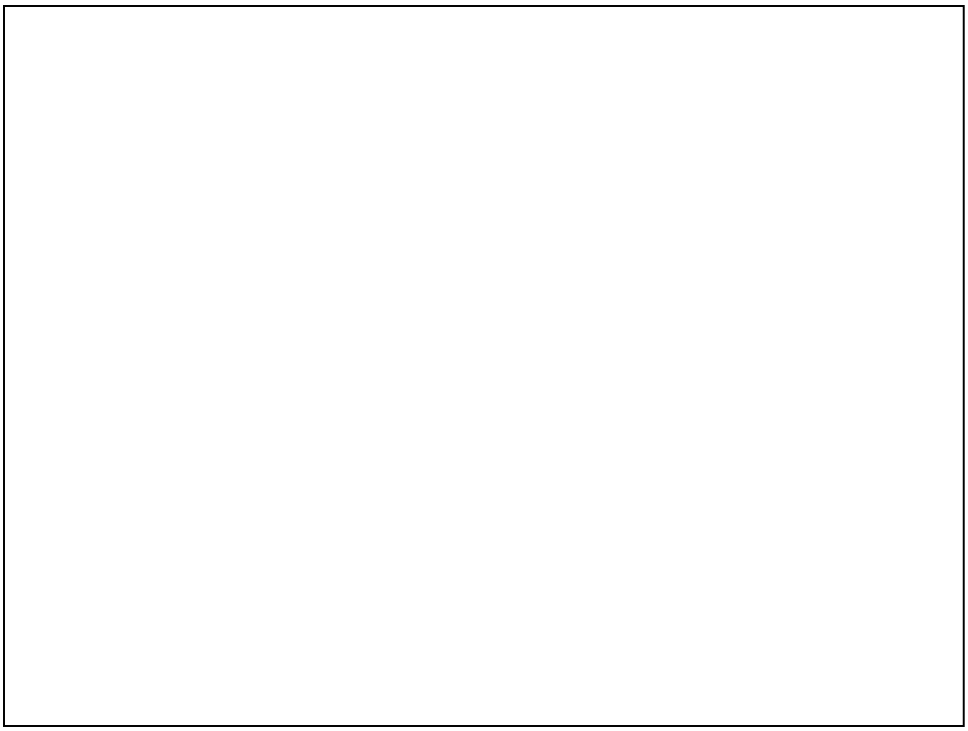


When OLAP functionality is available, a slightly more performant implementation is possible.



Count and group in one table, retrieve maximal count, and use its corresponding value in an other table

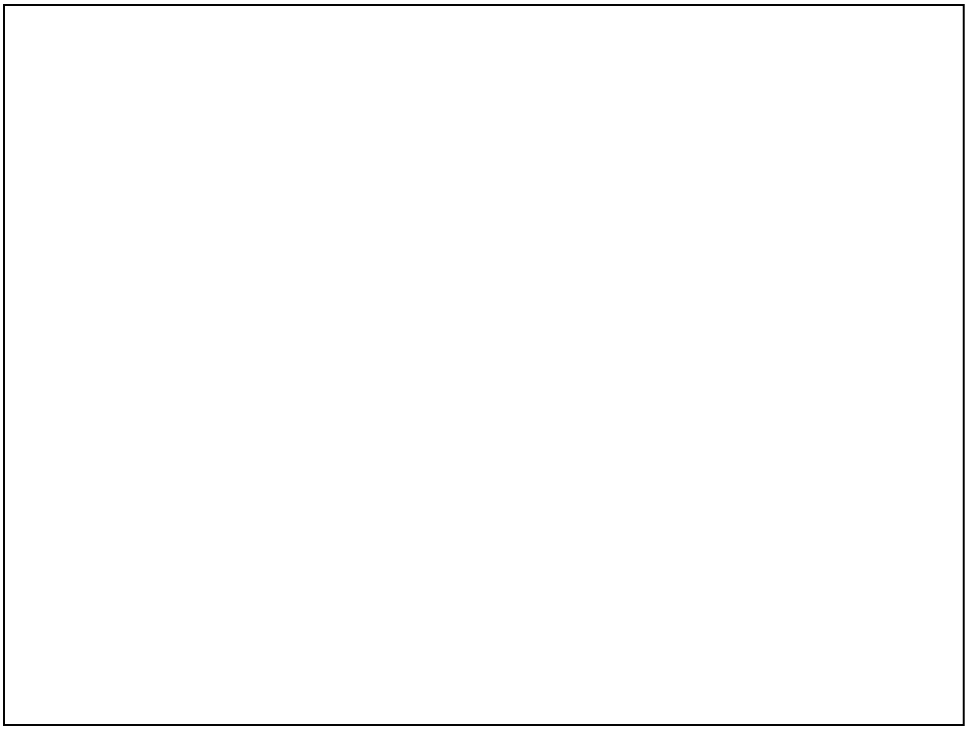
The double grouping (first counting, then taking the maximum) naturally leads to an “iterative” approach, for which CTEs are better suited than subqueries.



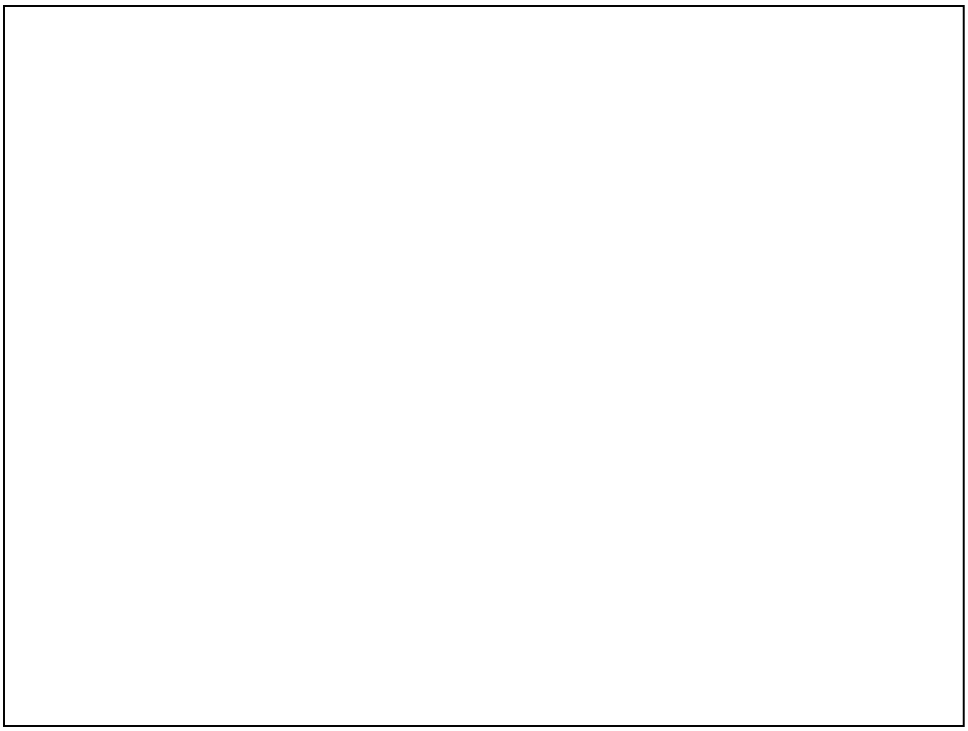
The performance of the “subquery” approach suffers from:

- table scan of the second table
- two scans of the first table

Can this be avoided by using CTEs?

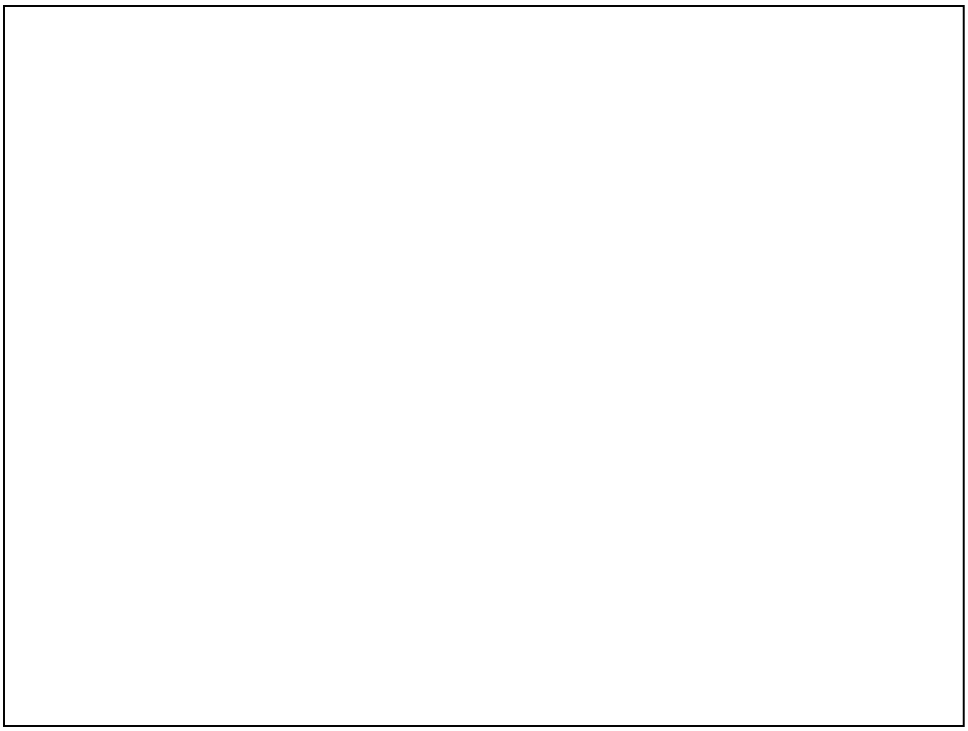


Create two CTEs; representing the two subqueries from the previous solution. Also replace subqueries by joins. This is possibly suboptimal.

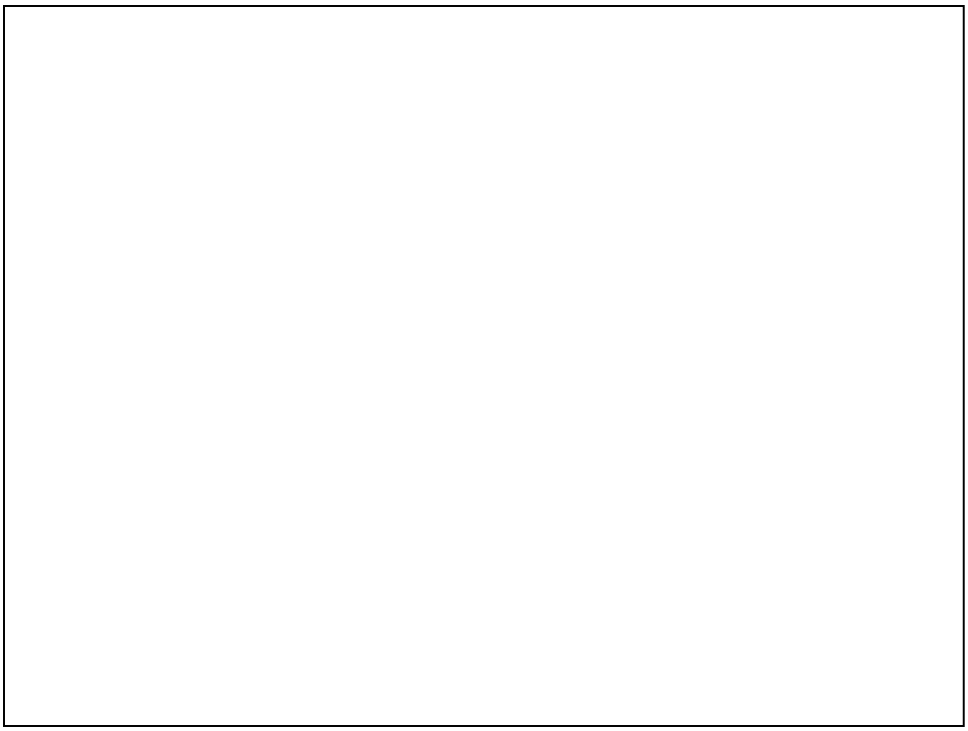


The first table is still accessed two times.

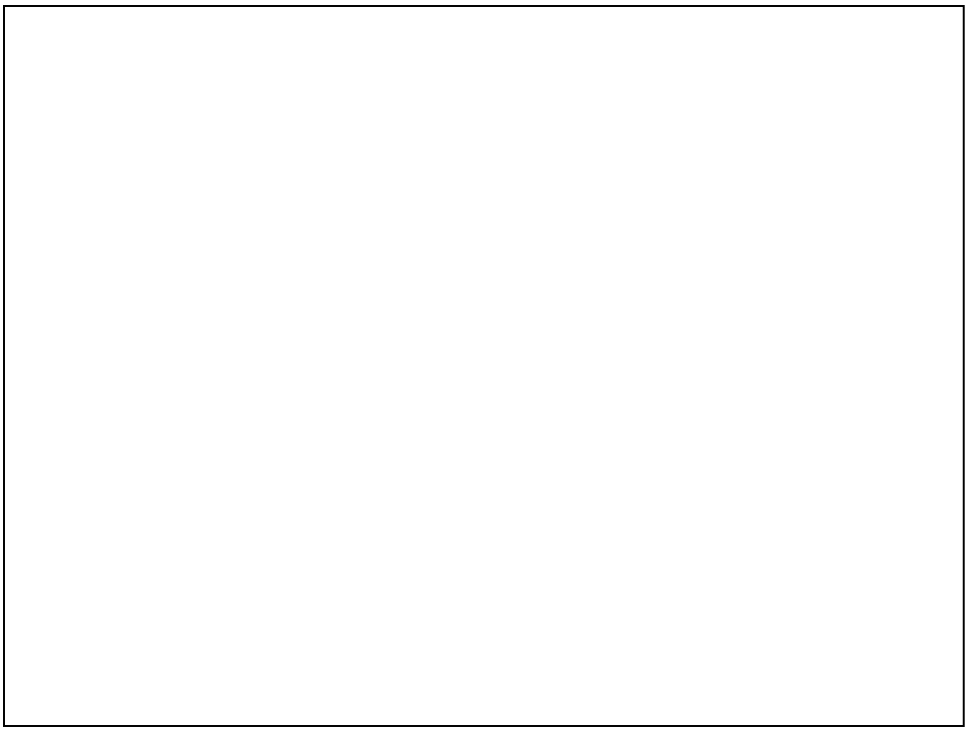
But since q is a 1-row table, the join is implemented very efficiently.



Same CTEs as before, but replace the JOIN by subqueries (as in the first solution)
Readability is better than in the first solution.
Naming of the CTEs can also improve the readability.



Although functionally equivalent to the first solution, performance is much better because the CTE is declared only once, but shared by the two subqueries.



Recursive CTEs are interesting tools for filling holes in tables.

A typical kind of holes are “date” holes.

If a table registers all orders, its easy to generate a list of dates on which at least one order was placed, **but** essentially impossible to generate a list of dates that are **not** present in that table!

Typical solution before the availability of recursive CTEs:

Create a physical table with all possible dates (e.g., between 1900 and 2100).

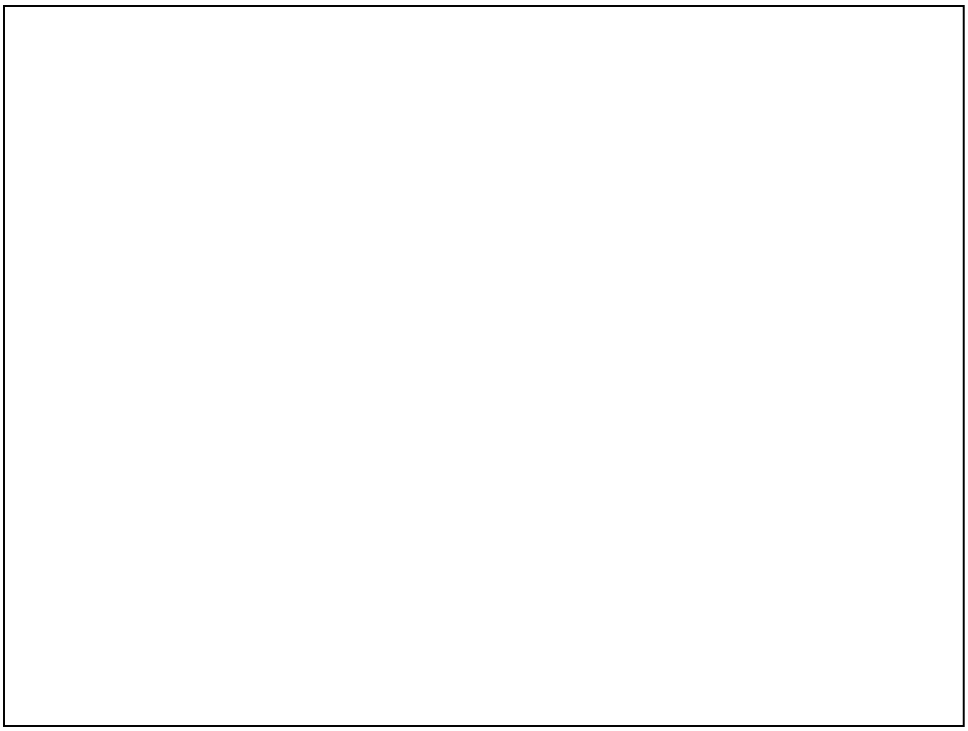
Likewise, tables with all integers (between 0 and 1000000), etc. are needed.

With CTEs, these tables are created “on the fly”, even possibly without materialization.

Note the SQLCODE +347, since the datatype of the generated column is not INT.

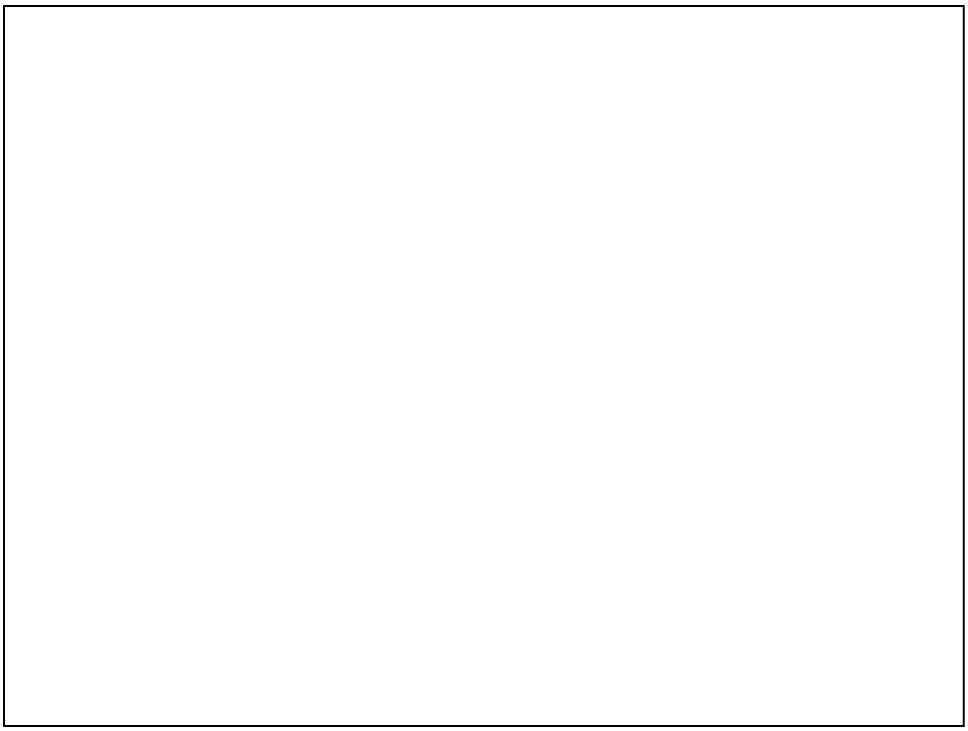
(This may change with newer versions of DB2.)

Also note the CAST in the initialization part of the CTE: this is the only way to set the datatype to anything but INT, VARCHAR or DECIMAL.

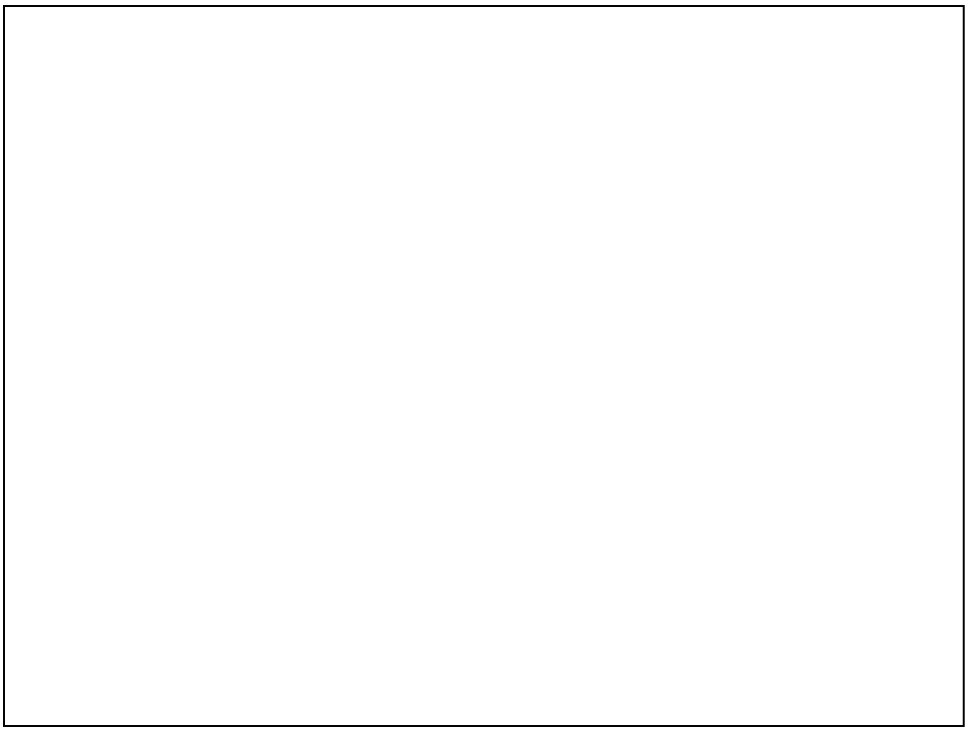


The table scan of the CTE is maybe not optimal, although that table is relatively small.

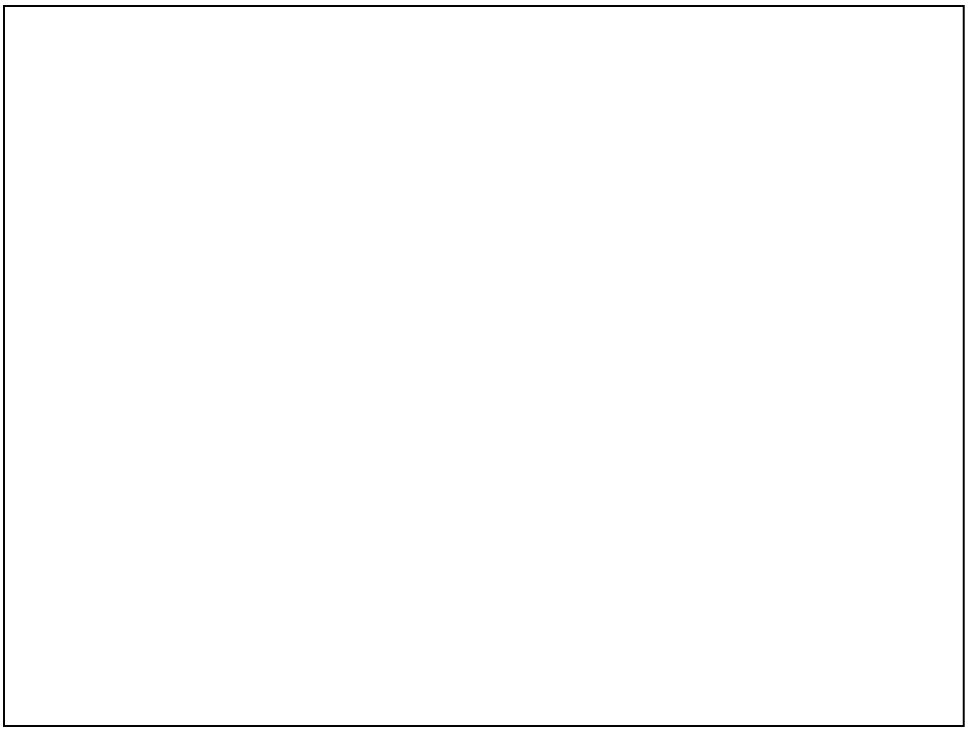
More problematic is the table scan (index-based but with no matching columns) of the base table.



Replacing a “NOT IN” by a “NOT EXISTS” is often a good idea in terms of performance: although logically equivalent, the optimizer never implements these two in the same way.

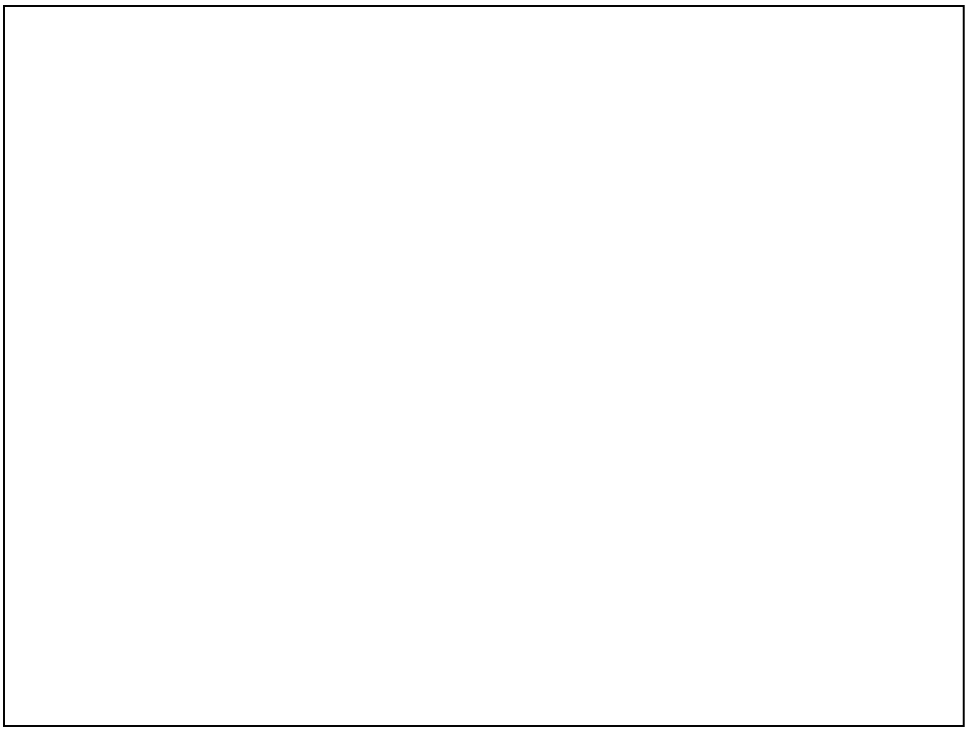


Indeed: no sorted list of dates from the base table needs to be generated.
Of course, a full scan of the base table is still needed, but this is unavoidable.



An alternative to a subquery is a join.

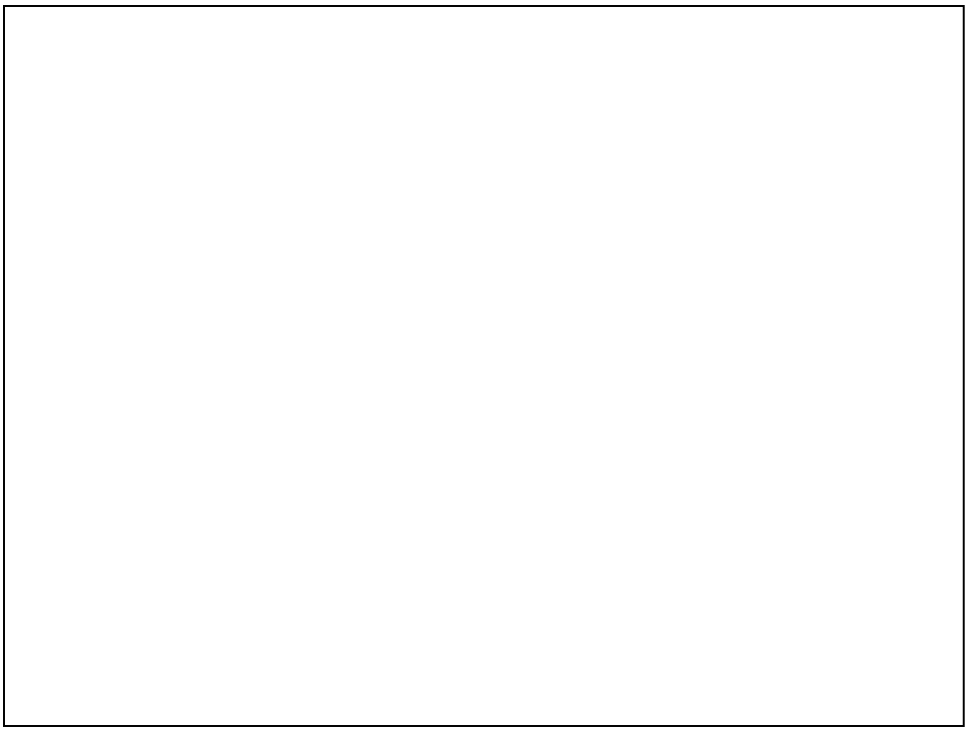
The “not in” part of the story must be translated to an outer join combined with an IS NULL on a column of the base table which is known to be NOT NULL.



We're back to the “sorted” solution, be it that no explicit duplicate removal is needed with a merge scan join.

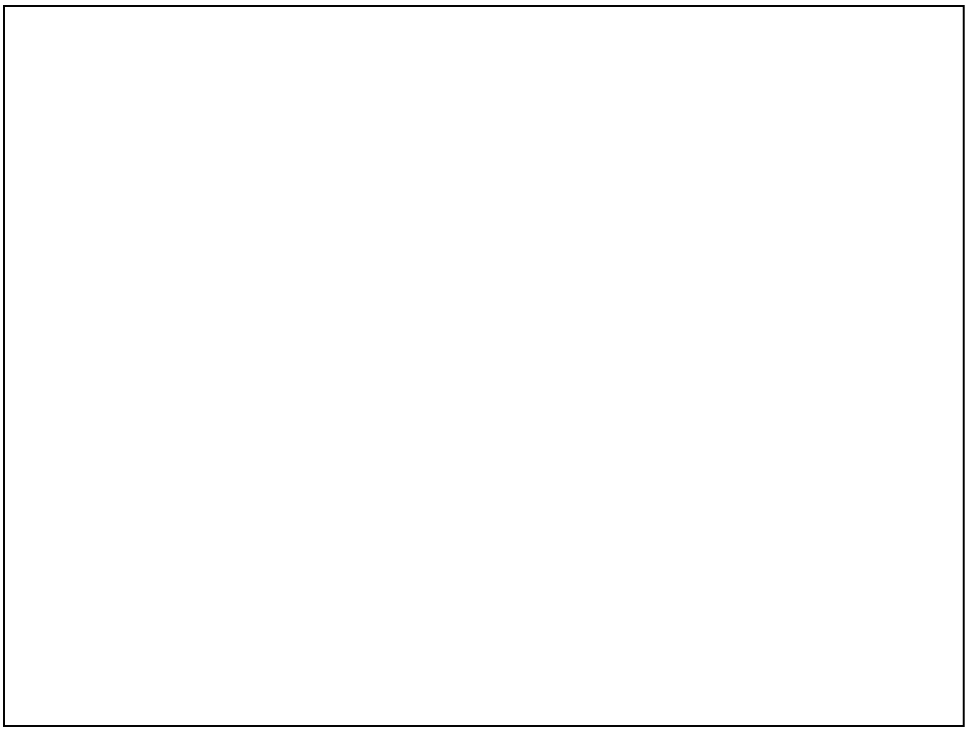
Remarkable fact: the CTE has to be sorted as well, even if that “table” was generated in a sorted way. Apparently, the optimizer “has forgotten” about that.

This is apparently a typical “child disease” of recursive CTEs, probably going to be solved in one of the next releases...



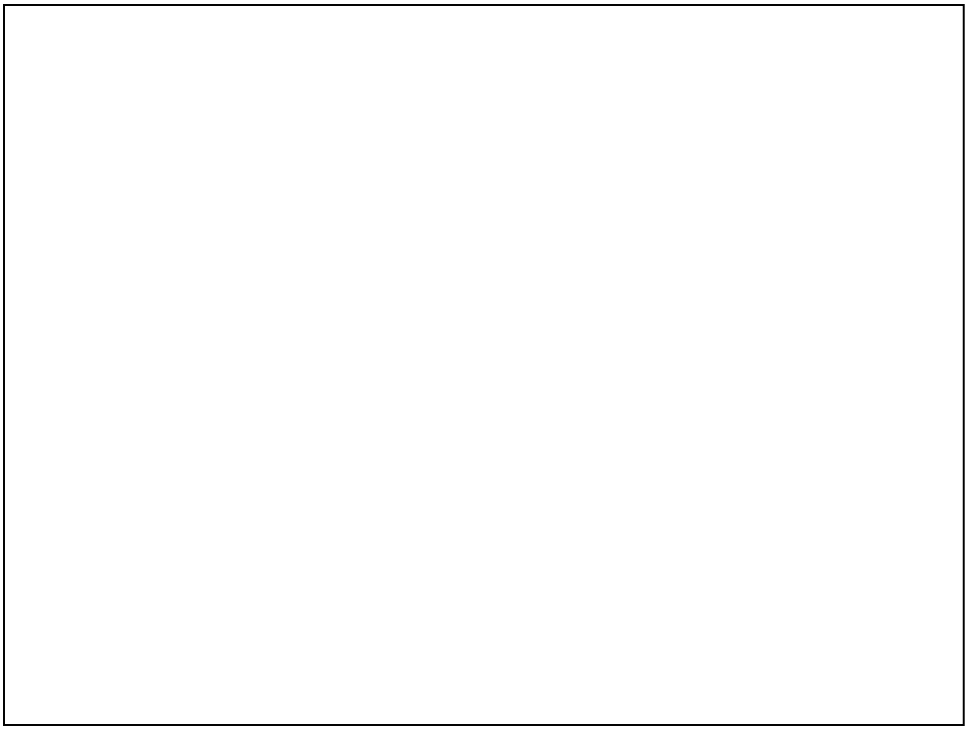
A variant of the previous question: only show the “not present” dates when they are working days.

An additional CTE helps the readability of this additional condition.



And indeed: the additional CTE is not visible in the EXPLAIN output.

So we are essentially back in case (b), with an additional filtering condition on the *dates* CTE

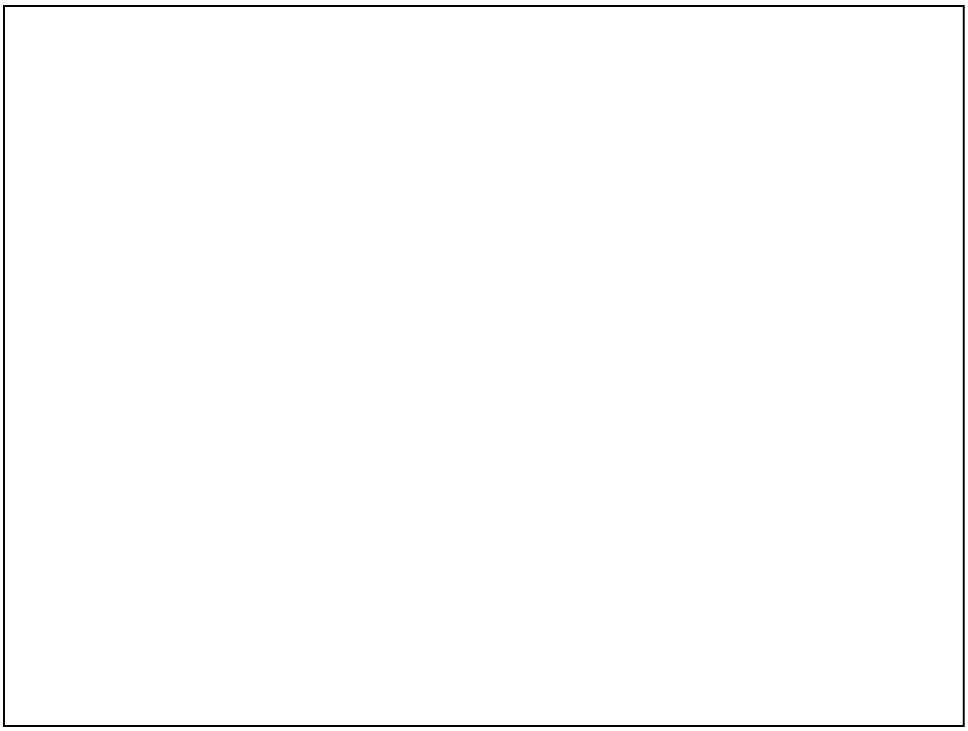


Because of the initial “%” in the LIKE expression, all email entries from the entire table must be scanned: no index based filtering is possible.

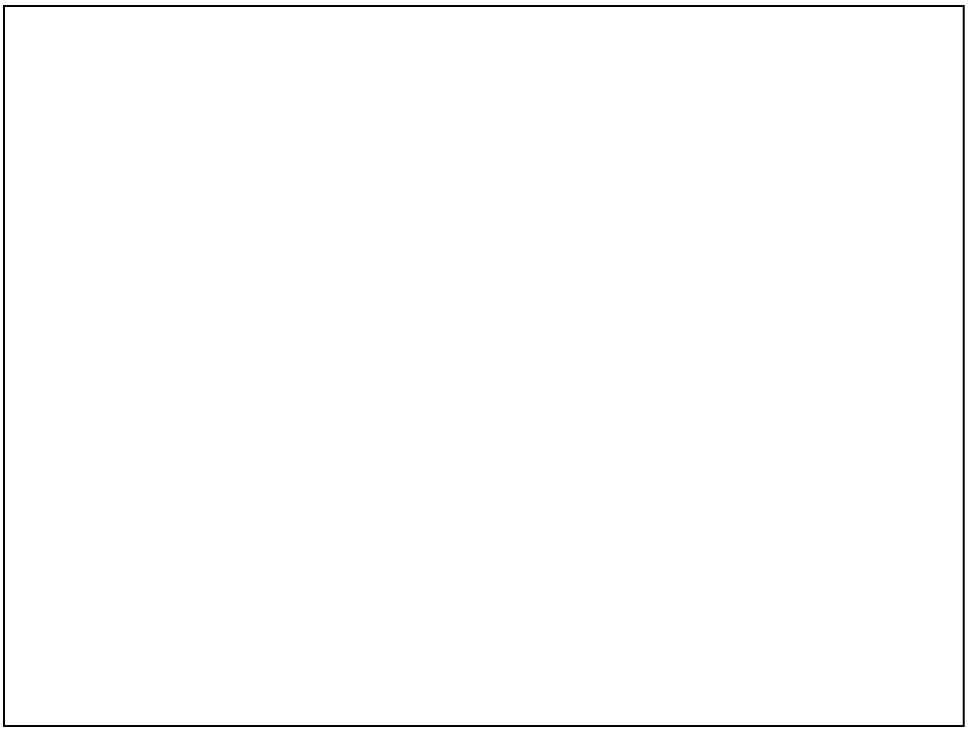
(At least: not before DB2 version 9.)

A similar situation would occur with a stage-2 expression in the WHERE condition, like “WHERE UPPER(first_name) = 'PETER' ”

Moreover, a (non-matching) index-only “screening” is not possible since the person's names are required, not their email addresses.



Is it possible to rewrite this query such that a table scan is avoided?



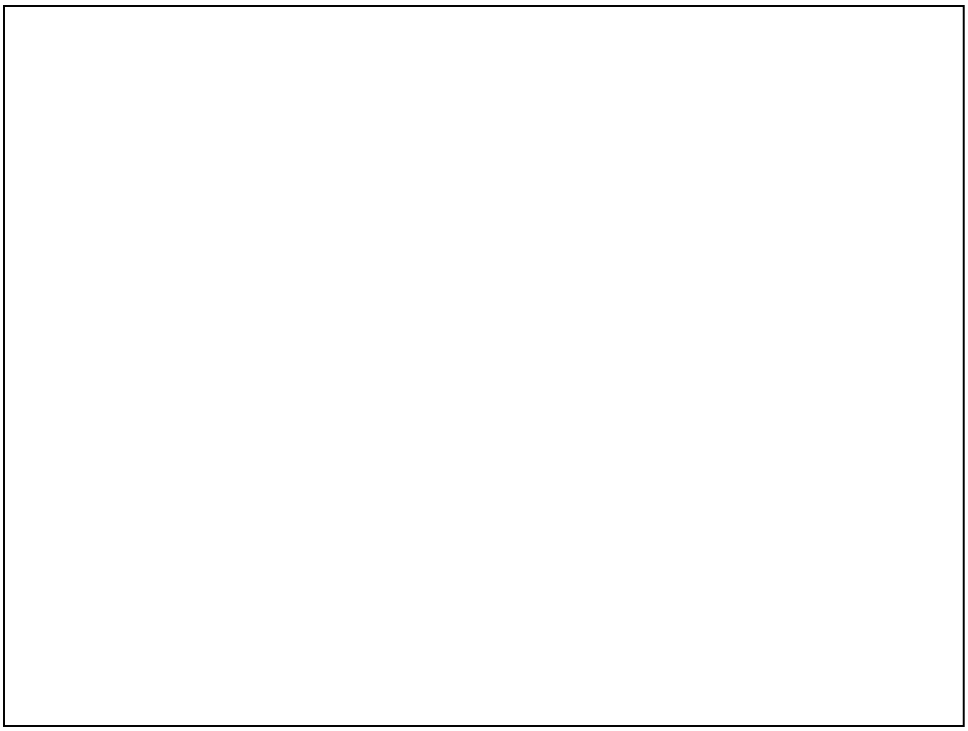
Yes, it is!

Again, the “iterative coding” paradigm helps:

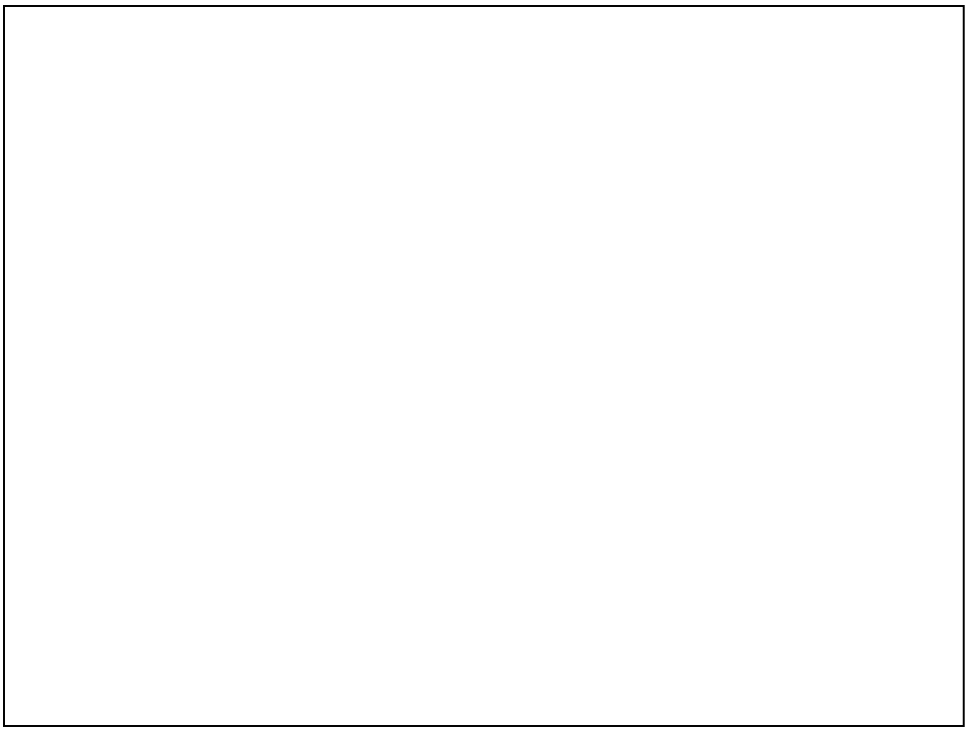
- first, find all email addresses that match. This requires an index-only screening.
- then, only access the **matching** table rows.

Variants to the IN come in mind (like a join or an EXISTS); each of them must be considered and passed through EXPLAIN.

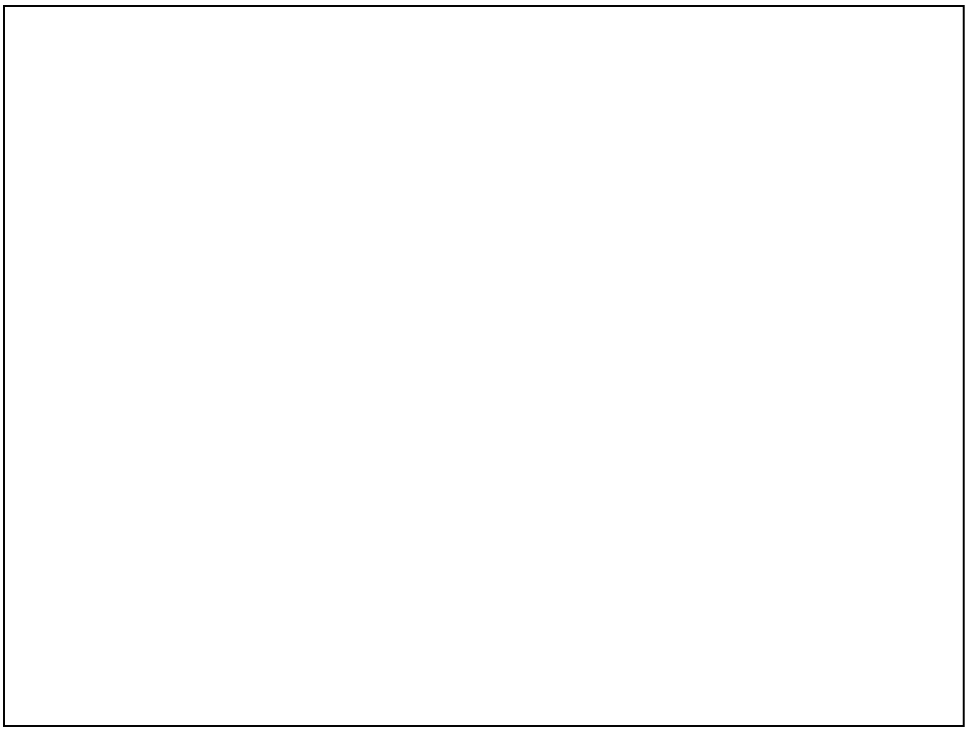
The optimality of one of these depends on the table size and other statistics, as seen by the optimizer.



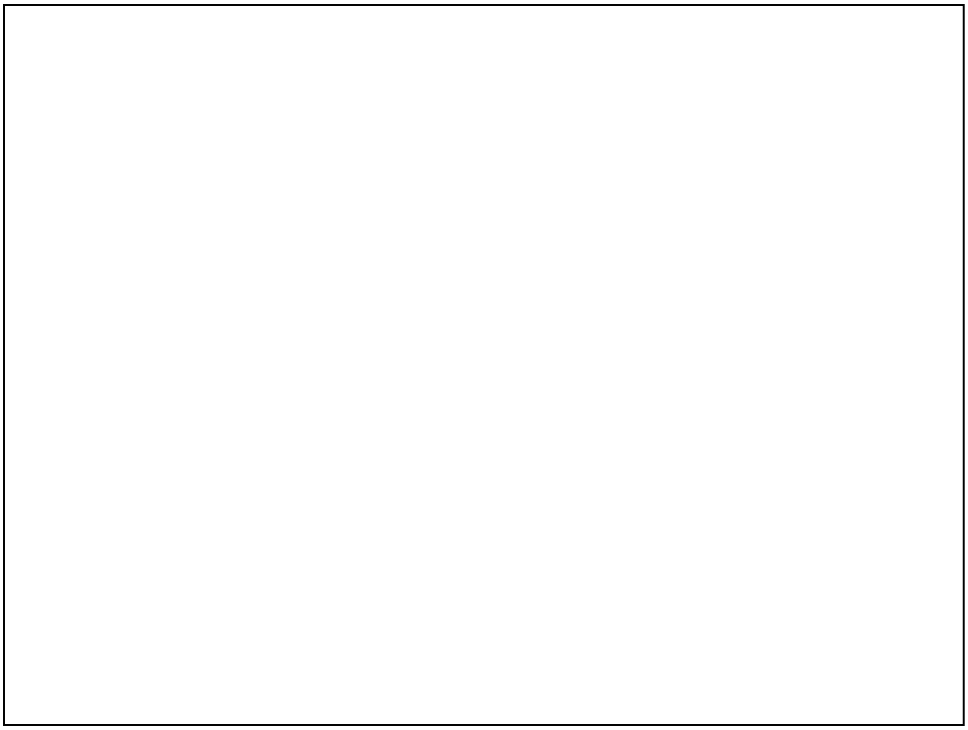
The EXPLAIN output indeed shows an *index-only* index screening, followed by a *matching index* based table access.



Here's the JOIN variant of the previous two-step idea; again, a table scan is avoided.



The nested loop join does essentially the same work as the “N” access type of the previous solution.

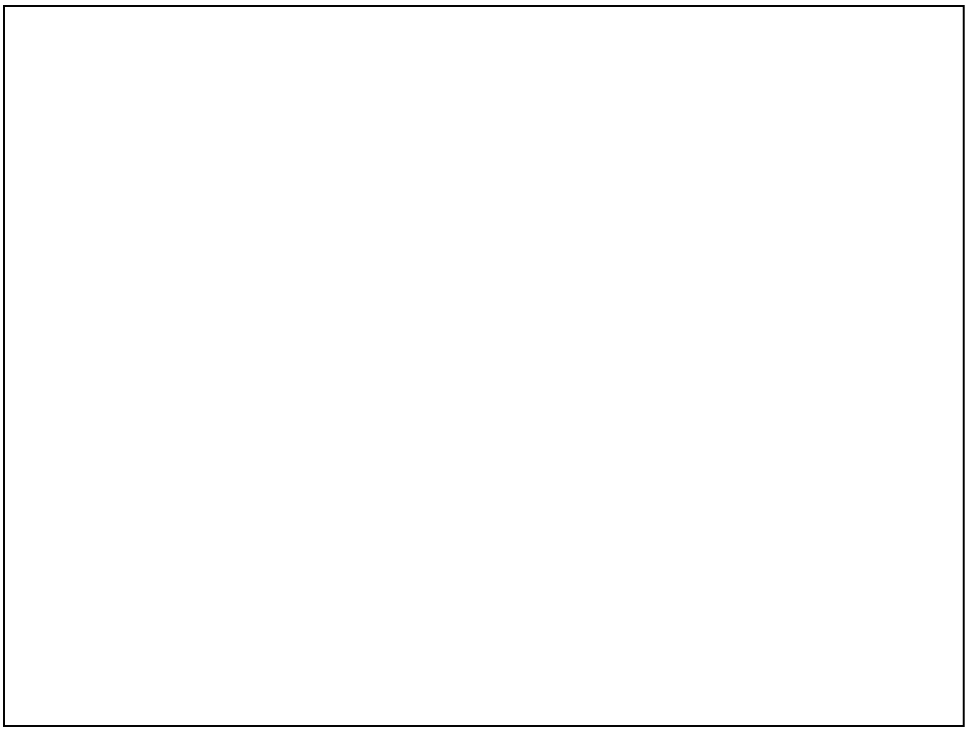


The problem lies in the fact that the range `BETWEEN start_d AND end_d` does not suffice: it should be extended to multiple ranges:

`BETWEEN start_d – 1 YEAR and end_d – 1 YEAR`, etc.

The first attempt tries to "merge" these multiple ranges by mapping all dates into the same year, i.e., to consider dates as being "modulo 1 year".

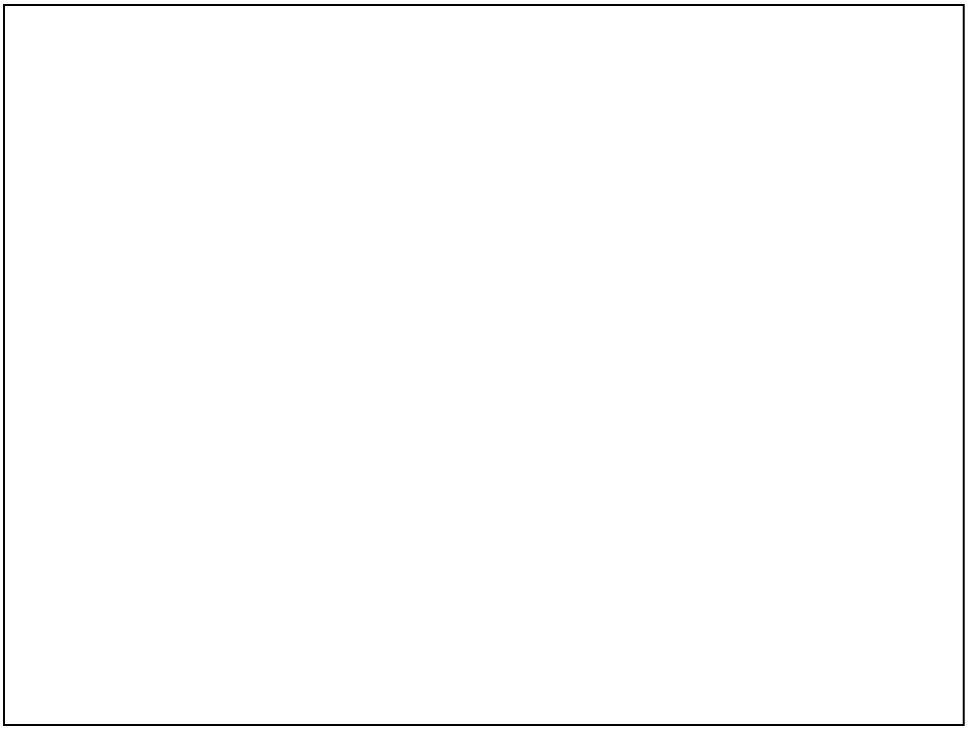
Unfortunately, the day number (1 to 365 or 1 to 366) does not correspond in a 1-to-1 way with the date ...



This leads us to the second attempt, where the (textual version of) the month and day are used.

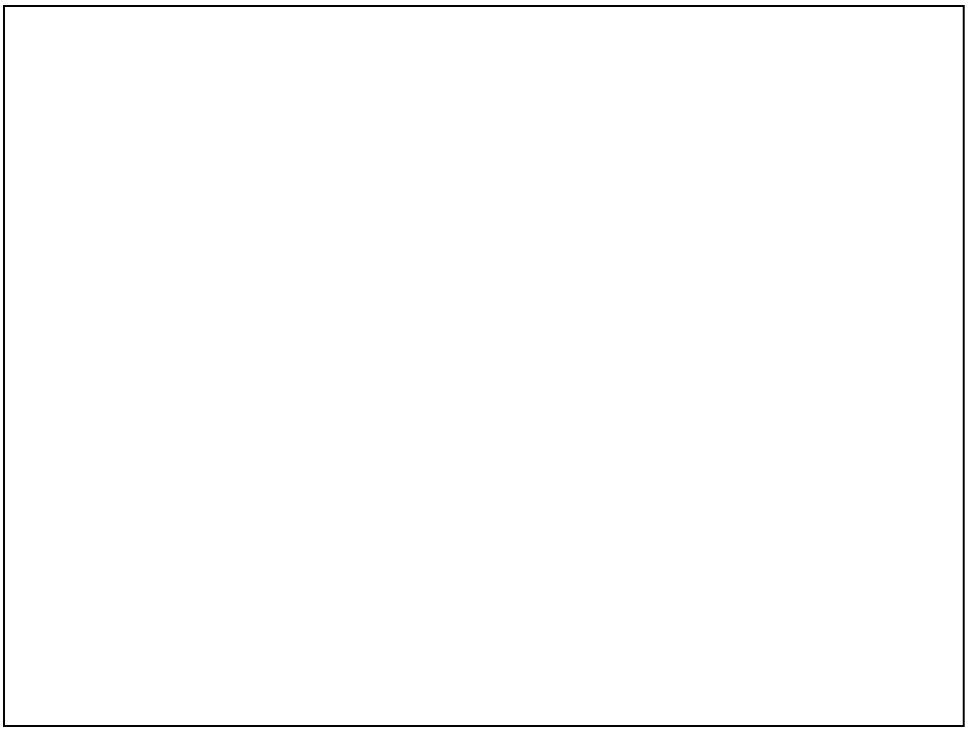
Of course, numeric values cannot be concatenated, so a more accurate form of the query would be:

```
SELECT name
FROM   persons
WHERE  DIGITS(month(date_birth))||DIGITS(day(date_birth))
       BETWEEN
       DIGITS(month(start_d))||DIGITS(day(start_d))
       AND
       DIGITS(month(end_d))||DIGITS(day(start_d))
```



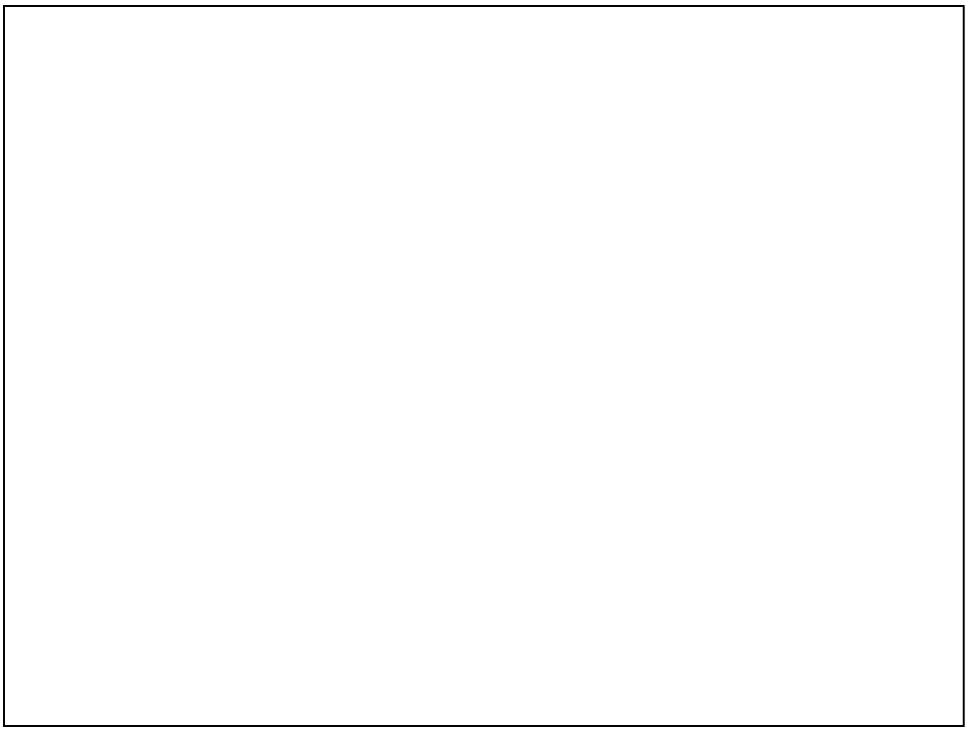
A date range crossing year boundaries will not work with the previous approach.

This special case could be corrected for by subtracting `YEAR(end_d)-YEAR(start_d)` from the date, making the query even more difficult to understand...



Let's start over again, turning around the question:
instead of trying to write out **all** possible **date ranges** for the matching birth dates, try to generate all birthday dates and catch those within the original **single range**!

The way to generate a list of birthday dates is of course through an iterative CTE.



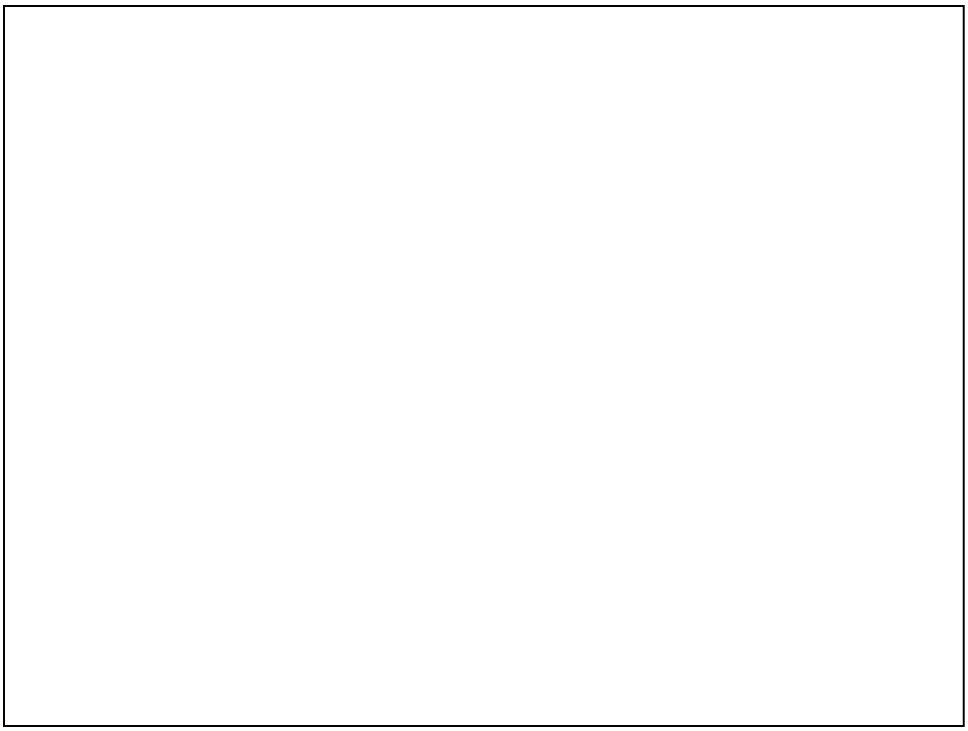
This is a very common request: can I display the results of a “group by” action without loosing the individual contents of the rows?

Of course this is not possible, but still... what about placing the grouped rows on a single row, as columns? I.e., essentially **pivoting** the base table.

Problem with that question in general is that the resulting table has a potentially unlimited number of columns, which SQL does not allow.

A variant (and often equally acceptable) solution is to produce just one column which is the **concatenation** of a certain field for all rows of the group.

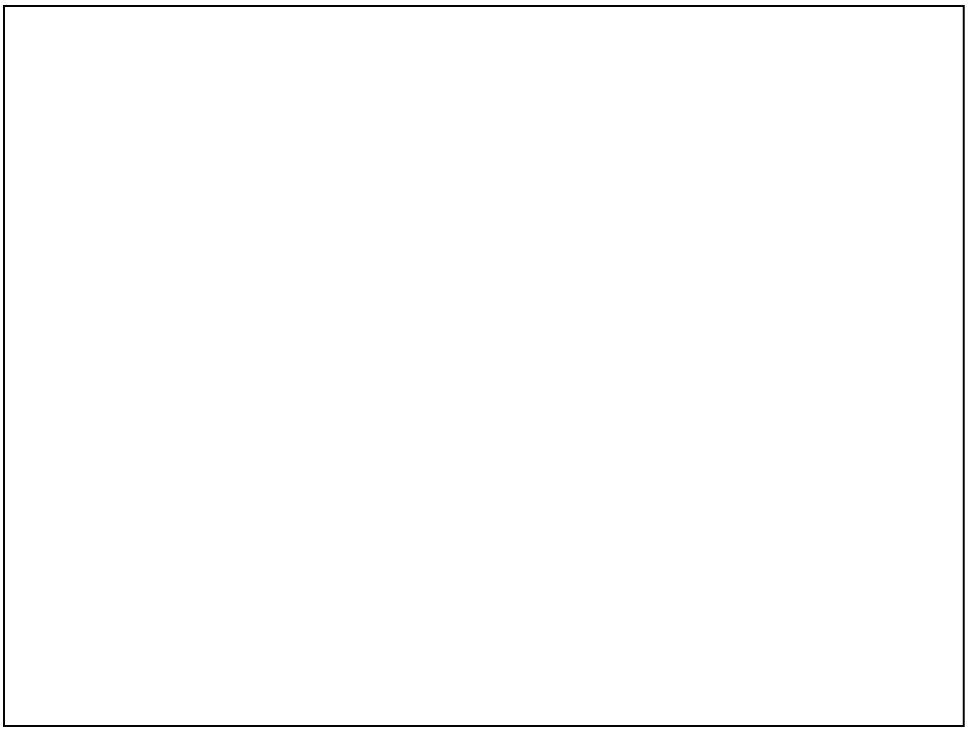
For this to work in general (without a limit on the number of rows per group), we have to resort to recursive CTEs.

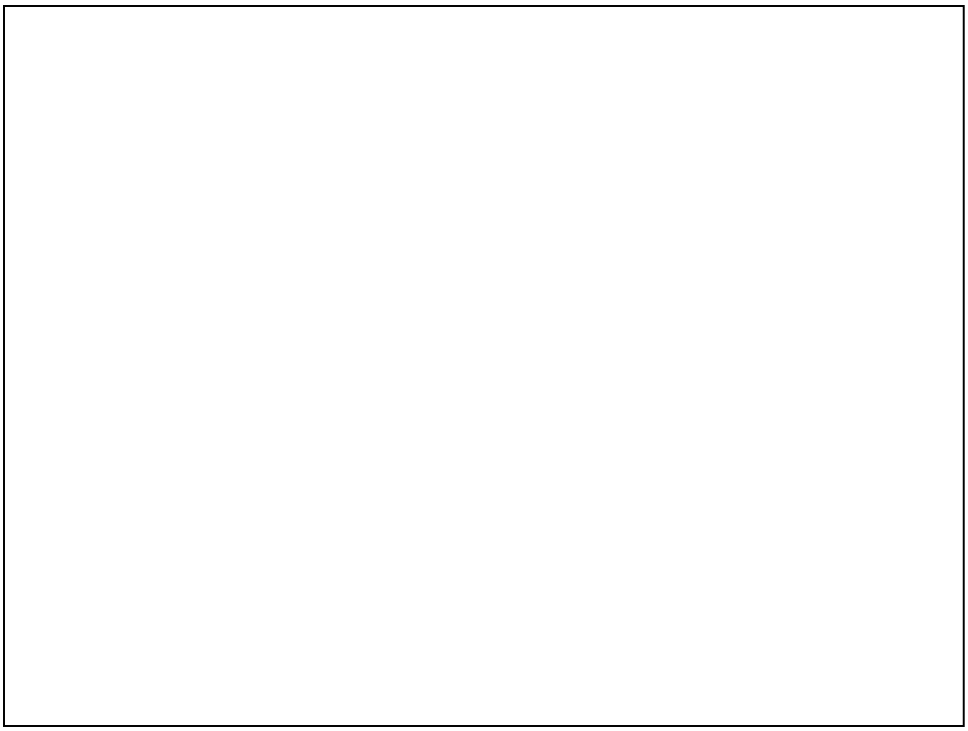


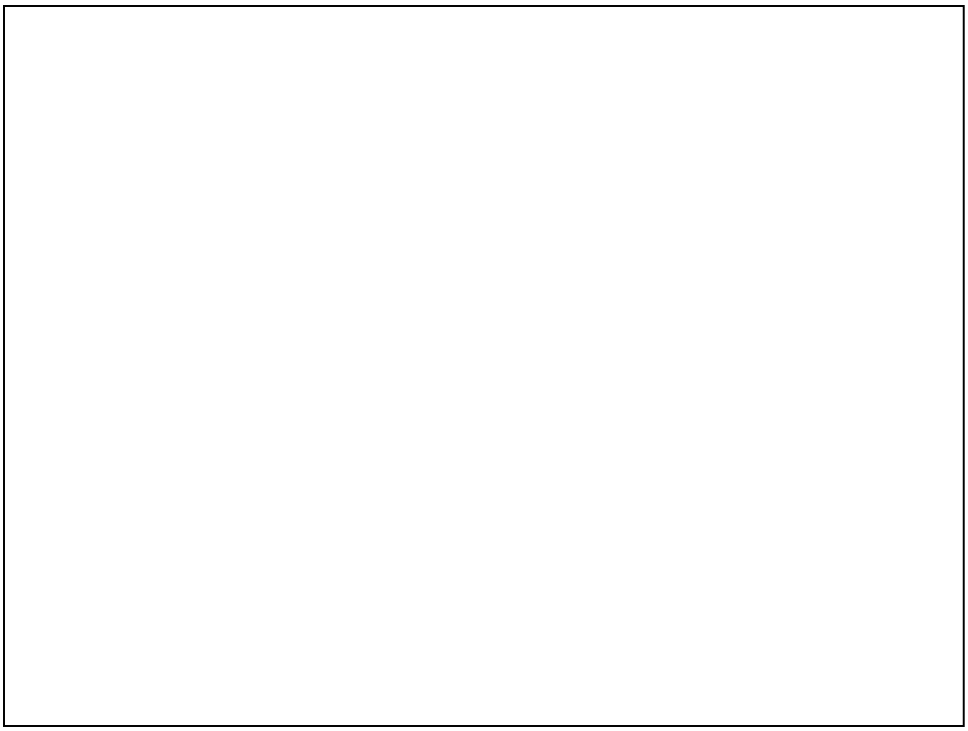
This would be for the case of two rows per group.

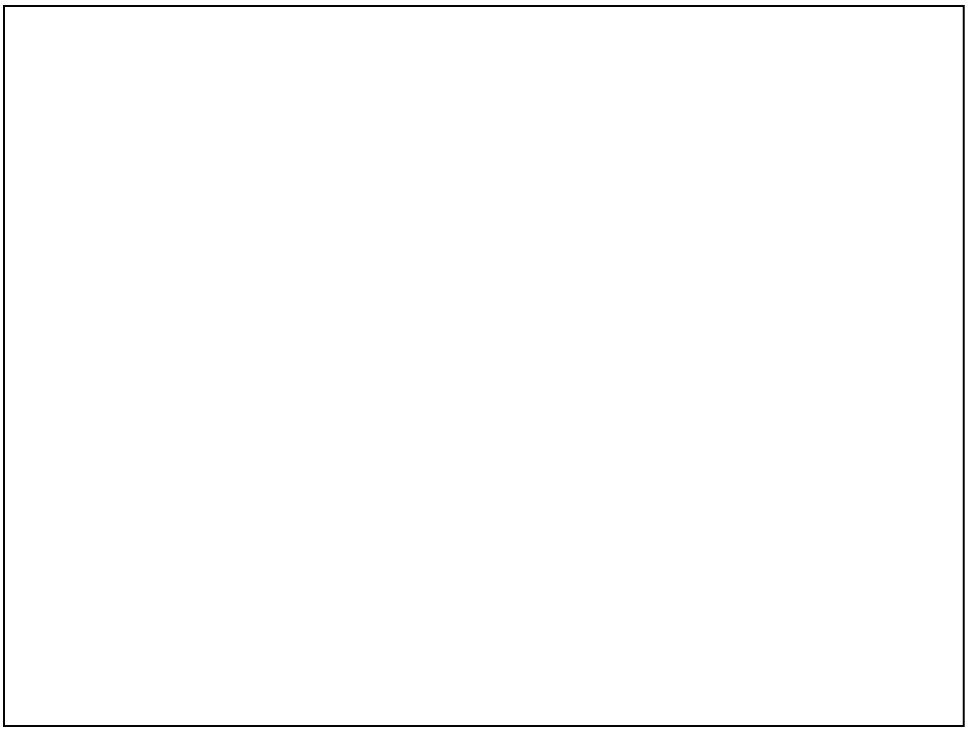
It's left as an exercise to generalize this to 3, 4, etc. rows per group.

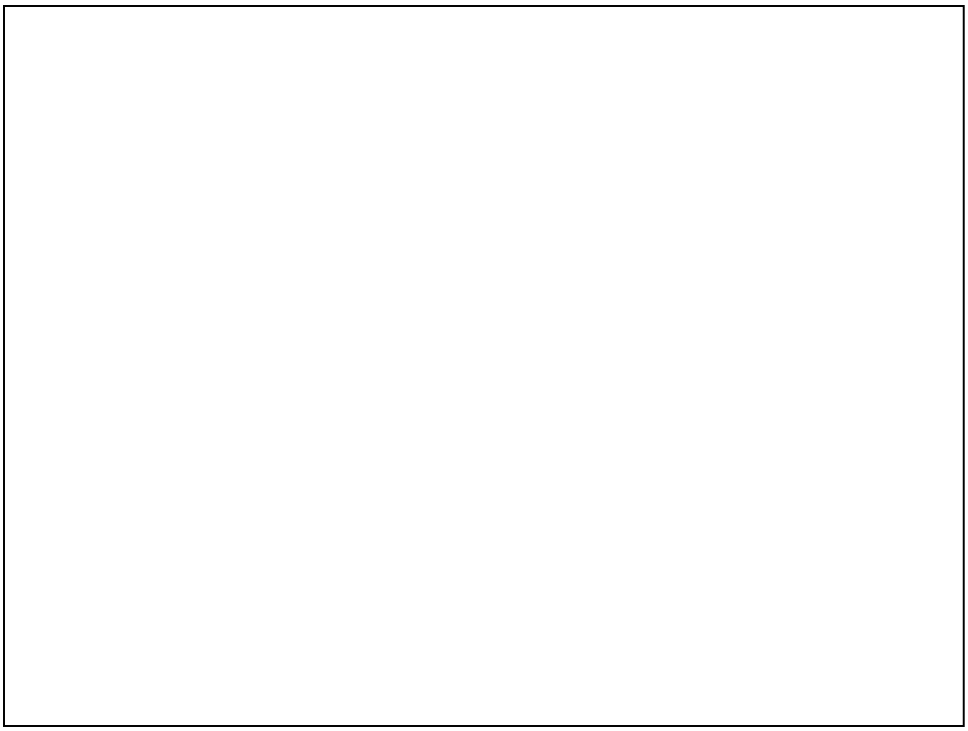
Hint: have a look at slides 17 and 18 for a way to obtain the ranking number.



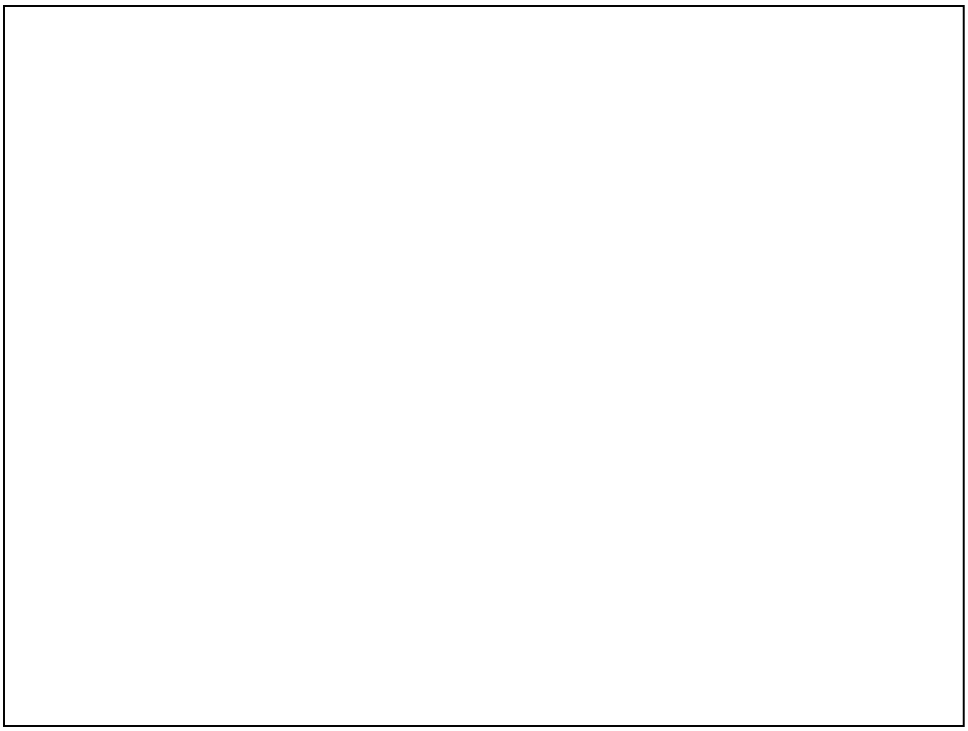








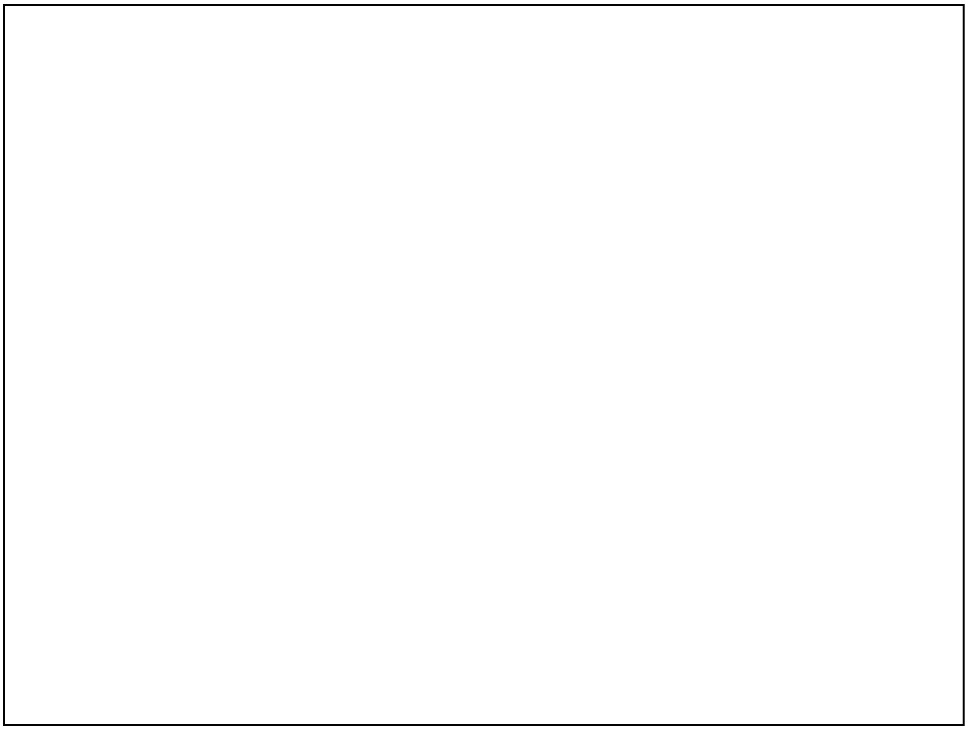
Exercise: work out the iterative way of building up the CTE `t` for the example `items` table.



“Group by expression” is a new feature of DB2 version 8.

Is it possible to obtain this effect in a more controlled way, especially without paying a (possibly high) performance price?

Yes: use CTEs!



Sometimes, chronological history information is stored in a different table from where the base information is.

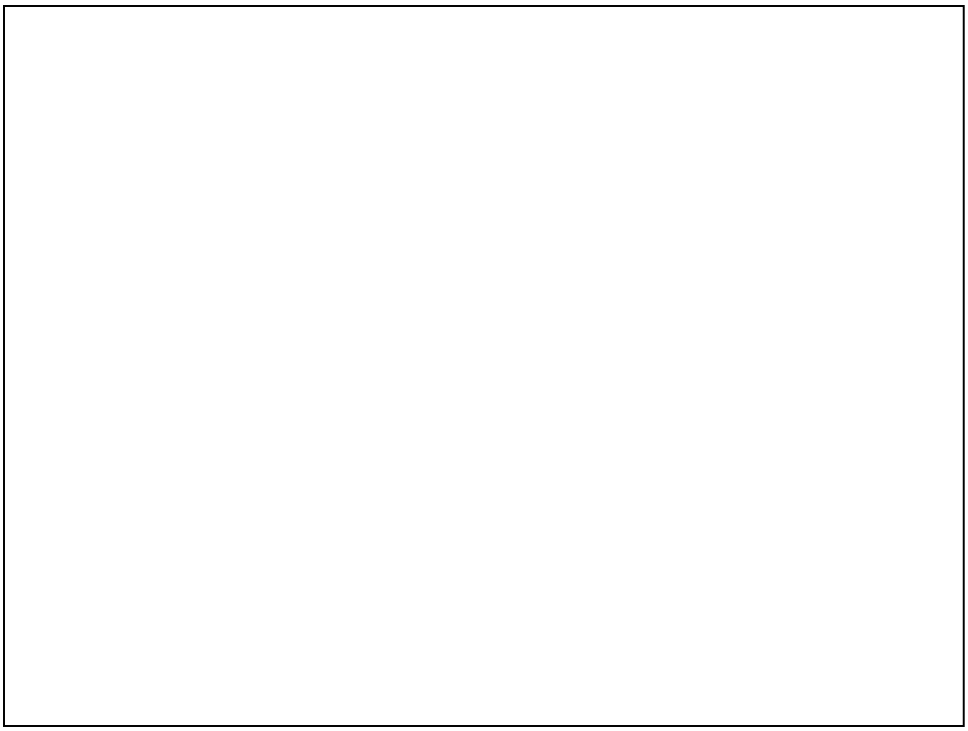
To have the full interpretation of a "history" entry, it must always be linked to either the *next* history entry, or the base table entry (= most recent situation).

A simplified example of this can be found in the catalog: the SYSCOPY table contains chronological information on backups, REORGs, ALTERs, etc.

How can we extract historic information from such a table? E.g. "what was the new situation after a certain operation in the past?"

Here, we want to know if there were operations between a certain COPY activity and the subsequent REORG (if present).

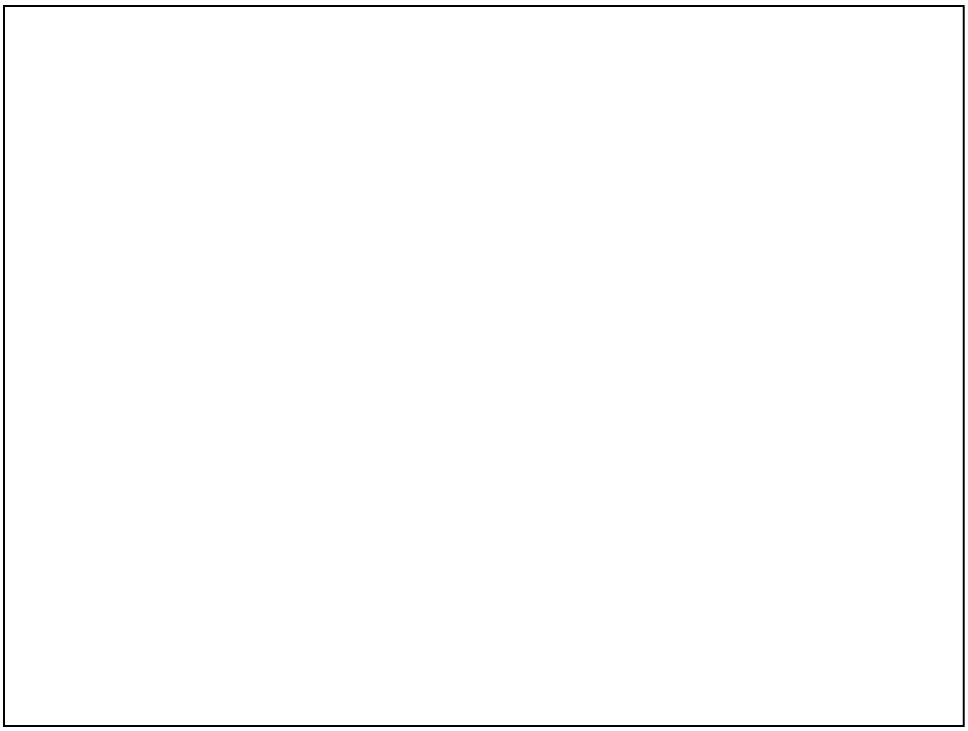
For simplicity, we just look inside the SYSCOPY table for other activity, but in general also other tables need to be queried for this purpose.



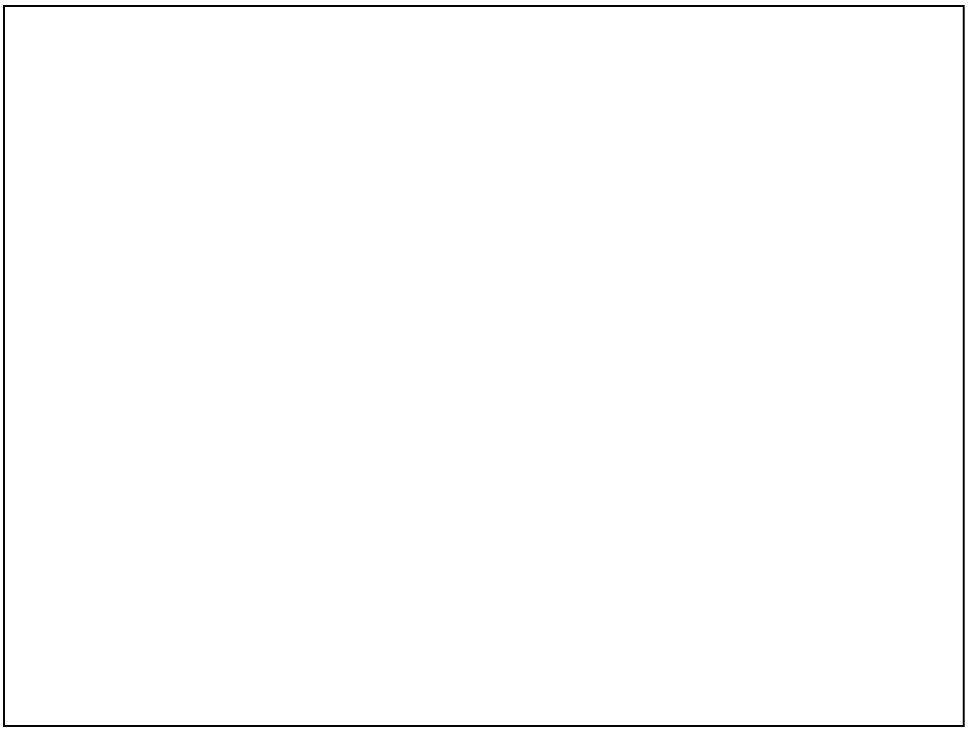
This is the (simplified) query which resulted from a question in the DB2 newsgroup on www.dbforums.com (entry t=1618384).

It asks for the cheapest holiday of 7 days (and possibly other conditions) where prices must be compared only within the same holiday code category.

To avoid duplicate code in the (complex) query, basic ingredients of the query should be isolated into CTEs.



Yet another example of a generated table through the use of recursive CTEs.



Is it possible to just show (any) 10 rows per category of a potentially very large table with a relatively low number of categories?

And more importantly, is this possible without having to scan the full table for as many times as the number of categories? (Assuming there is no index on the table.)

Try to come up with a readable query that requires just a single table scan !