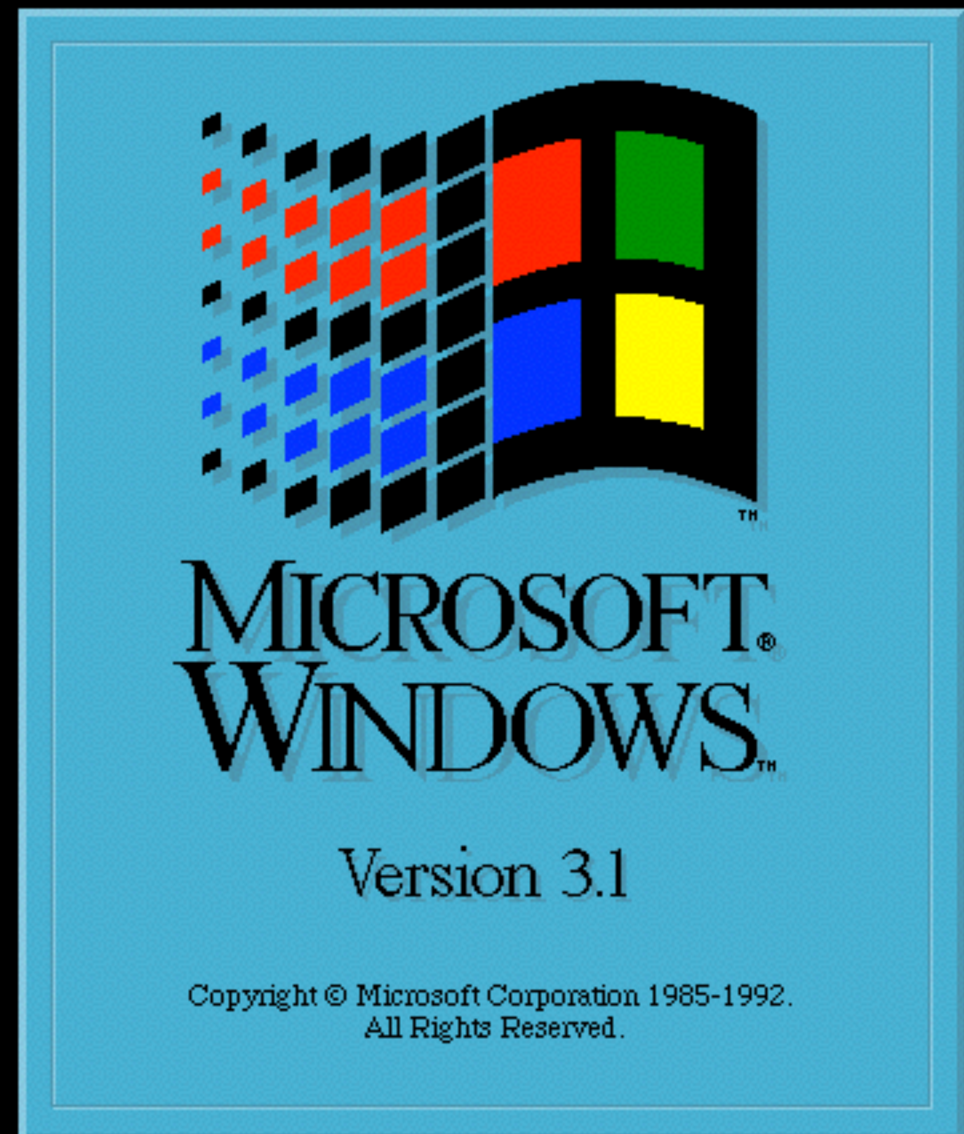
Still using

Windows 3.1?

So why stick to

SQL-92?



@ModernSQL in PostgreSQL

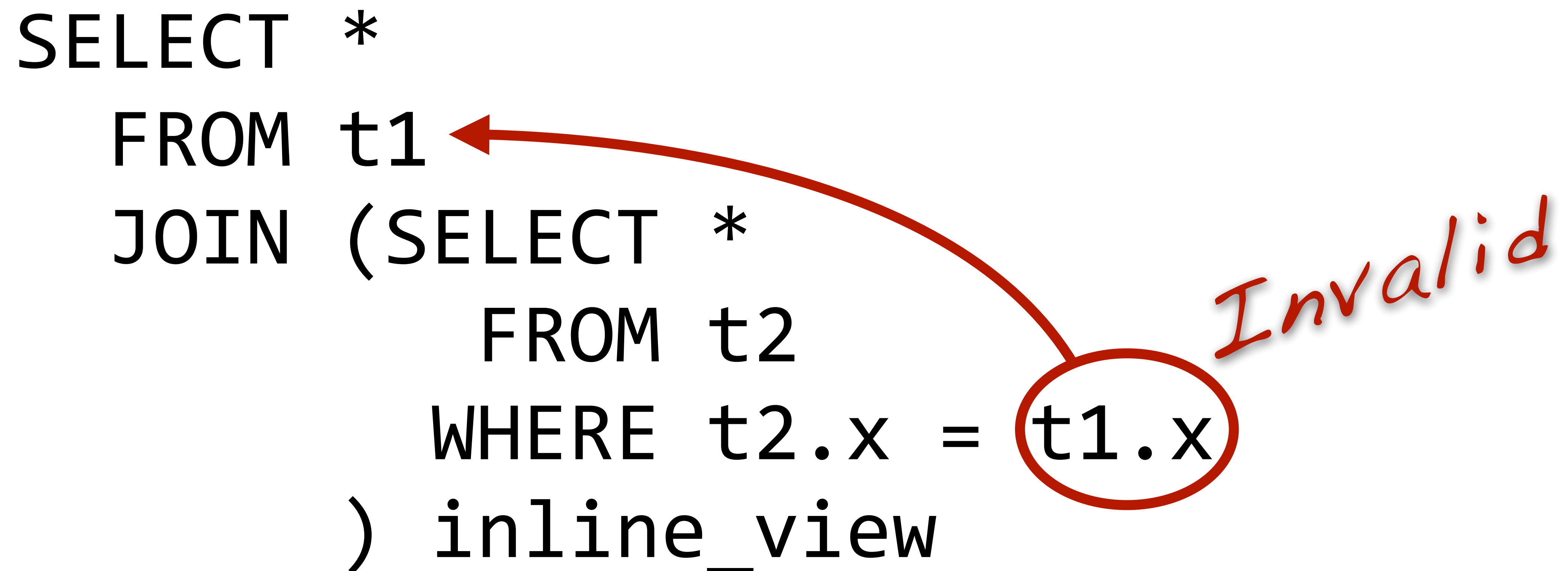
@MarkusWinand

SQL:1999

LATERAL

LATERAL Before SQL:1999

Inline views can't refer to outside the view:



LATERAL Before SQL:1999

Inline views can't refer to outside the view:

```
SELECT *
  FROM t1
  JOIN (SELECT *
          FROM t2
         WHERE t2.x = t1.x
       ) inline_view
    ON (inline_view.x = t1.x)
```

LATERAL Since SQL:1999

SQL:99 **LATERAL** views can:

```
SELECT *
  FROM t1
  JOIN LATERAL (SELECT *
                  FROM t2
                 WHERE t2.x = t1.x
               ) inline_view
    ON (true)
```

Valid due to LATERAL keyword

Useless, but still required

except for CROSS join

But **WHY**?

LATERAL and table functions

Join table functions:

```
SELECT t1.id, tf.*
  FROM t1
  JOIN LATERAL table_function(t1.id) tf
    ON (true)
```

Note: This is PostgreSQL specific. `LATERAL` is even optional here.

LATERAL and Top-N per Group

Apply `LIMIT` per row from previous table:

```
SELECT top_products.*
  FROM categories c
  JOIN LATERAL (SELECT *
                  FROM products p
                 WHERE p.cat = c.cat
                 ORDER BY p.rank DESC
                 LIMIT 3
               ) top_products
```

LATERAL and Multi-Source Top-N

Get the 10 most recent news for subscribed topics:

```
SELECT n.*
  FROM news n
  JOIN subscriptions s
    ON (n.topic = s.topic)
 WHERE s.user = ?
 ORDER BY n.created DESC
 LIMIT 10
```

LATERAL and Multi-Source Top-N

```
Limit (time=236707 rows=10)
-> Sort (time=236707 rows=10)
   Sort Method: top-N heapsort Mem: 30kB
      -> Hash Join (time=233800 rows=905029)
         -> Seq Scan on subscriptions s
            (time=369 rows=80)
         -> Hash (time=104986 rows=10^7)
            -> Seq Scan on news n
               (time=91218 rows=10^7)
Planning time: 0.294 ms
Execution time: 236707.261 ms
```

Sort/Reduce

Join everything

LATERAL and Multi-Source Top-N

```
Limit (time=236707 rows=10)
-> Sort (time=2                10)
   Sort Metho              rt Mem: 30kB
   -> Hash Join (          00 rows=905029)
      -> Seq Scan on subscriptions s
         (time=369 rows=80)
      -> Hash (time=104986
         -> Seq Scan on
            (time=91218
Planning time: 0.294 ms
Execution time: 236707.261 ms
```

Sort/Reduce

Join everything

Why producing 900k rows...

...when there are only 80 subscriptions?

LATERAL and Multi-Source Top-N

```
Limit (ti
-> Sort
   Sort M                                   30kB
     Hash                                 905029)
      -> Seq Scan                     iptions s
         (time=369 rows=80)
      -> Hash (time=104986 rows=10^7)
         -> Seq Scan on news n
            (time=91218 rows=10^7)
  g time: 0.294 ms
  n time: 236707.261 ms
```

reduce

Only the 10 most recent per subscription, you need.

LATERAL and Multi-Source Top-N

```
SELECT n.*
  FROM subscriptions s
  JOIN LATERAL (SELECT *
                 FROM news n
                WHERE n.topic = s.topic
                ORDER BY n.created DESC
                LIMIT 10
              ) top_news ON (true)
 WHERE s.user_id = ?
 ORDER BY n.created DESC
 LIMIT 10
```

LATERAL and Multi-Sour

```
Limit (time=2.488 rows=10)
-> Sort (time=2.487 rows=10)
   -> Nested Loop (time=2.339 rows=800)
      -> Index Only Scan using pk on s
         (time=0.042 rows=80)
      -> Limit
         (time=0.027 rows=10 loops=80)
         -> Index Scan Bac
            using news_to                n
Planning time: 0.161 ms
Execution time: 2.519 ms
```

Limited to 10 times # of subscriptions

About 100 000 times faster

LATERAL in an Nutshell

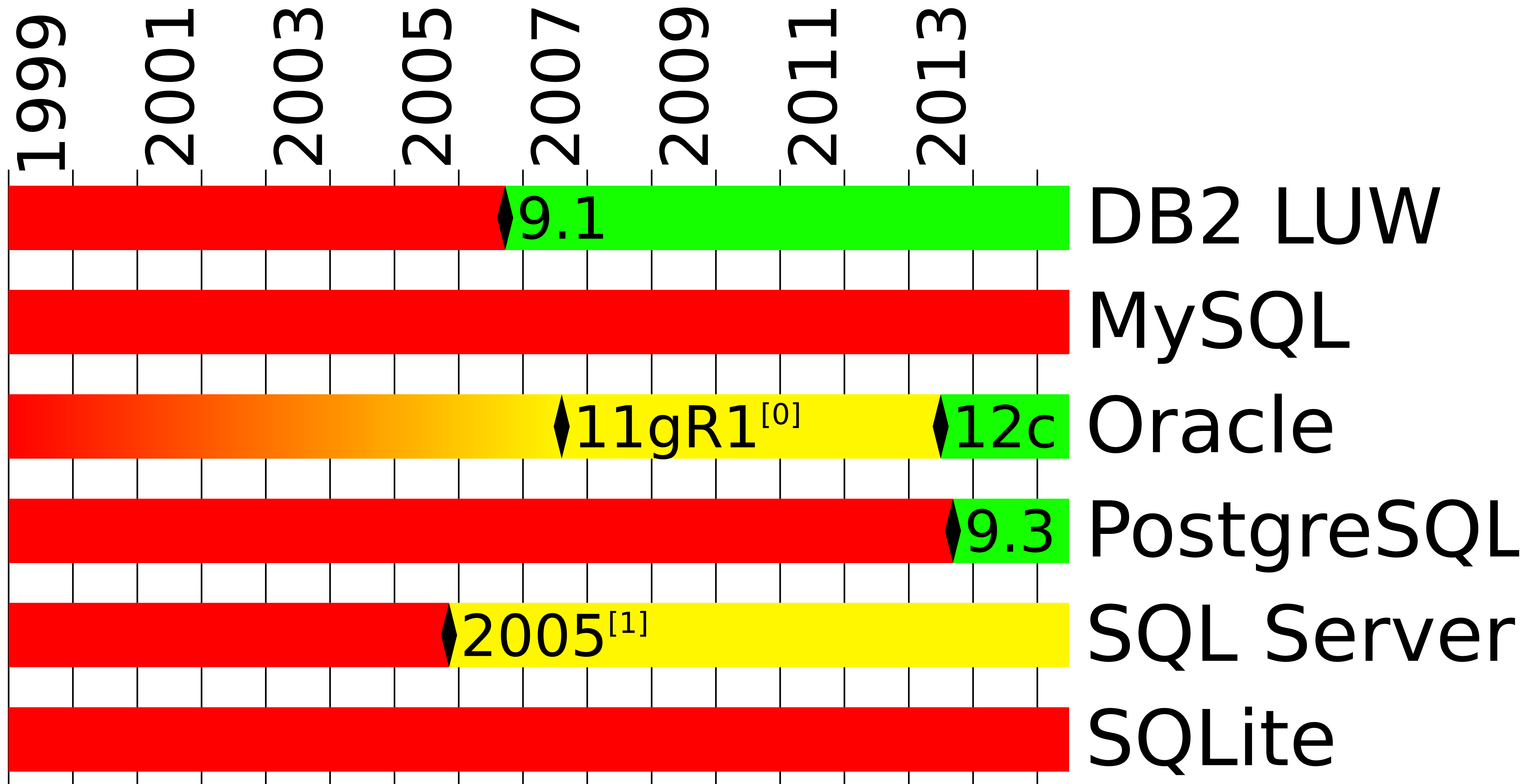
`LATERAL` is the "for each" loop of SQL

`LATERAL` plays well with outer joins

`LATERAL` is great for Top-N subqueries

`LATERAL` can join table functions (`unnest`!)

LATERAL Availability (SQL:1999)



[0] Undocumented. Requires setting trace event 22829.
[1] `LATERAL` is not supported as of SQL Server 2014 but `[CROSS|OUTER] APPLY` can be used for the same effect.

`WITH`

(Common Table Expressions)

WITH Before SQL:99

Nested queries are hard to read:

```
SELECT …
  FROM (SELECT …
          FROM t1
          JOIN (SELECT … FROM …
               ) a ON (…)
       ) b
  JOIN (SELECT … FROM …
       ) c ON (…)
```

WITH Before SQL:99

Nested queries are hard to read:

```
SELECT …
  FROM (SELECT …
          FROM t1
          JOIN (SELECT … FROM …
               ) a ON (…)
       ) b
  JOIN (SELECT … FROM …
       ) c ON (…)
```

Understand this first

WITH Before SQL:99

Nested queries are hard to read:

```
SELECT …
  FROM (SELECT …
          FROM t1
          JOIN (SELECT … FROM …
               ) a ON (…)
       ) b
  JOIN (SELECT … FROM …
       ) c ON (…)
```

Then this…

WITH Before SQL:99

Nested queries are hard to read:

```
SELECT …
  FROM (SELECT …
          FROM t1
          JOIN (SELECT … FROM …
               ) a ON (…)
       ) b
  JOIN (SELECT … FROM …
       ) c ON (…)
```

Then this...

WITH Before SQL:99

Nested queries are hard to read:

Finally the first line makes sense

```
SELECT …
  FROM (SELECT …
          FROM t1
          JOIN (SELECT … FROM …
               ) a ON (…)
       ) b
  JOIN (SELECT … FROM …
       ) c ON (…)
```

WITH Since SQL:99

CTEs are statement-scoped views:

```
WITH
 a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM …),
```

WITH Since SQL:99

CTEs are statement-scoped views:

Keyword

```
WITH
 a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM …),
```

WITH Since SQL:99

CTEs are statement-scoped views:

Name of CTE and (here optional) column names

```
WITH
 a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM …),
```

WITH Since SQL:99

CTEs are statement-scoped views:

```
WITH
 a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM …),
```

Definition

WITH Since SQL:99

CTEs are statement-scoped views:

```
WITH
 a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM …),
```

Introduces another CTE

Don't repeat WITH

WITH Since SQL:99

CTEs are statement-scoped views:

```
WITH
 a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM …),

 b (c4, …)
AS (SELECT c4, …
      FROM t1
      JOIN a
        ON (…)
   ),
```

May refer to previous CTEs

WITH Since SQL:99

```
 b (c4, …)
AS (SELECT c4, …
      FROM t1
      JOIN a
        ON (…)
   ),
 c (…)
AS (SELECT … FROM …)
SELECT …
  FROM b JOIN c ON (…)
```

Third CTE

WITH Since SQL:99

```
 b (c4, …)
AS (SELECT c4, …
      FROM t1
      JOIN a
        ON (…)
   ),

 c (…)
AS (SELECT … FROM …)

SELECT …
  FROM b JOIN c ON (…)
```

No comma!

WITH Since SQL:99

```
 b (c4, …)
AS (SELECT c4, …
      FROM t1
      JOIN a
        ON (…)
   ),

 c (…)
AS (SELECT … FROM …)

SELECT …
  FROM b JOIN c ON (…)
```

Main query

C

WITH Since SQL:99

```
WITH
 a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM …),

 b (c4, …)
AS (SELECT c4, …
      FROM t1
      JOIN a
        ON (…)
   ),

 c (…)
AS (SELECT … FROM …)

SELECT …
  FROM b JOIN c ON (…)
```

WITH in an Nutshell

`WITH` are the "private methods" of SQL

`WITH` views can be referred to multiple times

`WITH` allows chaining instead of nesting

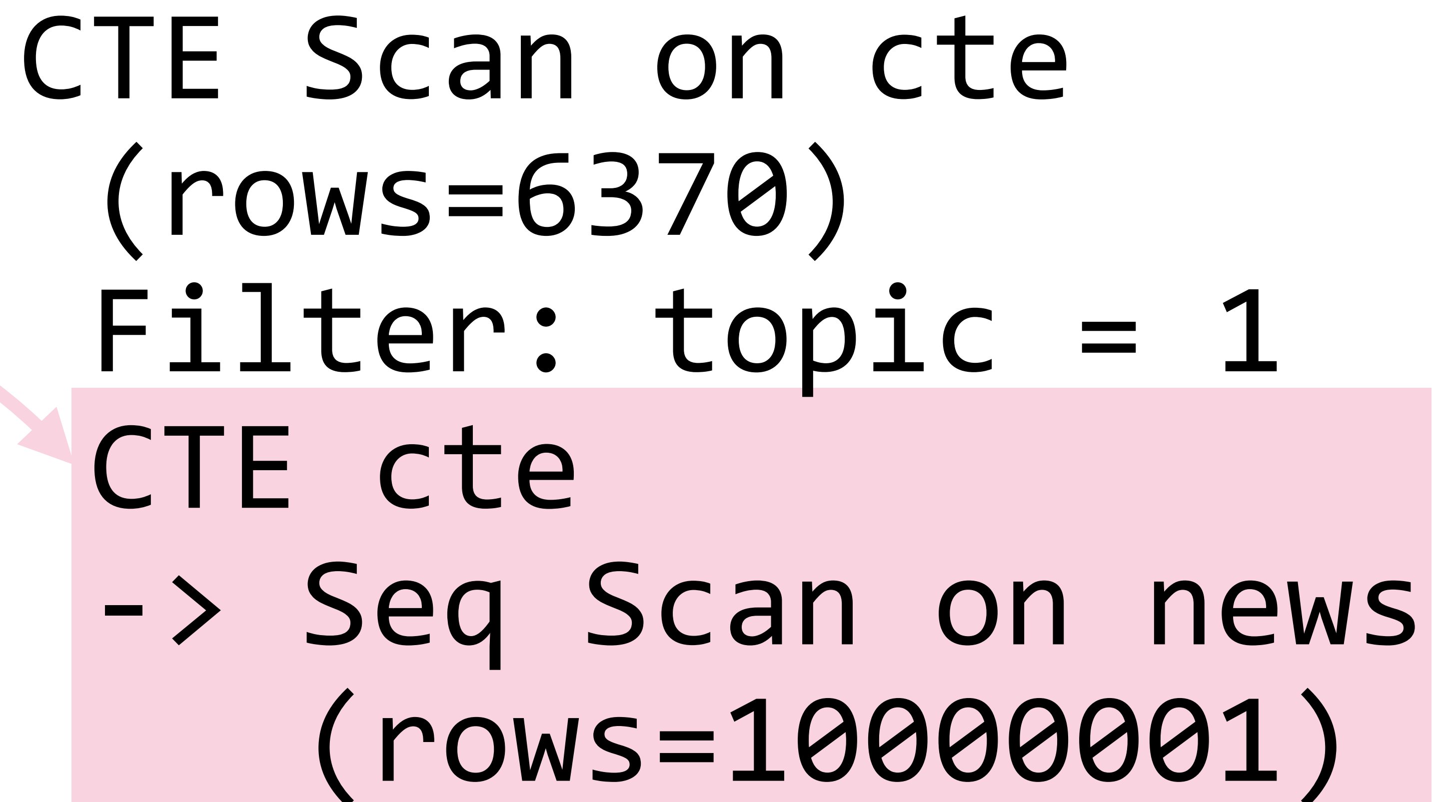
`WITH` is allowed where `SELECT` is allowed

```
INSERT INTO tbl
WITH ... SELECT ...
```

WITH PostgreSQL Particularities

In PostgreSQL **`WITH`** views are more like materialized views:

```
WITH cte AS
(SELECT *
   FROM news)
SELECT *
  FROM cte
 WHERE topic=1
```



WITH PostgreSQL Particularities

In PostgreS[...]ews are more like materialize[...]

```
WITH cte AS
(SELECT *
   FROM news)
SELECT *
  FROM cte
 WHERE topic=1
```

```
CTE Scan on cte
 (rows=6370)
 Filter: topic = 1
 CTE cte
 -> Seq Scan on news
    (rows=10000001)
```

WITH PostgreSQL Particularities

Normal views and inline-views support "predicate pushdown":

```
SELECT *
  FROM (
  SELECT *
    FROM news
  ) n
WHERE topic=1;
```

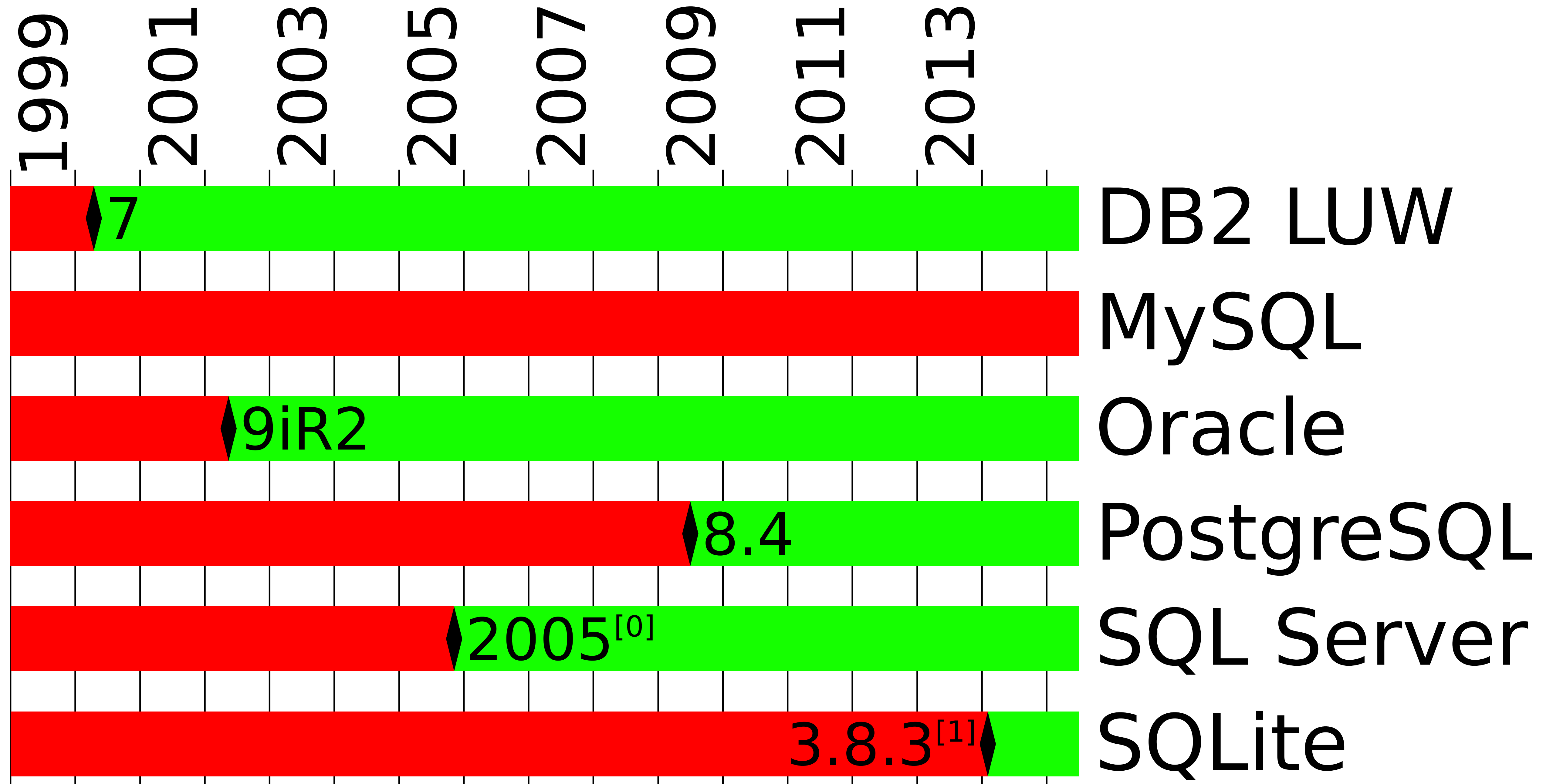
```
Bitmap Heap Scan
on news (rows=6370)
->Bitmap Index Scan
  on idx (rows=6370)
  Cond: topic=1
```

WITH PostgreSQL Particularities

PostgreSQL 9.1+ allows `INSERT`, `UPDATE` and `DELETE` within `WITH`:

```
WITH deleted_rows AS (
   DELETE FROM source_tbl
   RETURNING *
)
INSERT INTO destination_tbl
SELECT * FROM deleted_rows;
```

WITH Availability (SQL:99)



[0] Only allowed at the very begin of a statement. E.g. WITH...INSERT...SELECT.
[1] Only for top-level SELECT statements

WITH RECURSIVE

(Common Table Expressions)

WITH RECURSIVE Before SQL:99

WITH RECURSIVE Before SQL:99

(This page is intentionally left blank)

WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a `UNION [ALL]`:

```
WITH RECURSIVE cte (n)
  AS (SELECT 1
       UNION ALL
      SELECT n+1
        FROM cte
       WHERE n < 3)
SELECT * FROM cte;
```

WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a **`UNION [ALL]`**:

Keyword

```
WITH RECURSIVE cte (n)
   AS (SELECT 1
        UNION ALL
       SELECT n+1
         FROM cte
        WHERE n < 3)
SELECT * FROM cte;
```

WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a **UNION [ALL]**:

```
WITH RECURSIVE cte (n)
  AS (SELECT 1
       UNION ALL
      SELECT n+1
        FROM cte
       WHERE n < 3)
SELECT * FROM cte;
```

Column list mandatory here

WITH RECURSIVE Since SQL:99

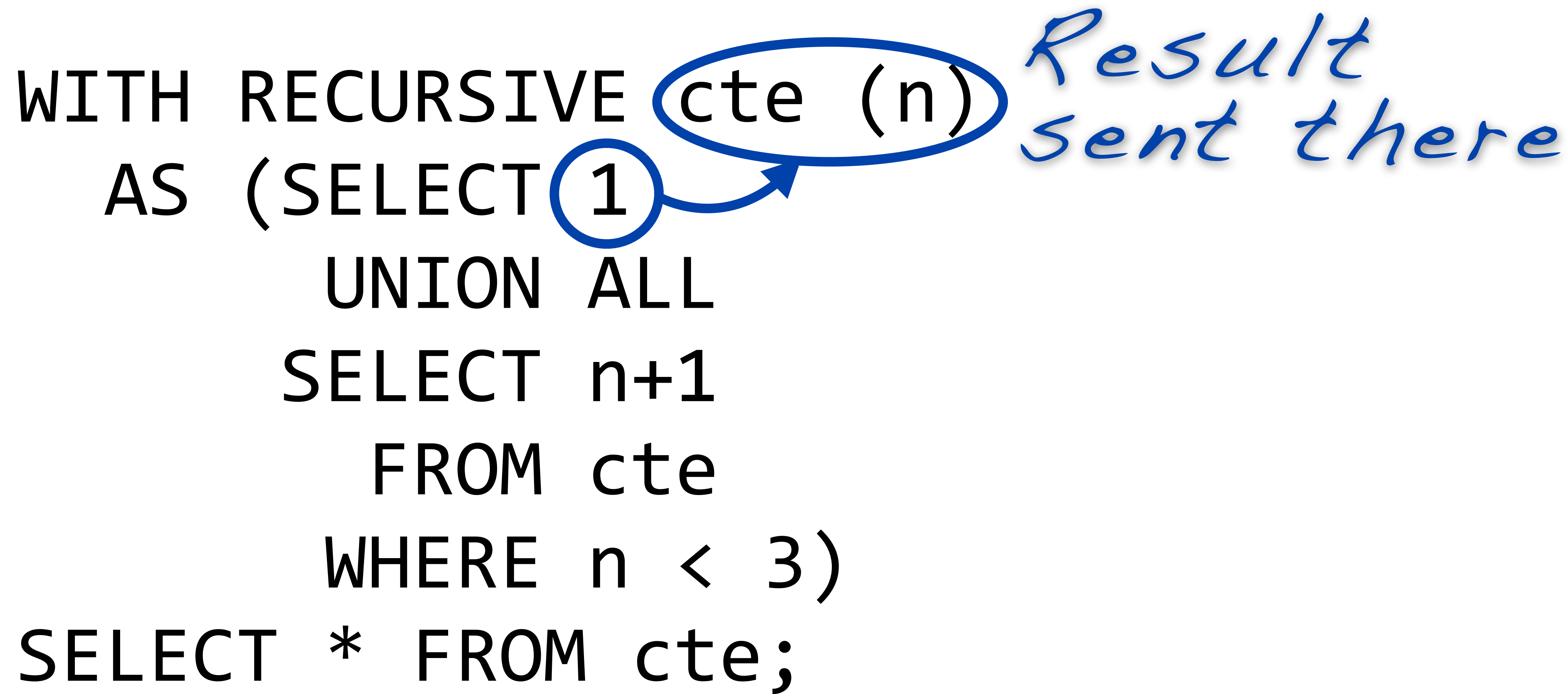
Recursive common table expressions may refer to themselves in the second leg of a **UNION [ALL]**:

```
WITH RECURSIVE cte (n)
   AS (SELECT 1
         UNION ALL
        SELECT n+1
          FROM cte
         WHERE n < 3)
SELECT * FROM cte;
```

Executed first

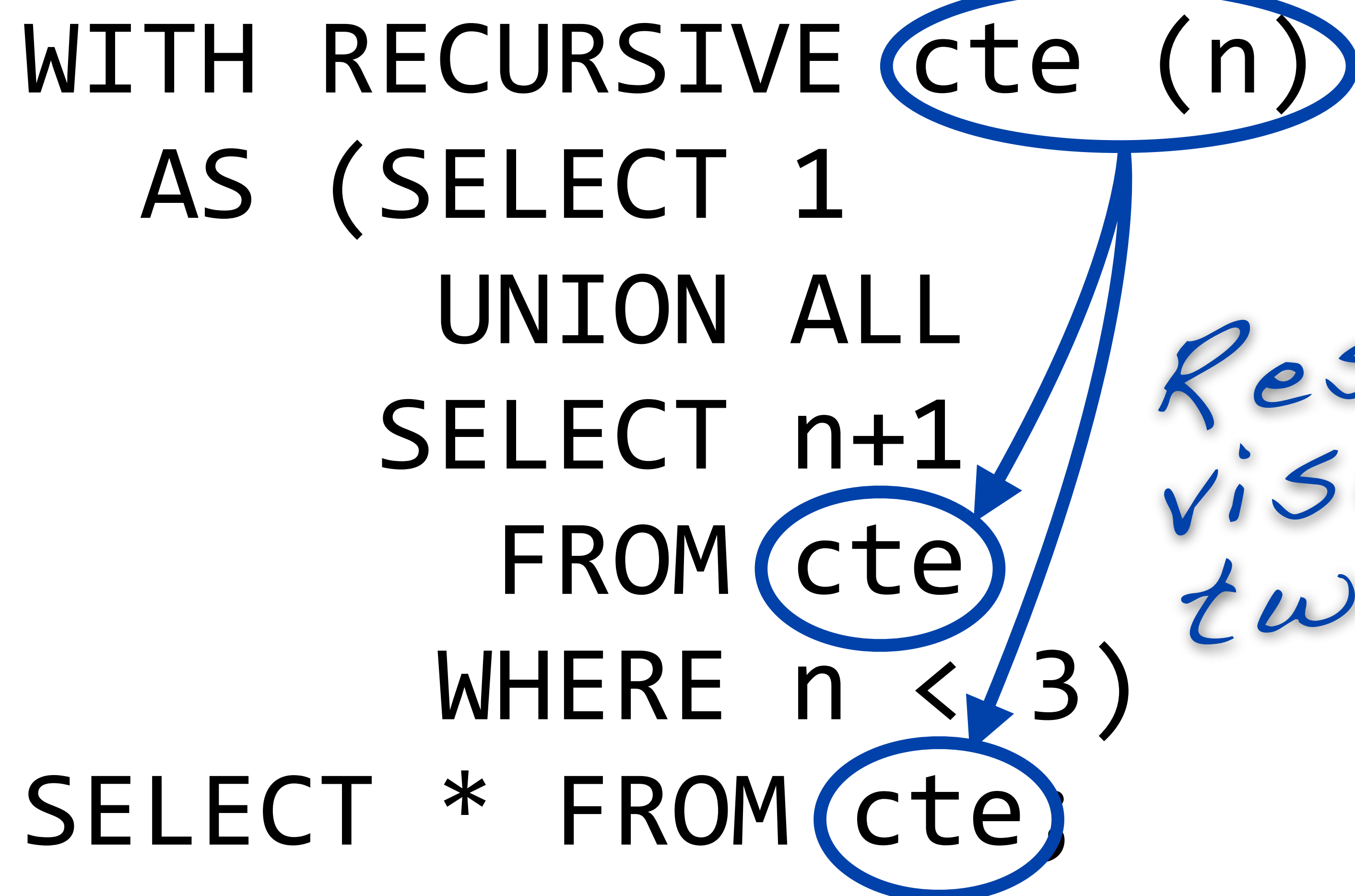
WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a `UNION [ALL]`:



WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a `UNION [ALL]`:



WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a **UNION [ALL]**:

```
WITH RECURSIVE cte (n)
  AS (SELECT 1
       UNION ALL
      SELECT n+1
        FROM cte
       WHERE n < 3)
SELECT * FROM cte;
```

```
 n
---
 1
```

Once it becomes part of the final result

WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a **UNION [ALL]**:

```
WITH RECURSIVE cte (n)
  AS (SELECT 1
       UNION ALL
      SELECT n+1
        FROM cte
       WHERE n < 3)
SELECT * FROM cte;
```

Second leg of UNION is executed

n
1

WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a **UNION [ALL]**:

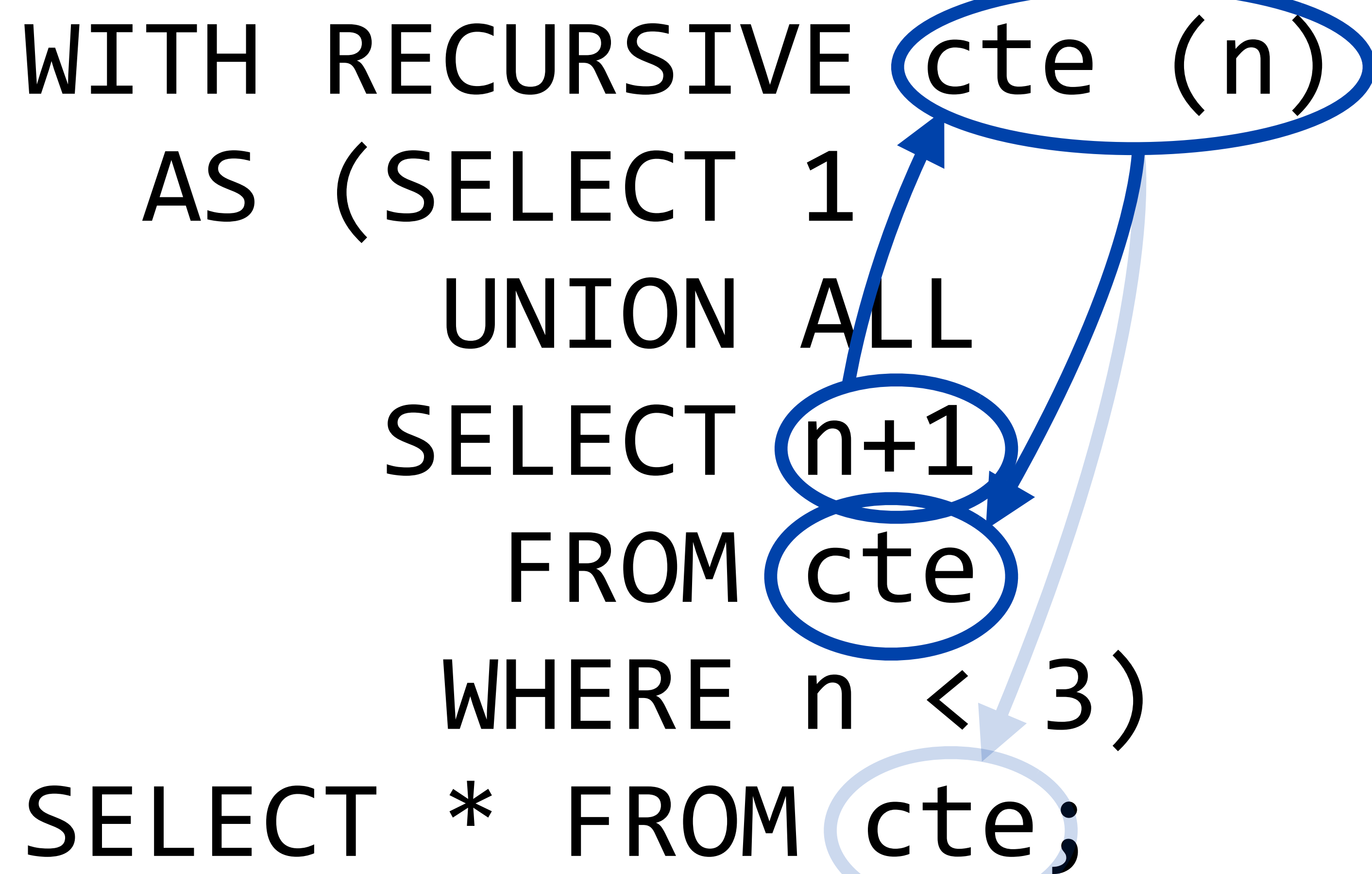
```
WITH RECURSIVE cte (n)
  AS (SELECT 1
        UNION ALL
       SELECT n+1
         FROM cte
        WHERE n < 3)
SELECT * FROM cte;
```

Result sent there again

n
1

WITH RECURSIVE Since SQL:99

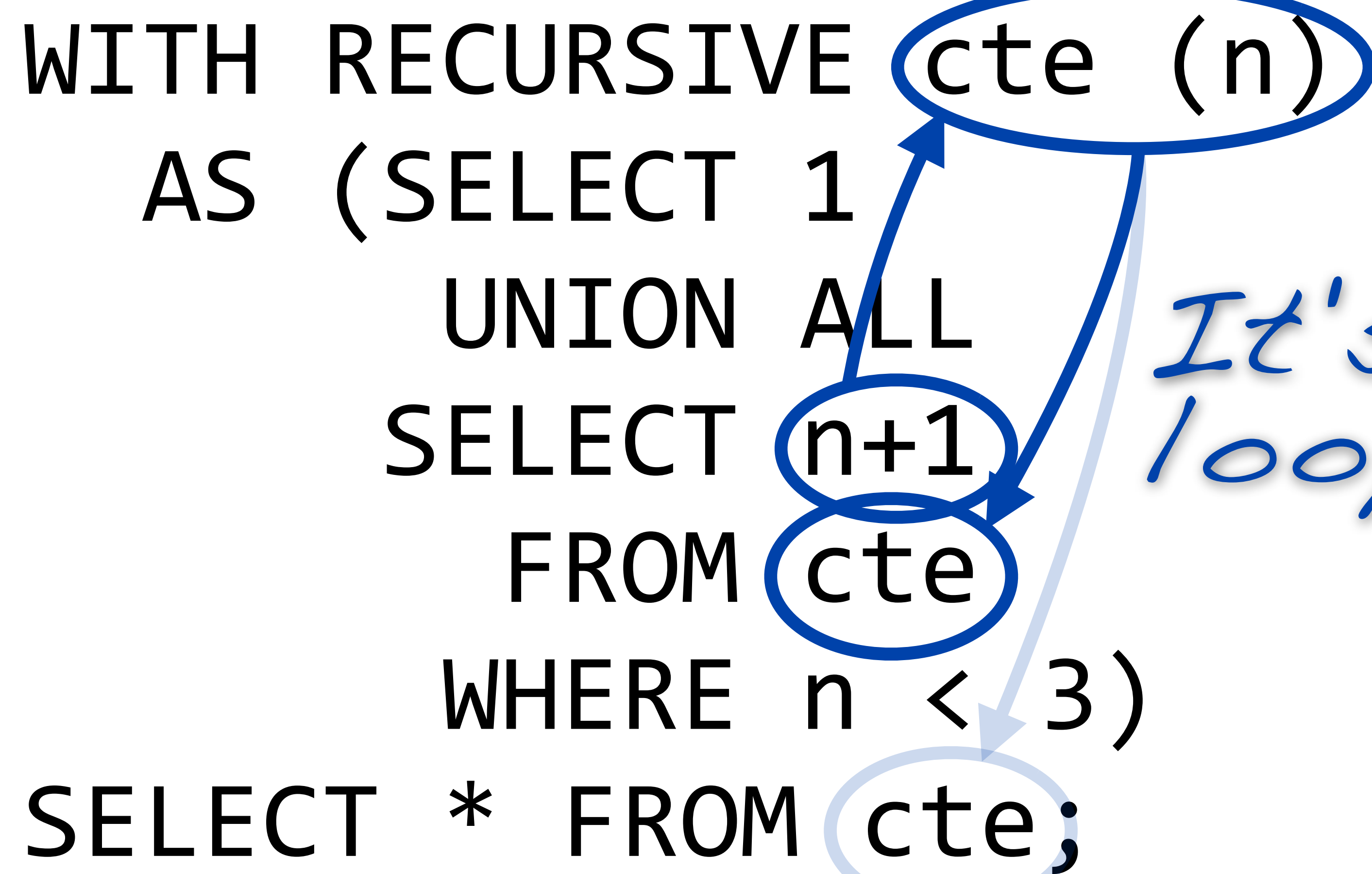
Recursive common table expressions may refer to themselves in the second leg of a **`UNION [ALL]`**:




```
 n
---
 1
```

WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a `UNION [ALL]`:




```
 n
---
 1
```

WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a `UNION [ALL]`:

```
WITH RECURSIVE cte (n)
  AS (SELECT 1
       UNION ALL
      SELECT n+1
        FROM cte
       WHERE n < 3)
SELECT * FROM cte;
```

```
 n
---
 1
 2
```

It's a loop!

WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a `UNION [ALL]`:

```
WITH RECURSIVE cte (n)
  AS (SELECT 1
       UNION ALL
      SELECT n+1
        FROM cte
       WHERE n < 3)
SELECT * FROM cte;
```

It's a loop!

```
 n
---
 1
 2
 3
```

WITH RECURSIVE Since SQL:99

Recursive common table expressions may refer to themselves in the second leg of a `UNION [ALL]`:

```
WITH RECURSIVE cte (n)
  AS (SELECT 1
       UNION ALL
      SELECT n+1
        FROM cte
       WHERE n < 3)
SELECT * FROM cte;
```

```
  n
 ---
  1
  2
  3
(3 rows)
```

n=3 doesn't match

Loop terminates

WITH RECURSIVE Use Cases

- Row generators (previous example)
  (`generate_series` is proprietary)

- Processing graphs
  (don't forget the cycle detection!)

- Generally said: Loops that...
  - ... pass data to the next iteration
  - ... need a "dynamic" abort condition

WITH RECURSIVE in a Nutshell

`WITH RECURSIVE` is the `while` of SQL

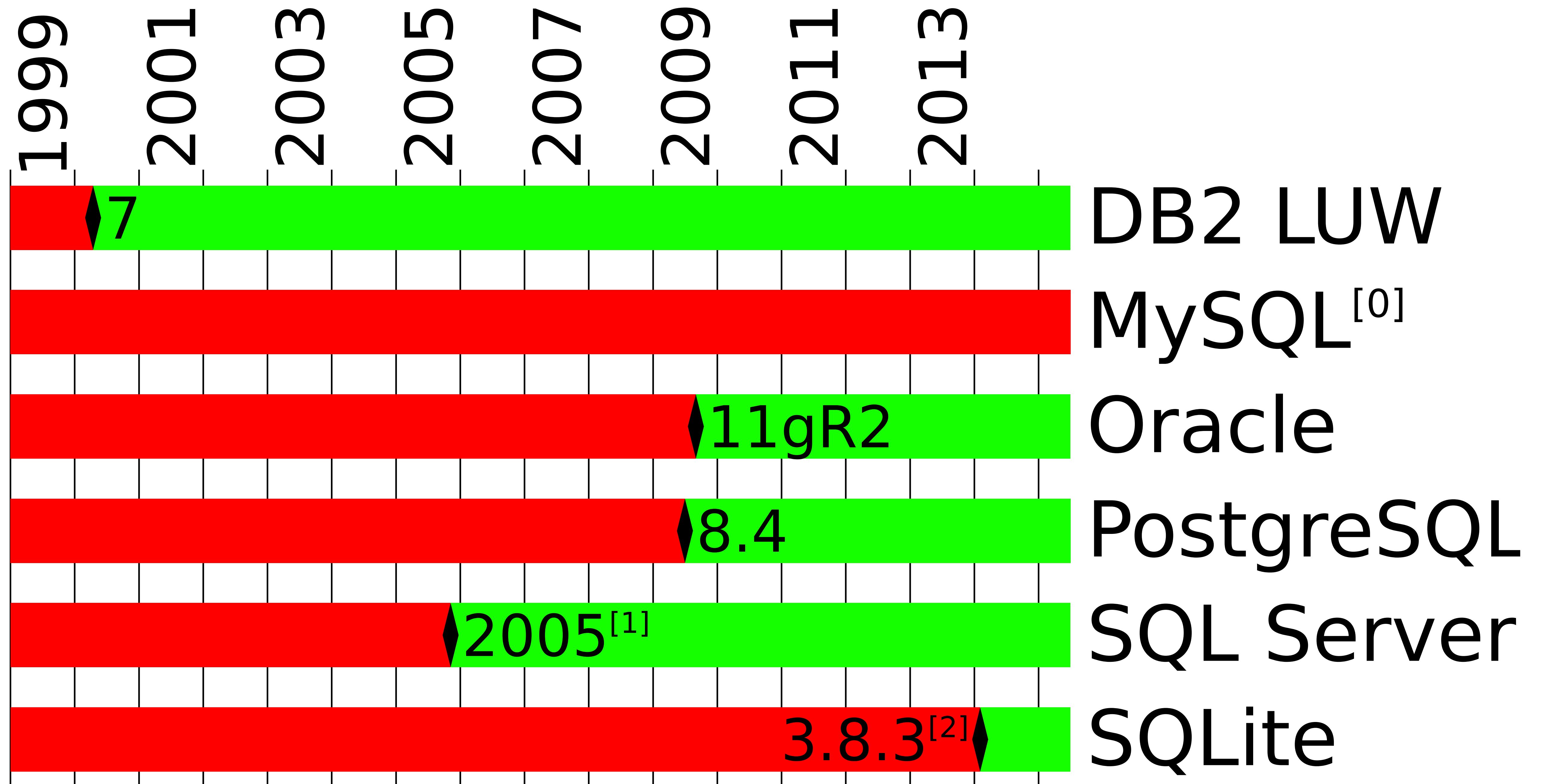
`WITH RECURSIVE` "supports" infinite loops

(SQL Server requires setting `MAXRECURSION 0`)

Except PostgreSQL, databases generally don't require the `RECURSIVE` keyword.

SQL Server & Oracle don’t even know the keyword, but allow recursive CTEs anyway.

WITH RECURSIVE Availability



[0] Feature request #16244 from 2006-01-06
[1] Default limit of 100 iterations. Use `OPTION (MAXRECURSION 0)` to disable
[2] Only for top-level `SELECT` statements

SQL:2003

FILTER

FILTER Before SQL:2003

Pivot table: Years on the Y asis, Month on X axis:

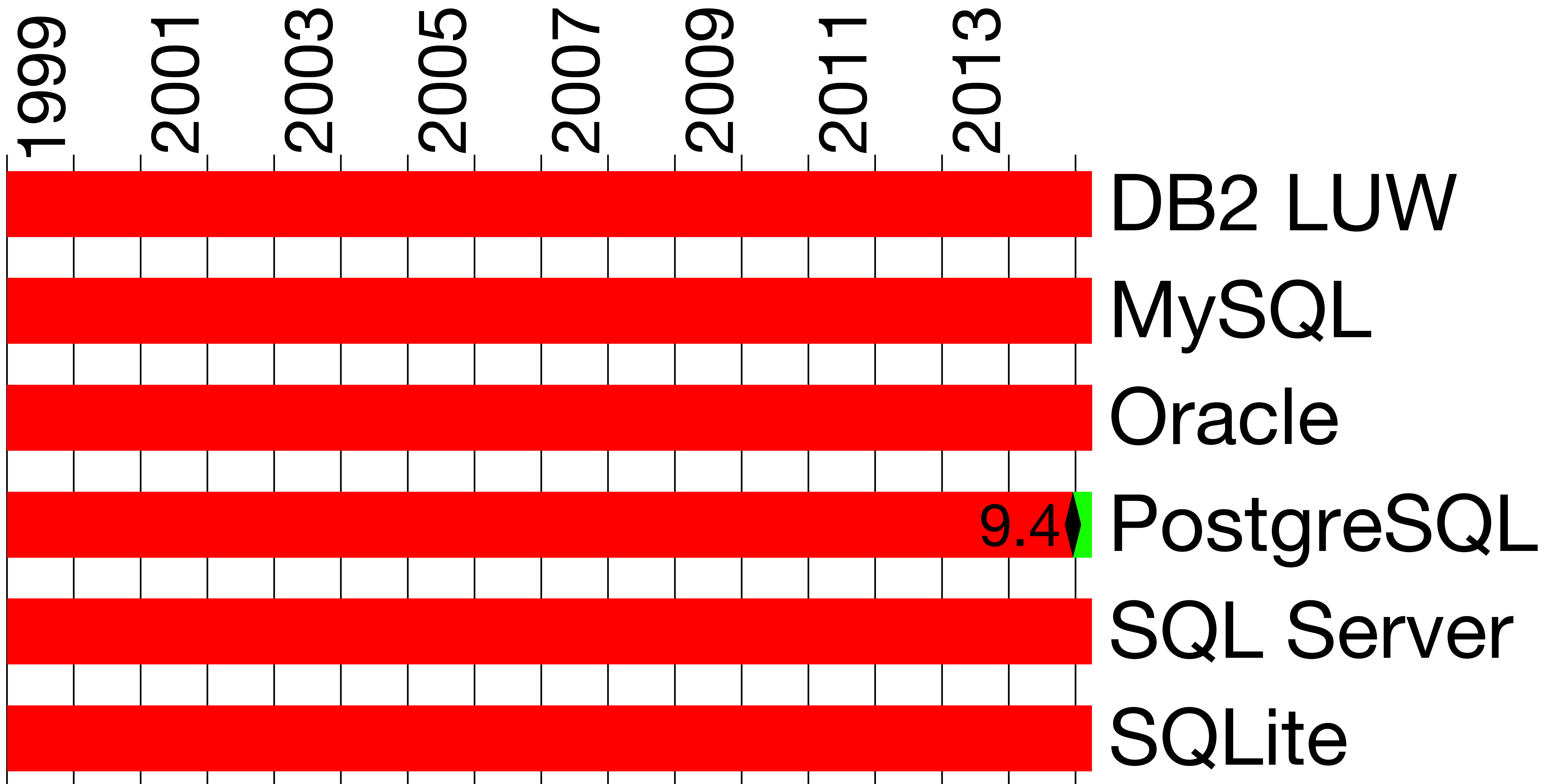
```
SELECT YEAR,
SUM(CASE WHEN MONTH = 1
         THEN sales ELSE 0 END) JAN,
SUM(CASE WHEN MONTH = 2
         THEN sales ELSE 0 END) FEB,…
  FROM sale_data
 GROUP BY YEAR
```

FILTER Since SQL:2003

SQL:2003 has `FILTER`:

```
SELECT YEAR,
SUM(sales) FILTER (WHERE MONTH = 1) JAN,
SUM(sales) FILTER (WHERE MONTH = 2) FEB,
…
  FROM sale_data
 GROUP BY YEAR;
```

FILTER Availability (SQL:2003)



OVER

and

PARTITION BY

OVER Before SQL:2003

Show percentage of department salary:

```
WITH total_salary_by_department
```

OVER Before SQL:2003

Show percentage of department salary:

```
WITH total_salary_by_department
  AS (SELECT dep, SUM(salary) total
        FROM emp
       GROUP BY dep)
```

OVER Before SQL:2003

Show percentage of department salary:

```
WITH total_salary_by_department
  AS (SELECT dep, SUM(salary) total
        FROM emp
       GROUP BY dep)
SELECT dep, emp_id, salary,
       salary/ts.total*100 "% of dep"
  FROM emp
  JOIN total_salary_by_department ts
    ON (emp.dep = ts.dep)
```

~~OVER Before SQL:2003~~ WITH intermezzo

Show percentage of department salary:

```
WITH total_salary_by_department
  AS (SELECT dep, SUM(salary) total
        FROM emp
       GROUP BY dep)
SELECT dep, emp_id, salary,
       salary/ts.total*100 "% of dep"
  FROM emp
  JOIN total_salary_by_department ts
    ON (emp.dep = ts.dep)
```

OVER ~~Before SQL:2003~~ WITH intermezzo

Show percentage of department salary:

```
WITH total_salary_by_department
  AS (SELECT dep, SUM(salary) total
        FROM emp
       GROUP BY dep)
SELECT dep, emp_id, salary,
       salary/ts.total*100 "% of dep"
  FROM emp
  JOIN total_salary_by_department ts
    ON (emp.dep = ts.dep)
```

OVER Before SQL:2003

WITH intermezzo

Show percentage of department salary:

```
WITH total_salary_by_department
  AS (SELECT dep, SUM(salary) total
        FROM emp
       GROUP BY dep)
SELECT dep, emp_id, salary,
       salary/ts.total*100 "% of dep"
  FROM emp
  JOIN total_salary_by_department ts
    ON (emp.dep = ts.dep)
```

~~OVER Before SQL:2003~~ WITH intermezzo

Show percentage of department salary:

```
WITH total_salary_by_department
  AS (SELECT dep, SUM(salary) total
        FROM emp
       GROUP BY dep)
SELECT dep, emp_id, salary,
       salary/ts.total*100 "% of dep"
  FROM emp
  JOIN total_salary_by_department ts
    ON (emp.dep = ts.dep)
```

OVER Before SQL:2003

`GROUP BY` =

`DISTINCT`

\+

Aggregates

OVER Since SQL:2003

Build aggregates without GROUP BY:

```
SELECT dep, emp_id, salary,
       salary/SUM(salary)
              OVER(PARTITION BY dep)
             * 100 "% of dep"
  FROM emp
```

OVER How It Works

```
SELECT dep,
       salary



  FROM emp;
```

dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

OVER How It Works

```
SELECT dep,
       salary,



  FROM emp;
```

dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

OVER How It Works

```
SELECT dep,
       salary,
       SUM(salary)

  FROM emp;
```

dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

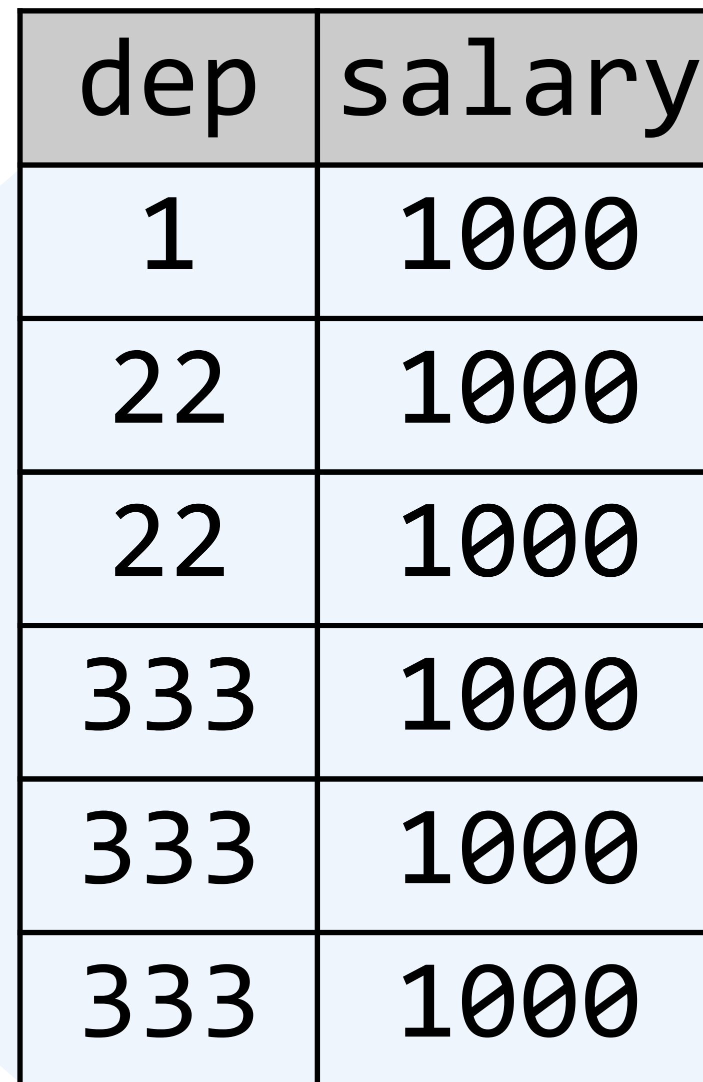
OVER How It Works

```
SELECT dep,
       salary,
       SUM(salary)
       OVER ()
  FROM emp;
```

dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

OVER How It Works

```
SELECT dep,
       salary,
       SUM(salary)
       OVER ()
  FROM emp;
```



dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

OVER How It Works

```
SELECT dep,
       salary,
       SUM(salary)
       OVER ()
  FROM emp;
```

dep	salary	
1	1000	6000
22	1000	6000
22	1000	6000
333	1000	6000
333	1000	6000
333	1000	6000

OVER How It Works

```
SELECT dep,
       salary,
       SUM(salary)

  FROM emp;
```

dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

OVER How It Works

```
T dep,
  salary,
  SUM(salary)

M emp;
```

dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

OVER How It Works

```
T dep,
  salary,
  SUM(salary)
  OVER(PARTITION BY dep)
M emp;
```

dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

OVER How It Works

```
T dep,
  salary,
  SUM(salary)
  OVER(PARTITION BY dep)
M emp;
```

dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

OVER How It Works

```
T dep,
  salary,
  SUM(salary)
  OVER(PARTITION BY dep)
M emp;
```

dep	salary	ts
1	1000	1000
22	1000	2000
22	1000	2000
333	1000	3000
333	1000	3000
333	1000	3000

OVER in a Nutshell

`OVER` may follow any aggregate function

`OVER` defines which rows are visible at each row
(it does not limit the result in any way)

`OVER()` makes all rows visible at every row

`OVER(PARTITION BY` x) segregates like `GROUP BY`

OVER

and

ORDER BY

OVER Before SQL:2003

Calculating a running total:

```
SELECT txid, value,




  FROM transactions tx1
 WHERE acnt = ?
 ORDER BY txid
```

OVER Before SQL:2003

Calculating a running total:

```
SELECT txid, value,
       (SELECT SUM(value)
          FROM transactions tx2
         WHERE tx2.acnt = tx1.acnt
           AND tx2.txid <= tx1.txid) bal
  FROM transactions tx1
 WHERE acnt = ?
 ORDER BY txid
```

OVER Before SQL:2003

Calculating a running total:

```
SELECT txid, value,
         (SELECT SUM(value)
            FROM transactions tx2
           WHERE tx2.acnt = tx1.acnt
             AND tx2.txid <= tx1.txid) bal
  FROM transactions tx1
 WHERE acnt = ?
 ORDER BY txid
```

OVER Before SQL:2003

Before SQL:2003 running totals were awkward:

- Requires a scalar sub-select or self-join
- Poor maintainability (repetitive clauses)
- Poor performance

The only real answer was:

Do it in the application

OVER Since SQL:2003

With SQL:2003 you can narrow the window:

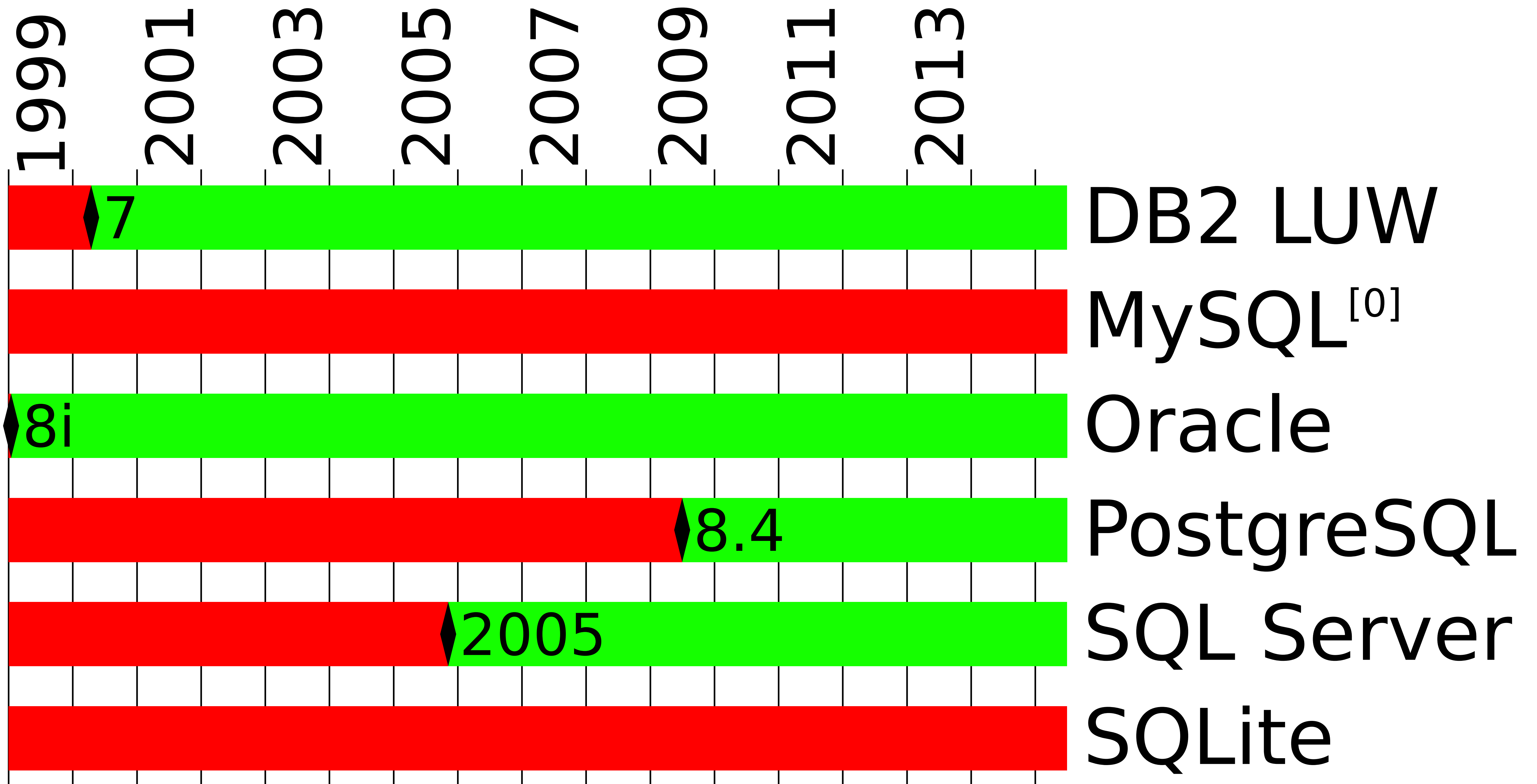
```
SELECT txid, value,
       SUM(value)
       OVER(ORDER BY txid
            ROWS
            BETWEEN UNBOUNDED PRECEDING
                AND CURRENT ROW) bal
  FROM transactions tx1
 WHERE acnt = ?
 ORDER BY txid
```

OVER Since SQL:2003

With `OVER (ORDER BY …)` a new type of functions makes sense:

- `ROW_NUMBER`
- Ranking functions:
  `RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST`

OVER Availability (SQL:2003)



[0] Feature request #35893 from 2008-04-08

WITHIN GROUP

WITHIN GROUP Before SQL:2003

Getting the median:

```
SELECT d1.val
  FROM data d1
  JOIN data d2
    ON (d1.val < d2.val
        OR (d1.val=d2.val AND d1.id<d2.id))
 GROUP BY d1.val
HAVING count(*) =
       (SELECT FLOOR(COUNT(*)/2)
          FROM data)
```

WITHIN GROUP Since SQL:2003

SQL:2003 introduced ordered-set functions...

Median

```
SELECT PERCENTILE_DISC(0.5
        WITHIN GROUP (ORDER BY val)
```

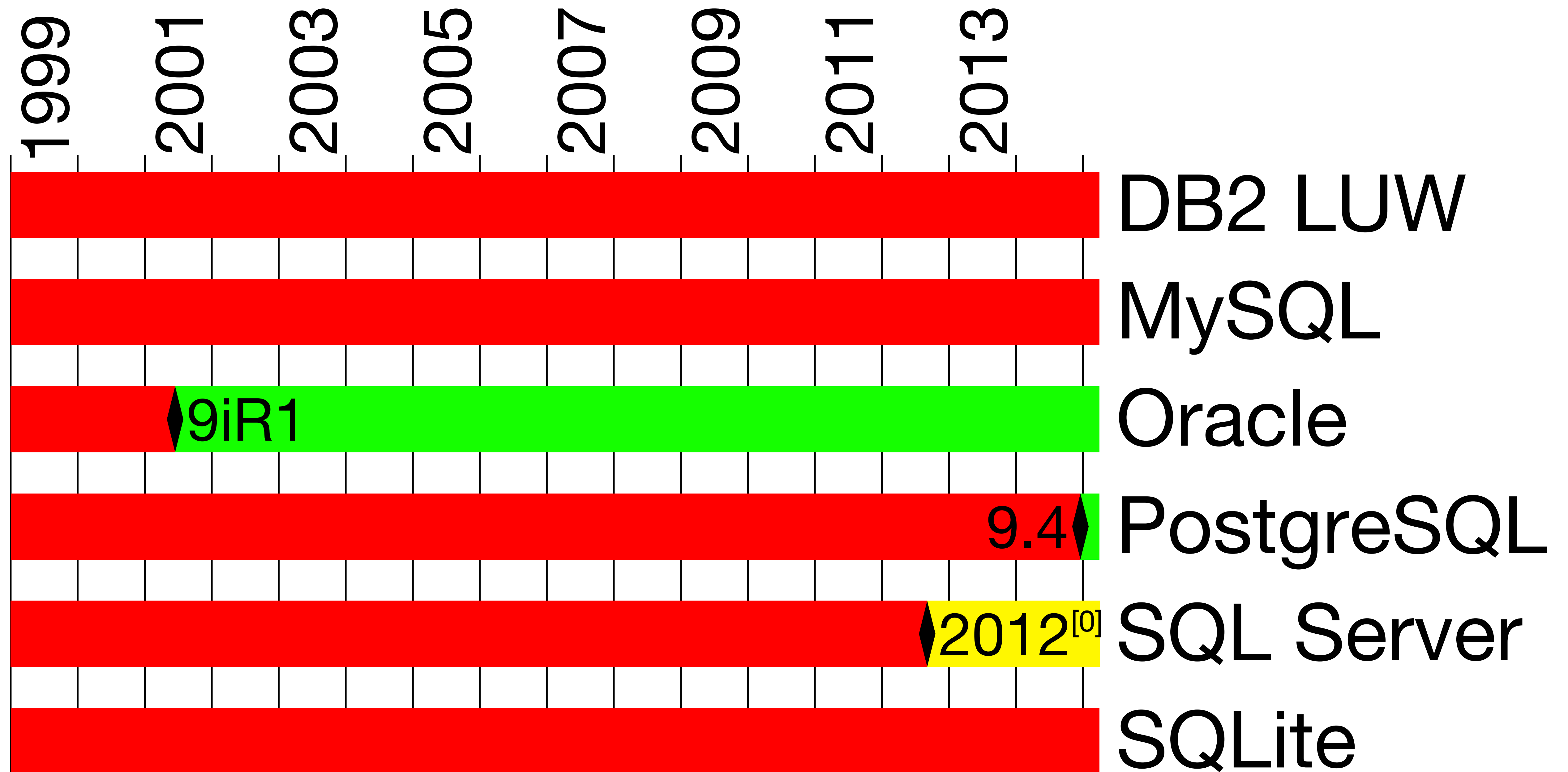
Which value?

WITHIN GROUP Since SQL:2003

SQL:2003 introduced ordered-set functions...

```
SELECT PERCENTILE_DISC(0.5)
       WITHIN GROUP (ORDER BY val)
  FROM data
```

WITHIN GROUP Availability



[0] Only as window function (OVER required). Feature request 728969 closed as "won't fix"

SQL:2008

OVER

OVER Before SQL:2008

Calculate the difference to a previous row:

```
WITH numbered_data AS (
 SELECT *,
        ROW_NUMBER() OVER(ORDER BY x) rn
   FROM data)
```

OVER Before SQL:2008

Calculate the difference to a previous row:

```
WITH numbered_data AS (
 SELECT *,
        ROW_NUMBER() OVER(ORDER BY x) rn
   FROM data)
SELECT cur.*, cur.balance-prev.balance
  FROM      numbered_data cur
  LEFT JOIN numbered_data prev
    ON (cur.rn = prev.rn-1)
```

OVER Since SQL:2008

SQL:2008 can access other rows directly:

```
SELECT *, balance - LAG(balance)
                    OVER(ORDER BY x)
  FROM data
```

OVER Since SQL:2008

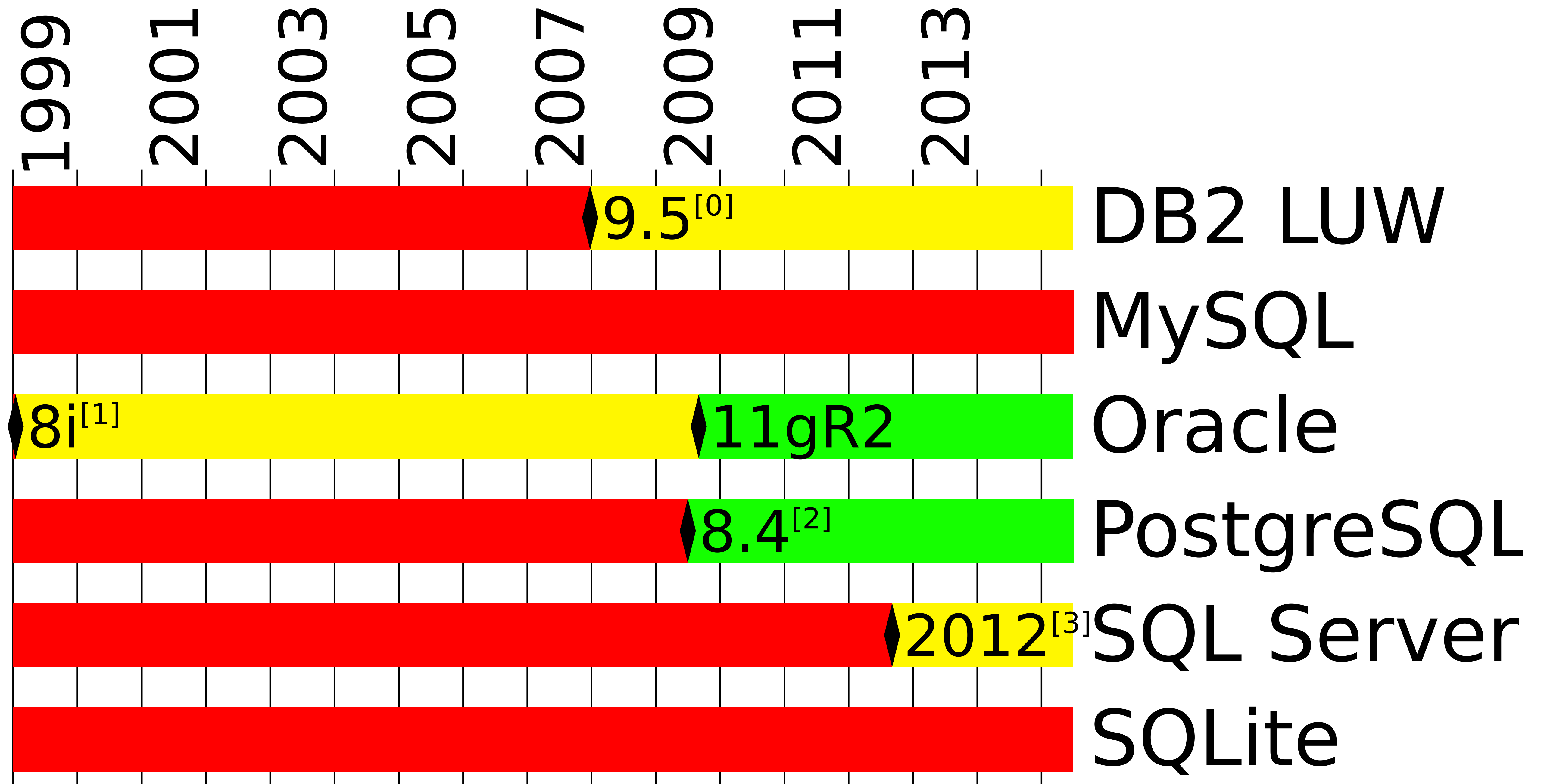
SQL:2008 can access other rows directly:

```
SELECT *, balance - LAG(balance)
                    OVER(ORDER BY x)
  FROM data
```

Available functions:

```
LEAD / LAG
FIRST_VALUE / LAST_VALUE
NTH_VALUE(col, n) FROM FIRST/LAST
                  RESPECT/IGNORE NULLS
```

OVER Availability (SQL:2008)



[0] No NTH_VALUE as of DB2 LUW 10.5
[1] No NTH_VALUE and IGNORE NULLS until Oracle release 11gR2
[2] No support for IGNORE NULLS and FROM LAST as of PostgreSQL 9.4
[3] No NTH_VALUE as of SQL Server 2014

FETCH FIRST

FETCH FIRST Before SQL:2008

Limit the number of selected rows:

```
SELECT *
  FROM (SELECT *,
          ROW_NUMBER() OVER(ORDER BY x) rn
   FROM data) numbered_data
 WHERE rn <=10
```

FETCH FIRST Before SQL:2008

Limit the number of selected rows:

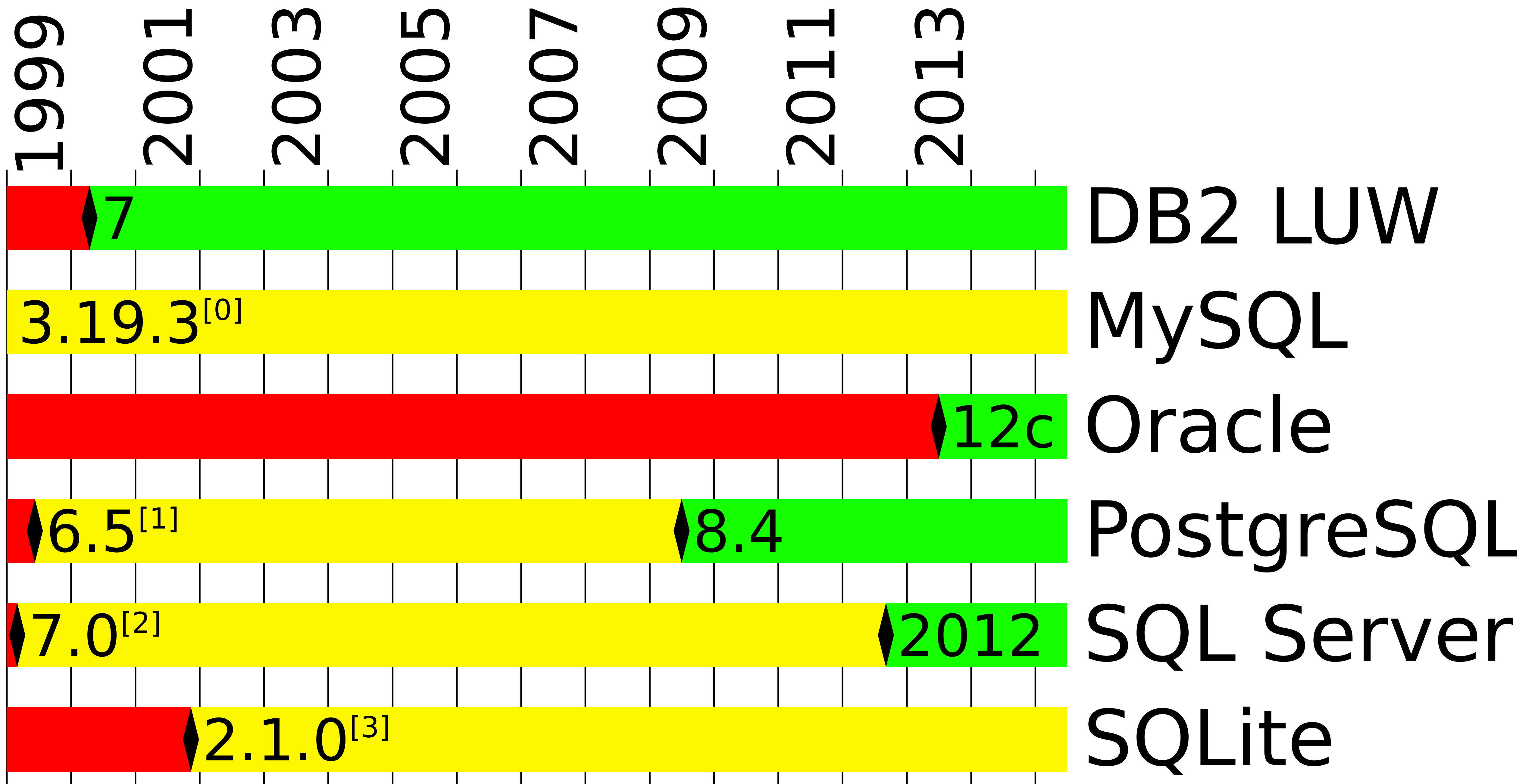
```
SELECT *
  FROM (SE
          RO                          DER BY x) rn
   FROM da
 WHERE rn <=
```

FETCH FIRST Since SQL:2008

SQL:2008 has `FETCH FIRST n ROWS ONLY`:

```
SELECT *
  FROM data
 ORDER BY x
 FETCH FIRST 10 ROWS ONLY
```

FETCH FIRST *Availability*



[0] Earliest mention of `LIMIT`. Probably inherited from mSQL
[1] Functionality available using `LIMIT`
[2] `SELECT TOP n ...` SQL Server 2000 also supports expressions and bind parameters
[3] Functionality available using `LIMIT`

SQL:2011

OFFSET

OFFSET Before SQL:2011

Skip 10 rows, then deliver only the next 10:

```
SELECT *
  FROM (SELECT *,
               ROW_NUMBER() OVER(ORDER BY x) rn
         FROM data
       FETCH FIRST 20 ROWS ONLY
) numbered_data
 WHERE rn > 10
```

OFFSET Since SQL:2011

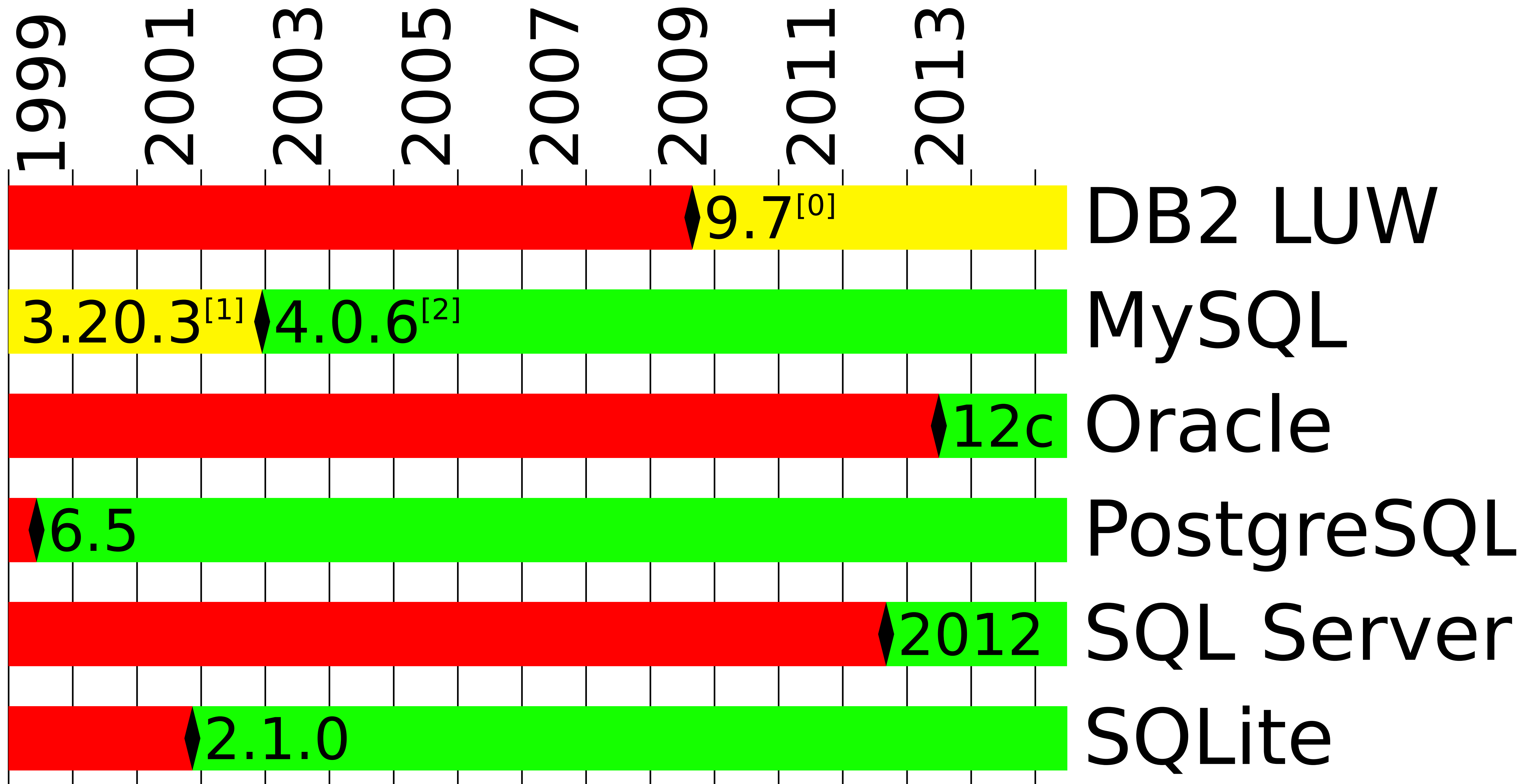
SQL:2011 introduced `OFFSET`, unfortunately:

```
SELECT *
  FROM data
 ORDER BY x
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY
```

OFFSET is EVIL

http://use-the-index-luke.com/no-offset

OFFSET Availability (SQL:2011)



[0] Requires enabling the MySQL compatibility vector: `db2set DB2_COMPATIBILITY_VECTOR=MYS`
[1] `LIMIT [offset,] limit`: "With this it's easy to do a poor man's next page/previous page WWW application."
[2] The release notes say "Added PostgreSQL compatible `LIMIT` syntax"

AS OF

AS OF Before SQL:2011

`INSERT`
`UPDATE`
`DELETE`

are

DESTRUCTIVE

AS OF Since SQL:2011

Tables can be system versioned:

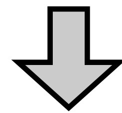
```
CREATE TABLE t (...,
 start_ts TIMESTAMP(9) GENERATED
          ALWAYS AS ROW START,
 end_ts   TIMESTAMP(9) GENERATED
          ALWAYS AS ROW END,

 PERIOD FOR SYSTEM TIME (start_ts, end_ts)
) WITH SYSTEM VERSIONING
```

AS OF Since SQL:2011

```
INSERT ... (ID, DATA) VALUES (1, 'X')
```

AS OF Since SQL:2011

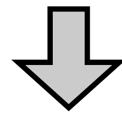
```
INSERT ... (ID, DATA) VALUES (1, 'X')
```



ID	Data	start_ts	end_ts
1	X	10:00:00	

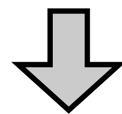
AS OF Since SQL:2011

```
INSERT ... (ID, DATA) VALUES (1, 'X')
```



ID	Data	start_ts	end_ts
1	X	10:00:00	

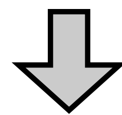
```
UPDATE ... SET DATA = 'Y' ...
```



ID	Data	start_ts	end_ts
1	X	10:00:00	11:00:00
1	Y	11:00:00	

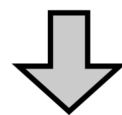
AS OF Since SQL:2011

```
UPDATE ... SET DATA = 'Y' ...
```



ID	Data	start_ts	end_ts
1	X	10:00:00	11:00:00
1	Y	11:00:00	

```
DELETE ... WHERE ID = 1
```



ID	Data	start_ts	end_ts
1	X	10:00:00	11:00:00
1	Y	11:00:00	12:00:00

AS OF Since SQL:2011

ID	Data	start_ts	end_ts
1	X	10:00:00	11:00:00
1	Y	11:00:00	12:00:00

Although multiple versions exist, only the “current” one is visible per default.

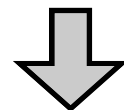
After 12:00:00, `SELECT * FROM t` doesn’t return anything anymore.

AS OF Since SQL:2011

ID	Data	start_ts	end_ts
1	X	10:00:00	11:00:00
1	Y	11:00:00	12:00:00

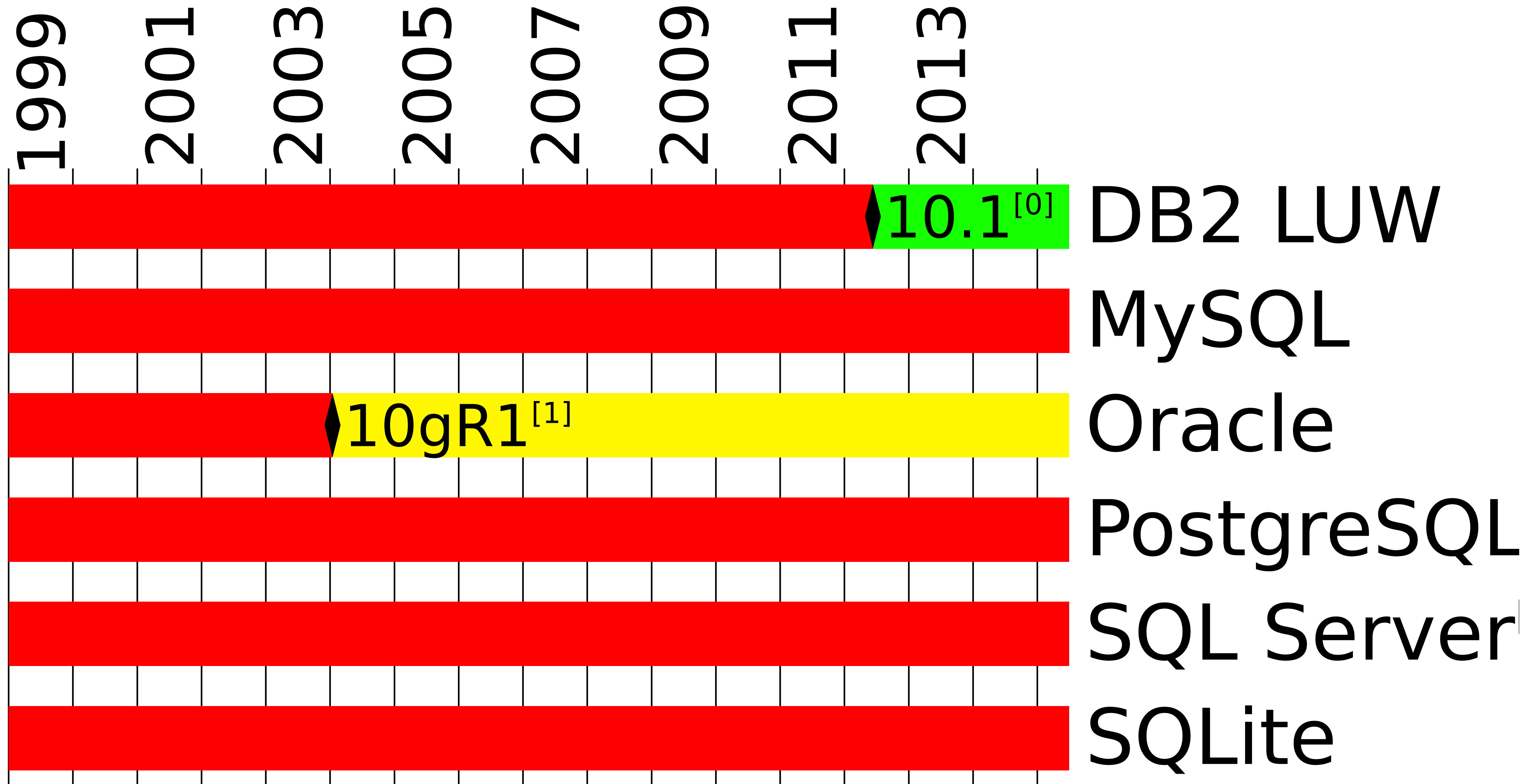
With **AS OF** you can query anything you like:

```
SELECT *
  FROM t FOR SYSTEM_TIME AS OF
       TIMESTAMP '2015-04-02 10:30:00'
```



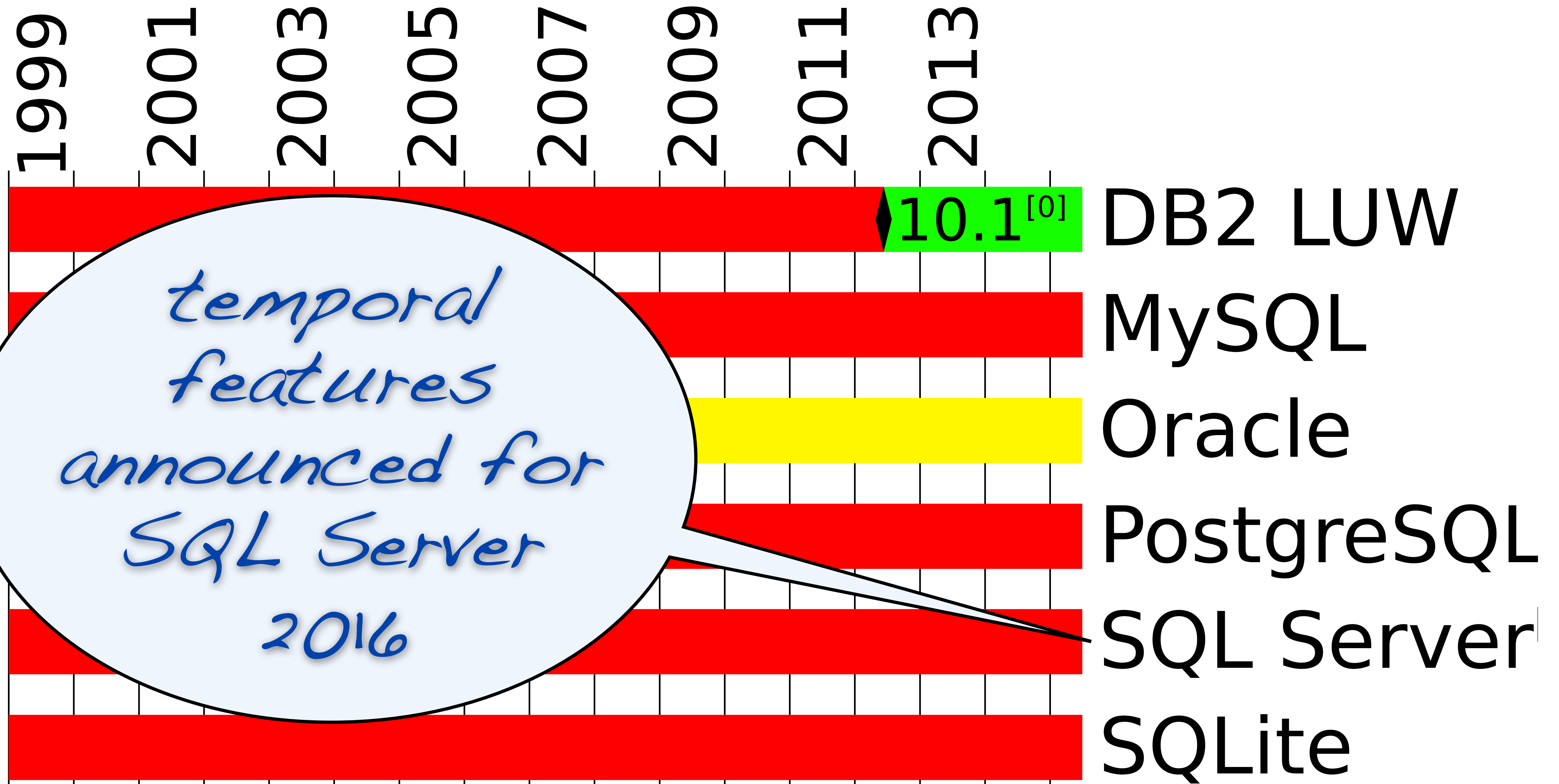
ID	Data	start_ts	end_ts
1	X	10:00:00	11:00:00

SYSTEM_TIME AS OF Availability



[0] Third column required (tx id), history table required.
[1] Functionality available using Flashback
[2] "Temporal Databases: Track historical changes" mentioned in SQL Server 2016 datasheet

SYSTEM_TIME AS OF *Availability*



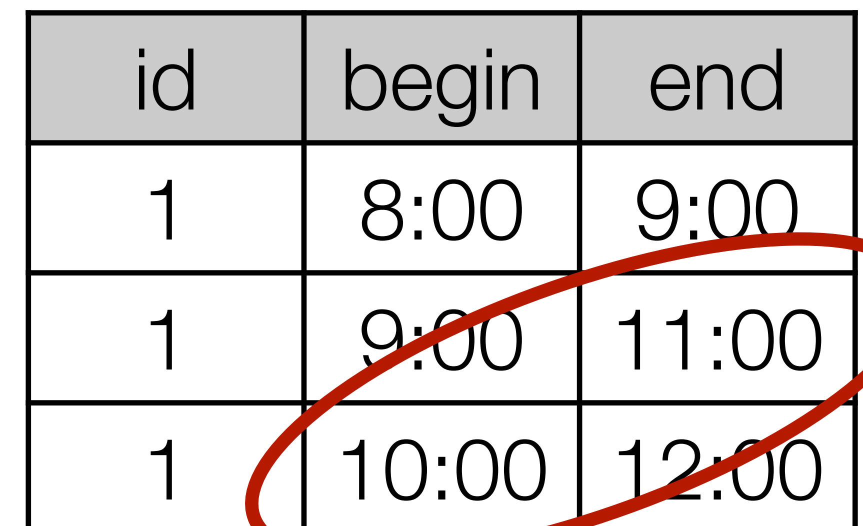
[0] Third column required (tx id), history table required.
[1] Functionality available using Flashback
[2] "Temporal Databases: Track historical changes" mentioned in SQL Server 2016 datasheet

WITHOUT OVERLAPS

WITHOUT OVERLAPS Before SQL:2011

Prior SQL:2011 it was not possible to define constraints that avoid overlapping periods.

Workarounds are possible, but no fun: `CREATE TRIGGER`



id	begin	end
1	8:00	9:00
1	9:00	11:00
1	10:00	12:00

WITHOUT OVERLAPS Since SQL:2011

SQL:2011 introduced temporal and bi-temporal features —e.g., for constraints:

```
PRIMARY KEY (id, period WITHOUT OVERLAPS)
```

PostgreSQL 9.2 introduced range types and "exclusive constraints" which can accomplish the same effect:

```
EXCLUDE USING gist
        (id WITH =, period WITH &&)
```

Temporal/Bi-Temporal SQL

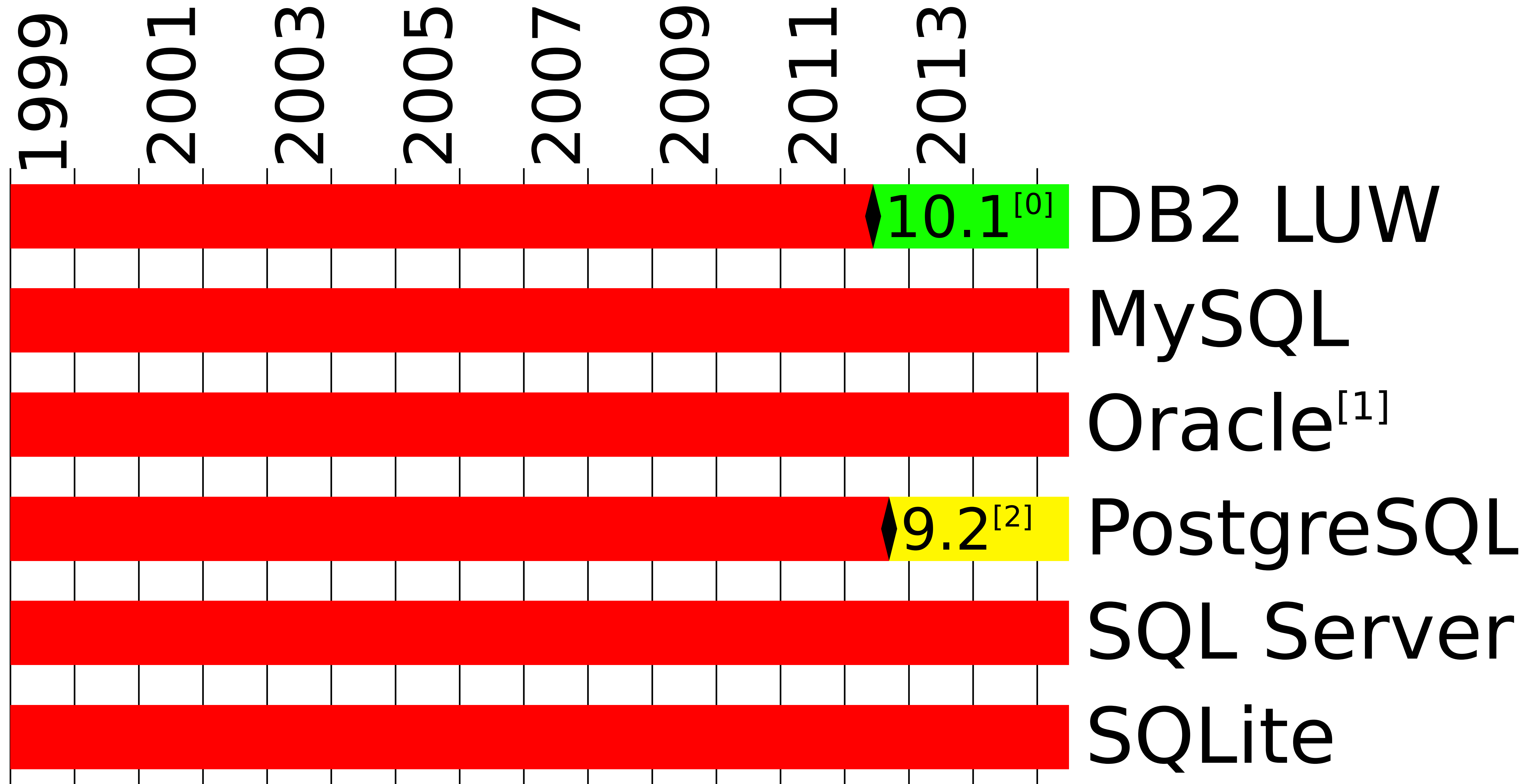
SQL:2011 goes way further.

Please read these papers to get the idea:

Temporal features in SQL:2011
http://cs.ulb.ac.be/public/_media/teaching/infoh415/tempfeaturessql2011.pdf

What's new in SQL:2011?
http://www.sigmod.org/publications/sigmod-record/1203/pdfs/10.industry.zemke.pdf

WITHOUT OVERLAPS Availability



[0] Minor differences: `PERIOD` without `FOR`; period name must be `BUSINESS_TIME`
[1] Oracle 12c has partial temporal support, but no direct equivalent of `WITHOUT OVERLAPS`
[2] Functionality available using `EXCLUDE` constraints

About @MarkusWinand

Tuning developers for high SQL performance

Training & tuning:
http://winand.at/

Author of:
http://sql-performance-explained.com/

Geeky blog:

http://use-the-index-luke.com