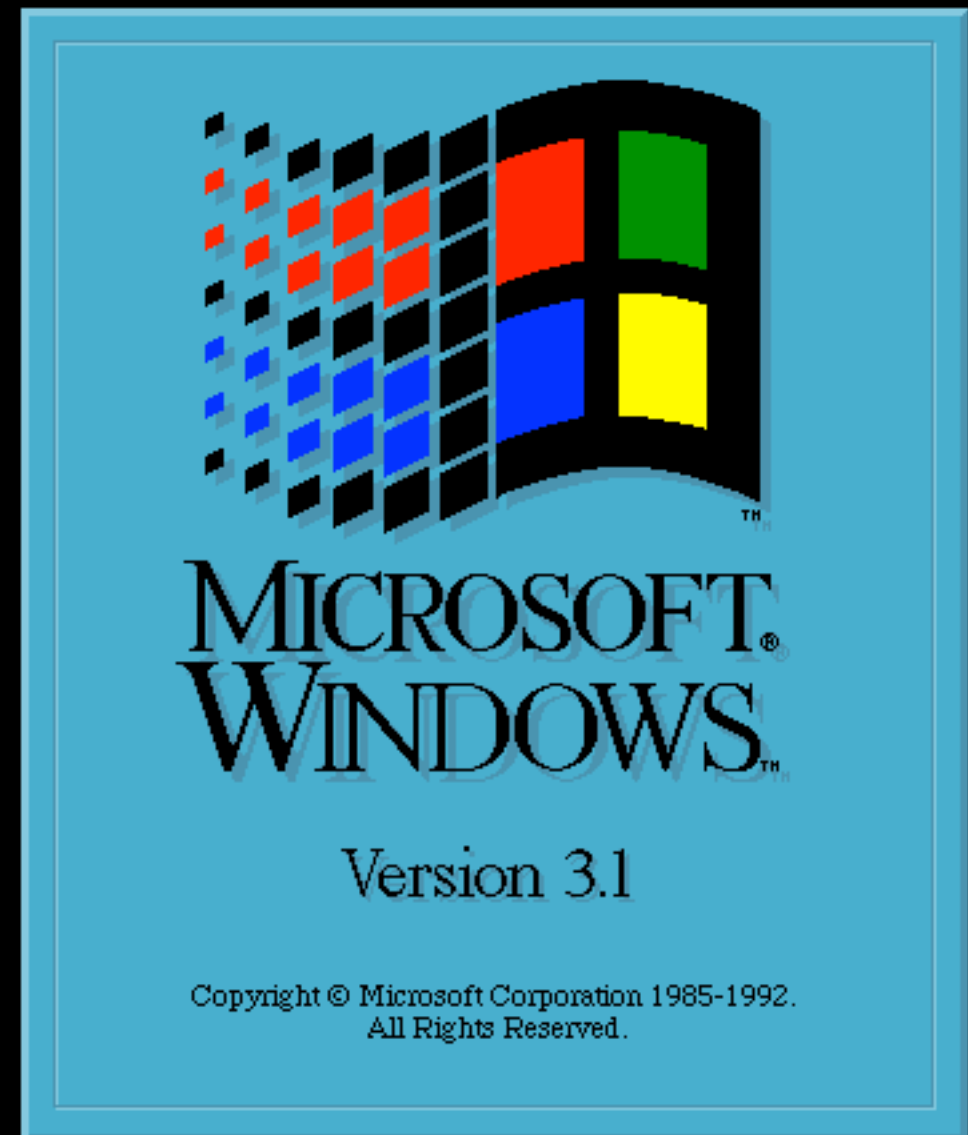


Still using  
Windows 3.1?

So why stick to  
SQL-92?



Modern SQL in PostgreSQL  
@MarkusWinand



SQL: 1999

**LATERAL**

# LATERAL Before SQL:1999

---

Inline views can't refer to outside the view:

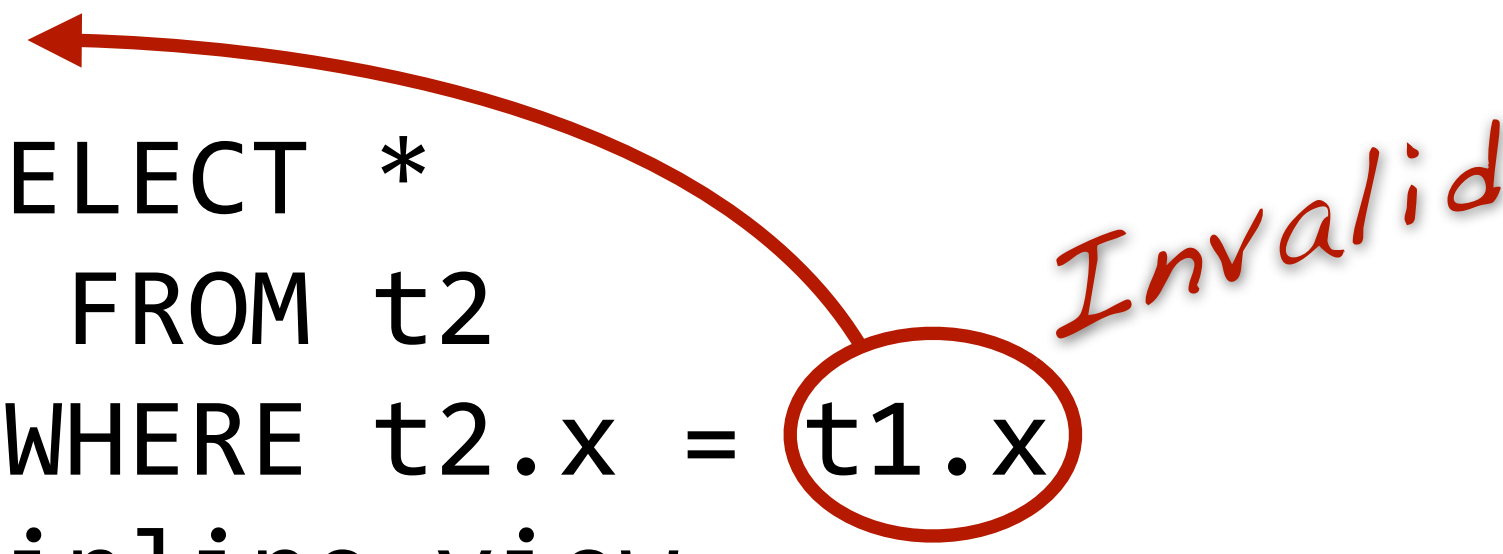
```
SELECT *  
  FROM t1  
 JOIN (SELECT *  
        FROM t2  
        WHERE t2.x = t1.x  
      ) inline_view
```

# 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
```



*Invalid*

# 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)
```

*Belongs  
there*



# 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)
```

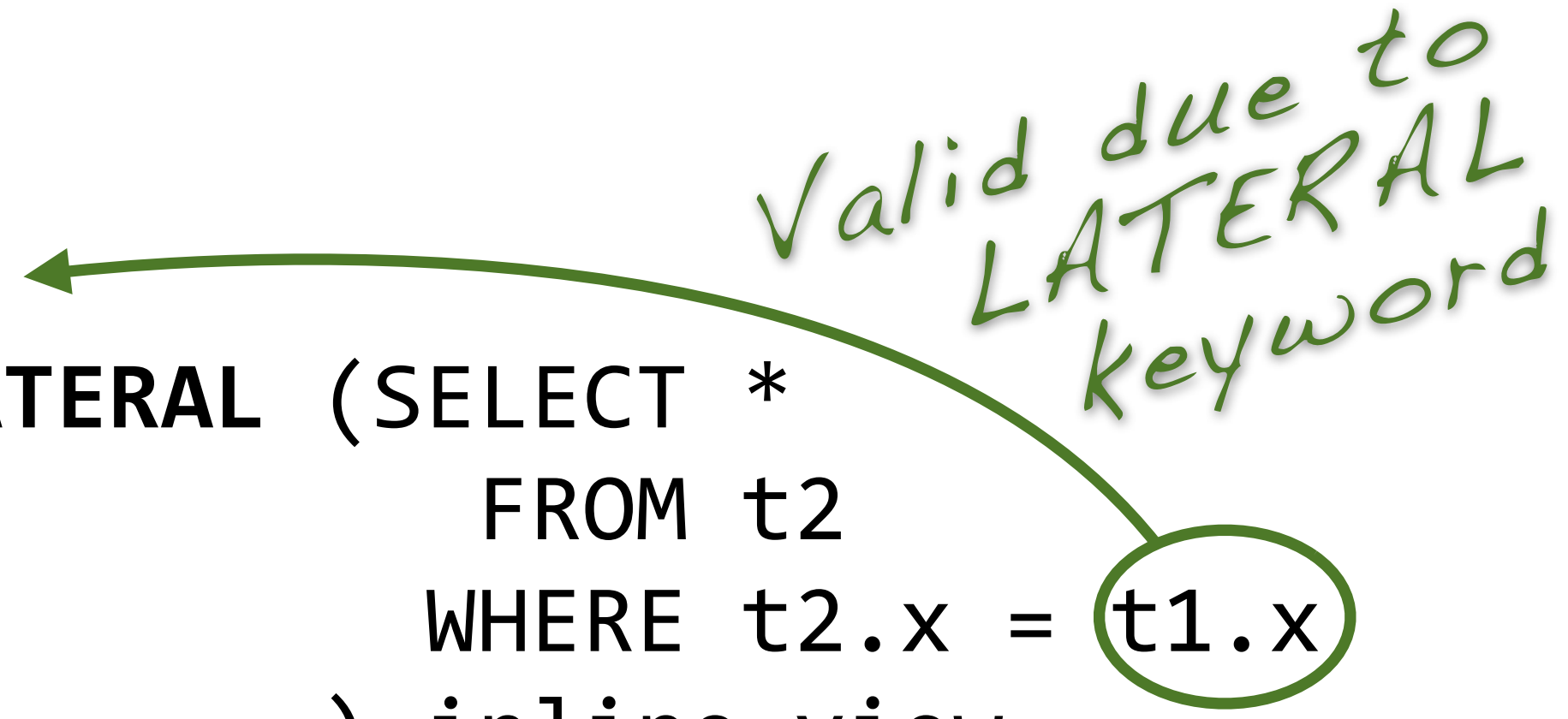


# LATERAL Since SQL:1999

---

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*

# 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*

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.  
The ISO standard foresees TABLE() for this use case.

# 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<sup>7</sup>)

-> Seq Scan on news n  
(time=91218 rows=10<sup>7</sup>)

Planning time: 0.294 ms

Execution time: 236707.261 ms

# 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<sup>7</sup>)

-> Seq Scan on news n  
(time=91218 rows=10<sup>7</sup>)

*Join  
everything*

Planning time: 0.294 ms

Execution time: 236707.261 ms



# LATERAL and Multi-Source Top-N

Limit (time=236707 rows=10) *Sort/Reduce*  
-> Sort (time=236707 rows=10)  
Sort Method: top-N heapsort Mem: 30kB

*Join everything*  
-> Hash Join (time=233800 rows=905029)  
-> Seq Scan on subscriptions s  
(time=369 rows=80)  
-> Hash (time=104986 rows=10<sup>7</sup>)  
-> Seq Scan on news n  
(time=91218 rows=10<sup>7</sup>)

Planning time: 0.294 ms

Execution time: 236707.261 ms

# LATERAL and Multi-Source Top-N

Limit (time=236707 rows=10) *Sort/Reduce*  
-> Sort (time=? rows=10)  
Sort Method Sort Mem: 30kB

*Why  
producing  
900k rows...*

-> Hash Join (time=? 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= $10^7$ )

*Join  
everything*

Planning time: 0.294 ms

Execution time: 236707.261 ms

# LATERAL and Multi-Source Top-N

Limit (time=236707 rows=10) *Sort/Reduce*  
-> Sort (time=? rows=10)  
Sort Method: Merge Sort Mem: 30kB

*Why  
producing  
900k rows...*

-> Hash Join (time=? rows=905029)

*Join  
everything*

-> Seq Scan on subscriptions s  
(time=369 rows=80)

-> Hash (time=104986 rows=80)

-> Seq Scan on  
(time=91218 rows=80)

*...when there  
are only 80  
subscriptions?*

Planning time: 0.294 ms

Execution time: 236707.261 ms

# LATERAL and Multi-Source Top-N

Limit (time=10000000 rows=10000000) Reduce 30kB  
-> Sort  
Sort M  
a  
> Seq Scan on subscriptions s  
(time=369 rows=80)  
> Hash (time=104986 rows=10<sup>7</sup>)  
-> Seq Scan on news n  
(time=91218 rows=10<sup>7</sup>)  
time: 0.294 ms  
time: 236707.261 ms

*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-Source Top-N

---

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 Backward  
using news\_topic\_ts\_id on n

Planning time: 0.161 ms

Execution time: 2.519 ms

# LATERAL and Multi-Source

Limited to 10  
times # of  
subscriptions

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 Backward  
using news\_topic\_ts\_id on n

Planning time: 0.161 ms

Execution time: 2.519 ms

# LATERAL and Multi-Source

Limited to 10  
times # of  
subscriptions

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 Back  
using news\_to

About  
100,000 times  
faster

Planning time: 0.161 ms

Execution time: 2.519 ms



# LATERAL in an Nutshell

---

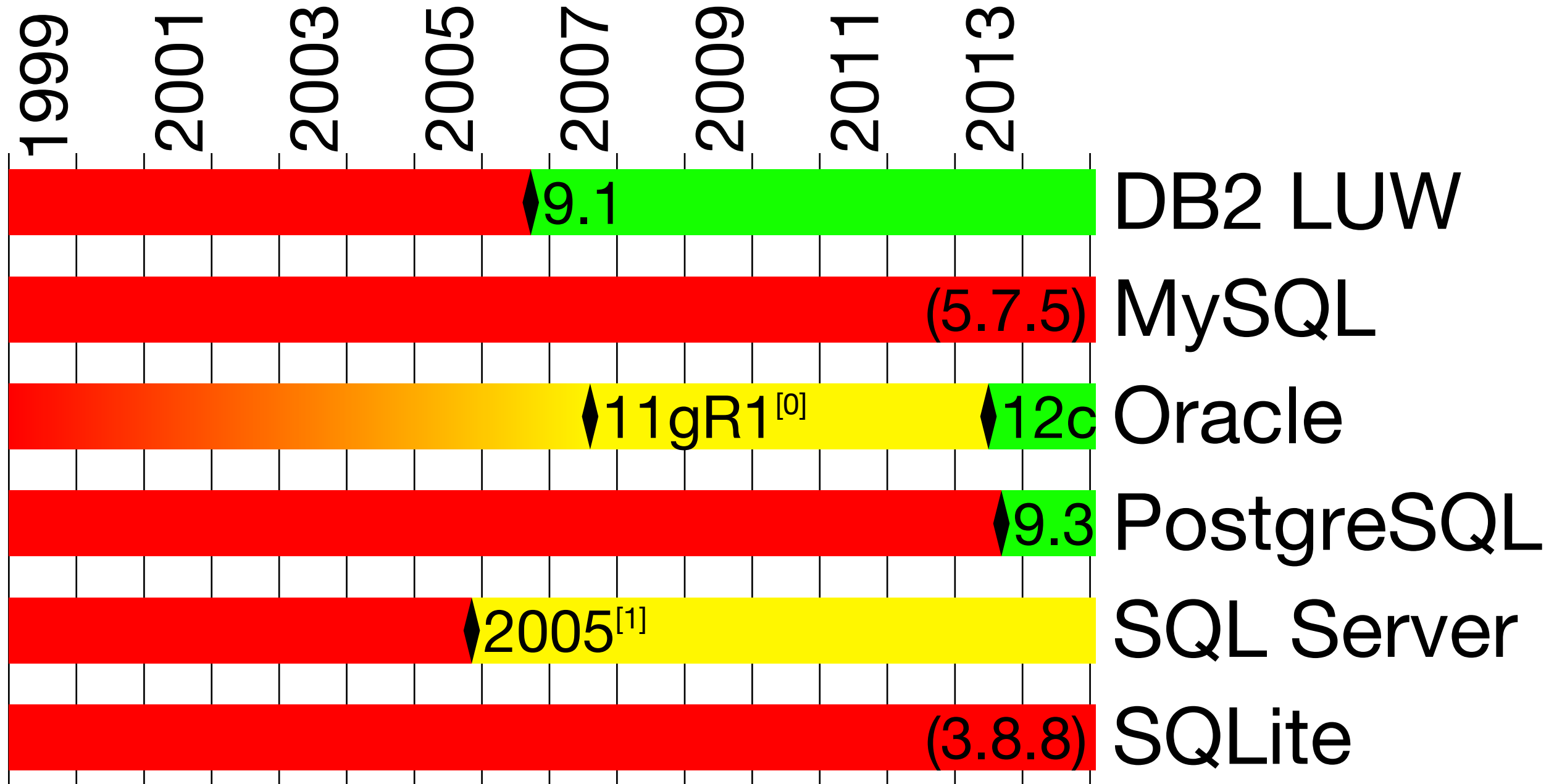
LATERAL is the "for each" loop of SQL

LATERAL plays well with outer joins

LATERAL is an optimization Super-Power

LATERAL handy to join table functions

# LATERAL Availability (SQL:1999)



<sup>[0]</sup> Undocumented. Requires setting trace event 22829.

<sup>[1]</sup> 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:

WITH *Name of CTE and (here optional) column names*

*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 (...)

),

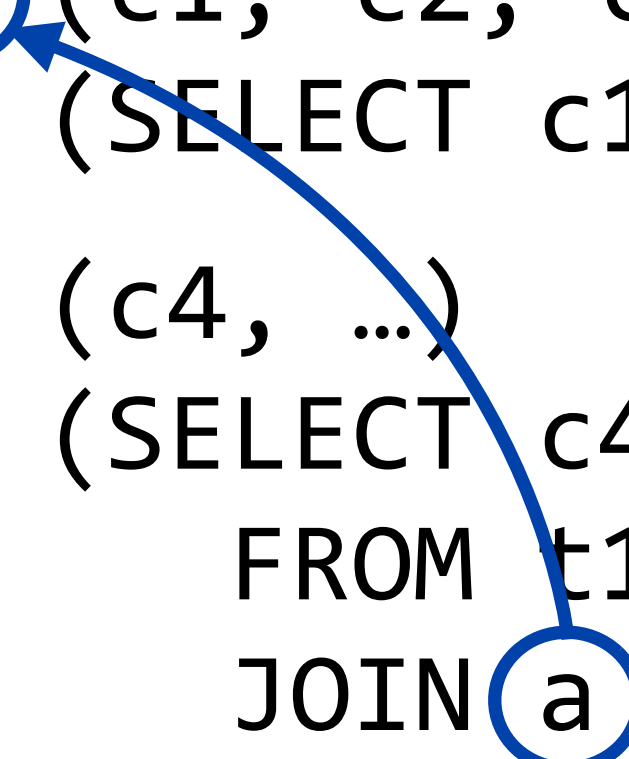
# WITH Since SQL:99

---

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 (...)
```



# WITH Since SQL:99

---

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

*Third CTE*

```
c (...)  
AS (SELECT ... FROM ...)
```

```
SELECT ...  
FROM b JOIN c ON (...)
```

# WITH Since SQL:99

---

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

```
    c (...)  
AS (SELECT ... FROM ..)
```

*No comma!*

```
SELECT ...  
FROM b JOIN c ON (...)
```

# 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*

# 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 (...)
```

*Read  
top down*



# 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 PostgreSQL **WITH** views are more like materialized views:

```
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

---

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

```
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

---

In PostgreSQL, CTEs are more like materialized views

*CTE doesn't know about the outer filter*

```
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;
```

# 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

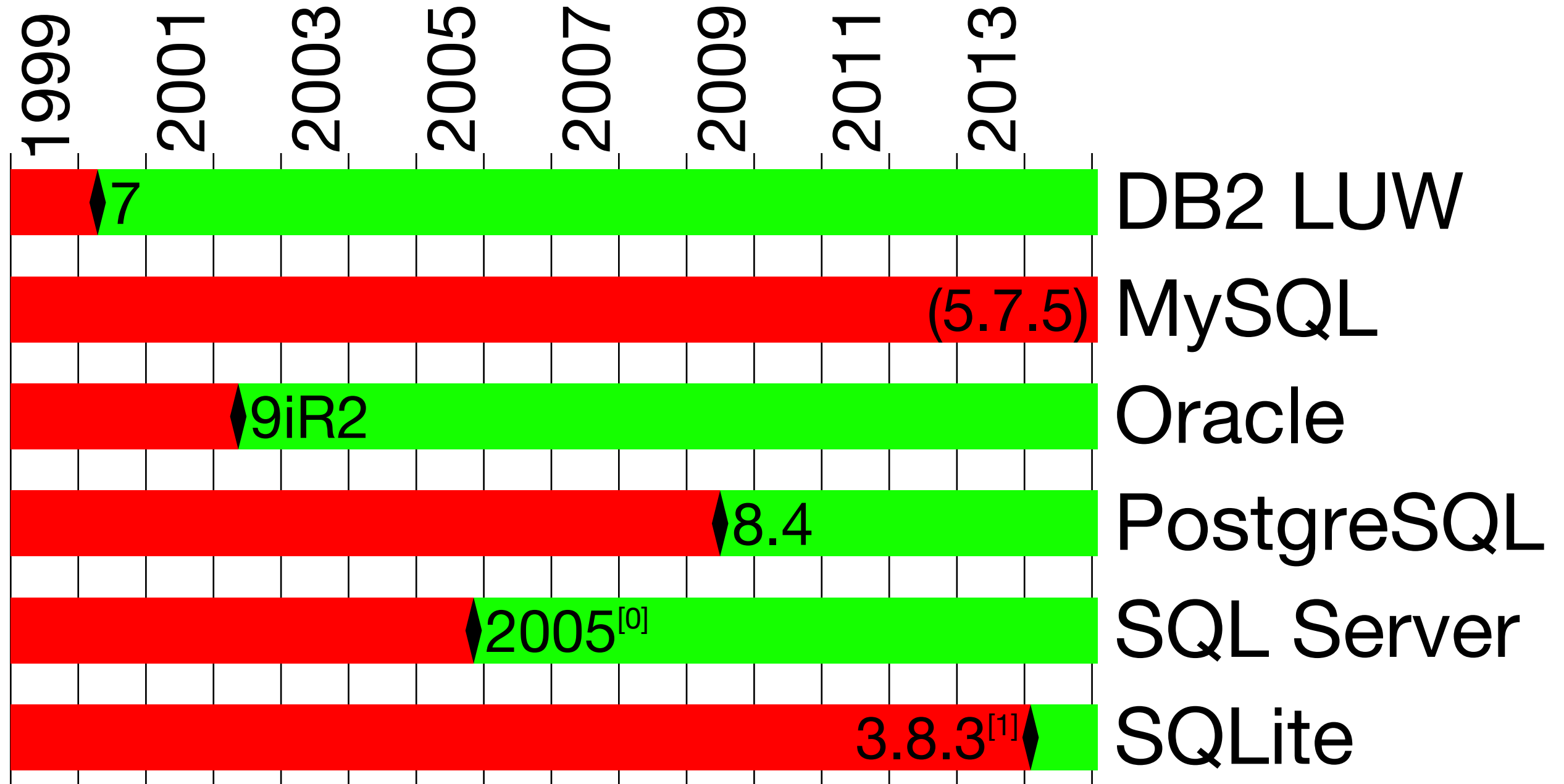
---

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

```
WITH deleted_rows AS (  
    DELETE FROM source  
    RETURNING *  
)  
INSERT INTO destination  
SELECT * FROM deleted_rows;
```

# WITH Availability (SQL:99)

---



<sup>[0]</sup> Only allowed at the very begin of a statement. E.g. `WITH...INSERT...SELECT`.

<sup>[1]</sup> 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) Column list  
mandatory here  
    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 Executed first
      UNION ALL
      SELECT n+1
      FROM cte
      WHERE n < 3)
SELECT * FROM cte;
```

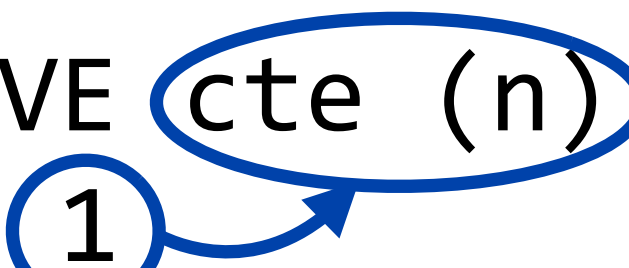
# WITH RECURSIVE Since SQL:99

---

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;
```

*Result sent there*

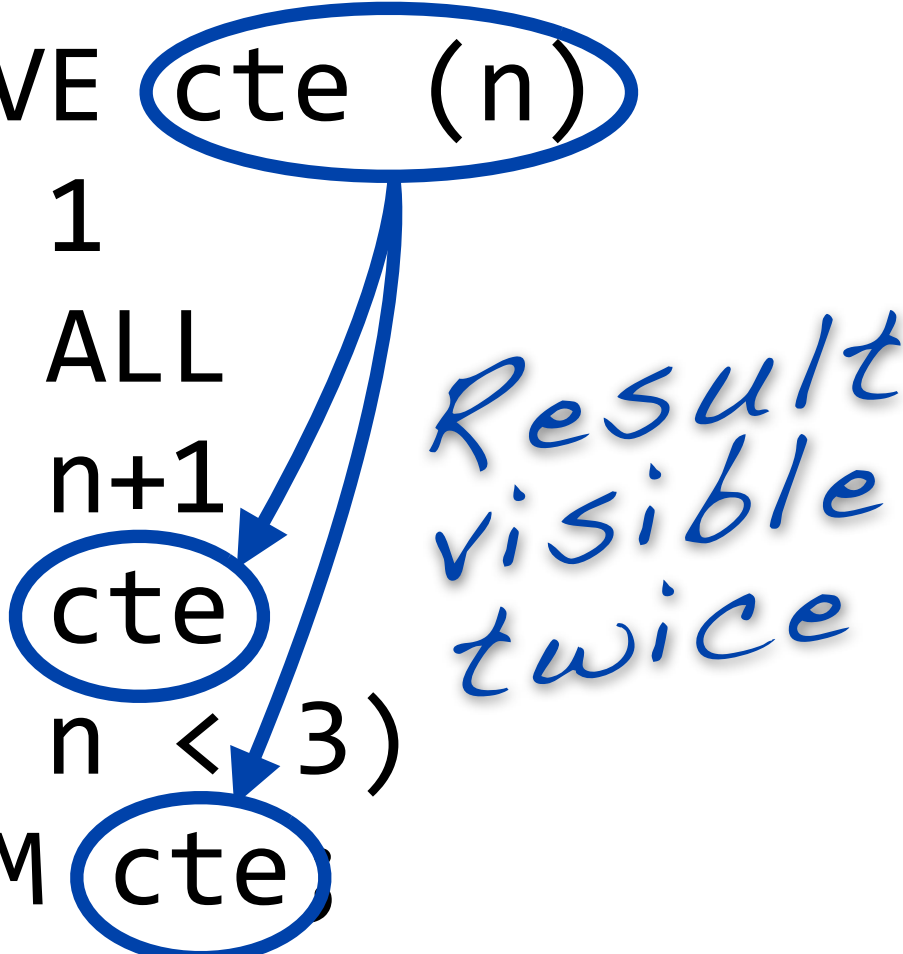


# WITH RECURSIVE Since SQL:99

---

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;
```

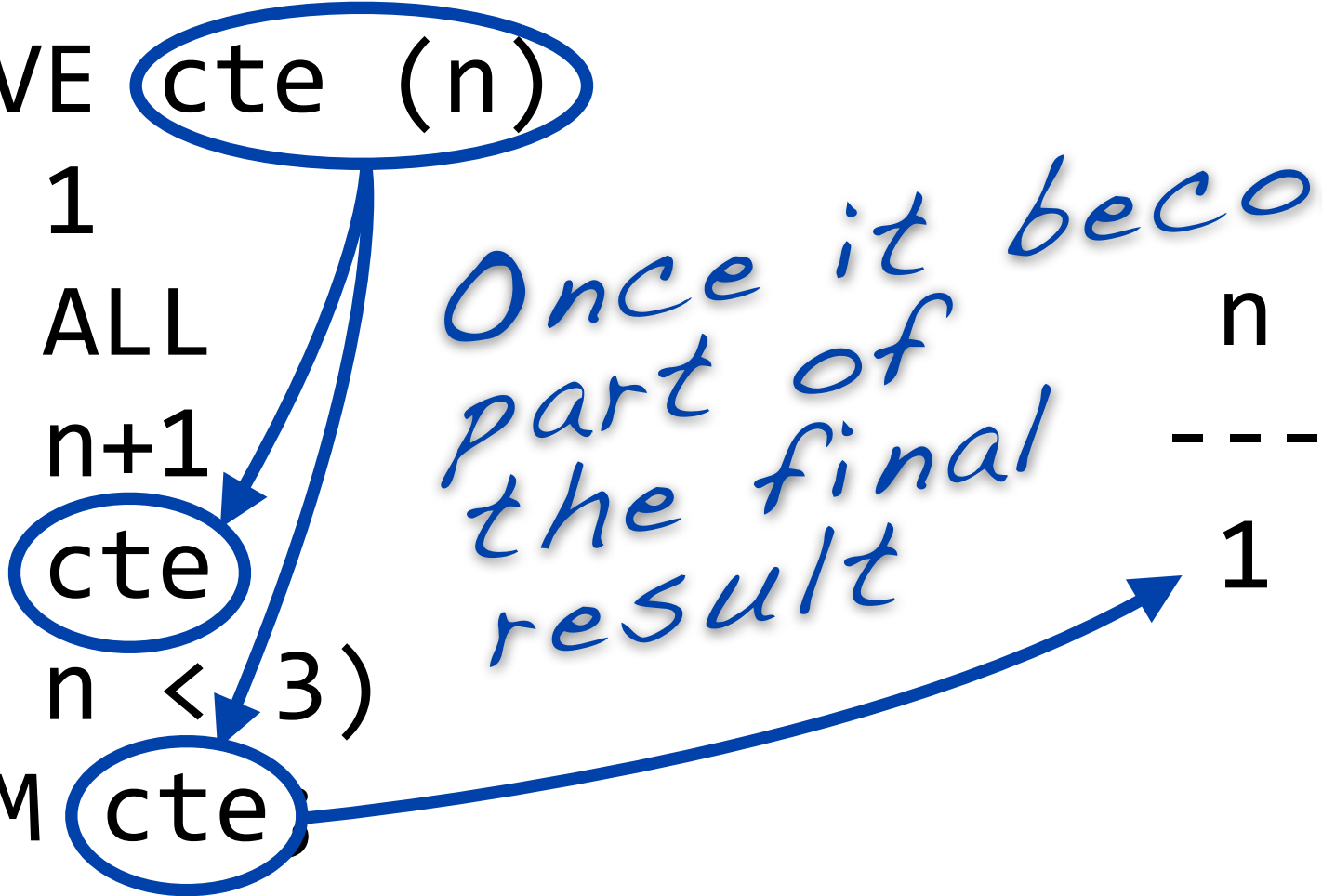


The diagram illustrates the recursive nature of the CTE. Three instances of the CTE name 'cte' are circled in blue. A blue arrow points from the first 'cte' (in the definition) to the 'cte' in the 'FROM' clause of the recursive leg. Another blue arrow points from the first 'cte' to the 'cte' in the final 'SELECT \* FROM cte;' statement. A handwritten blue note 'Result visible twice' is positioned next to these arrows, indicating that the result of the recursive query is visible both in the recursive leg and in the final query 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;
```



*Once it becomes  
part of the final  
result*

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



# 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]**:

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

$$\begin{array}{c} n \\ --- \\ 1 \end{array}$$

# 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!*

$$\frac{n}{1}$$

# 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;
```

*It's a loop!*

n
1
2

# 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=3  
doesn't  
match*

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=3  
doesn't  
match*

*Loop  
terminates*

n
1
2
3

(3 rows)



# 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

---

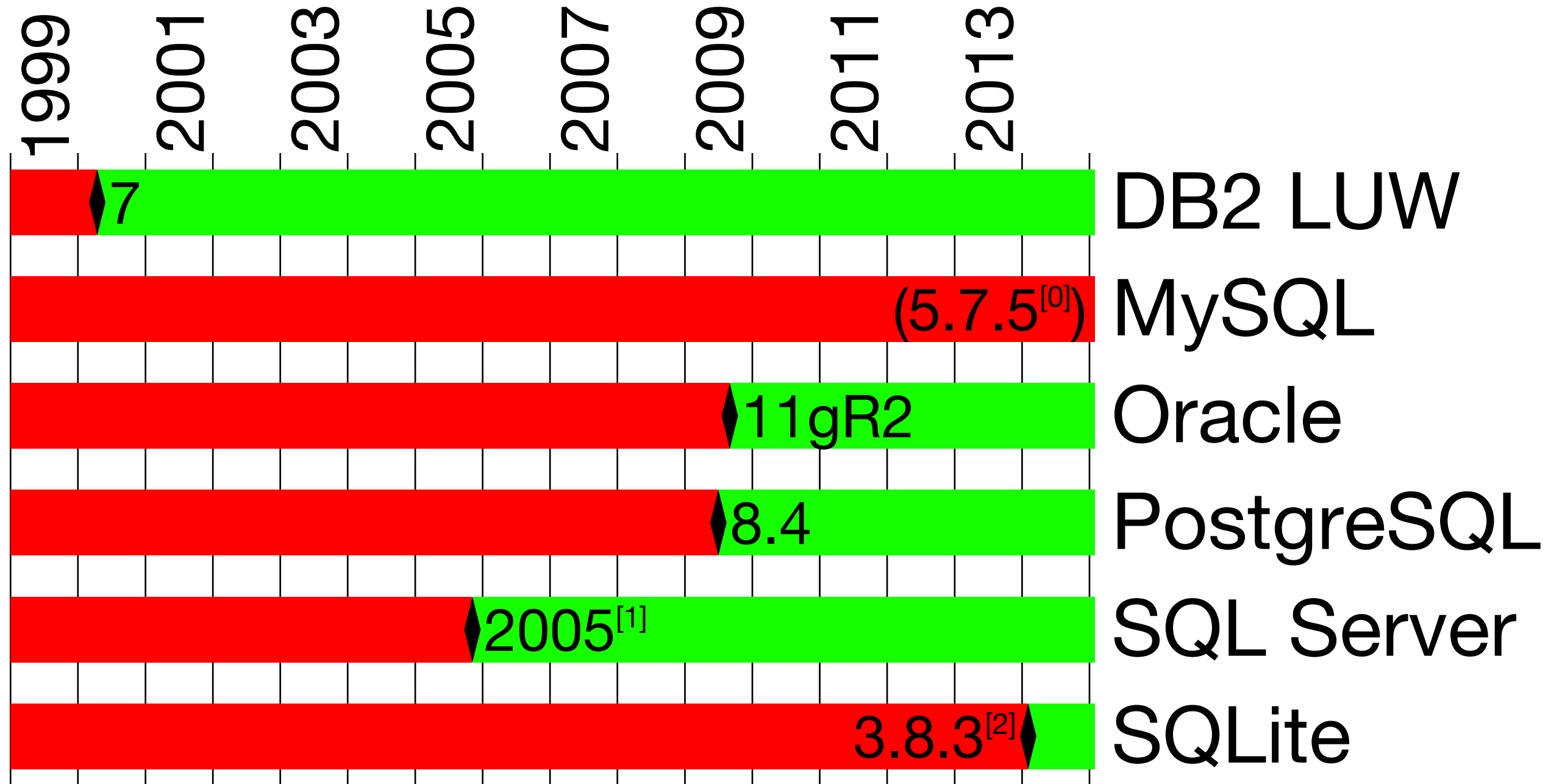
**WITH RECURSIVE** is the `while` of SQL

**WITH RECURSIVE** "supports" infinite loops

(not in SQL Server where `MAXRECURSION` is limited to 32767)

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

# WITH RECURSIVE Availability



<sup>[0]</sup> Feature request #16244 from 2006-01-06

<sup>[1]</sup> Default limit of 100 iterations. `OPTION (MAXRECURSION n)` can push this up to 32767

<sup>[2]</sup> Only for top-level `SELECT` statements

SQL: 2003

**FILTER**

# **FILTER** Before SQL:2003

---

Pivot table: Years on the Y axis, 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,**

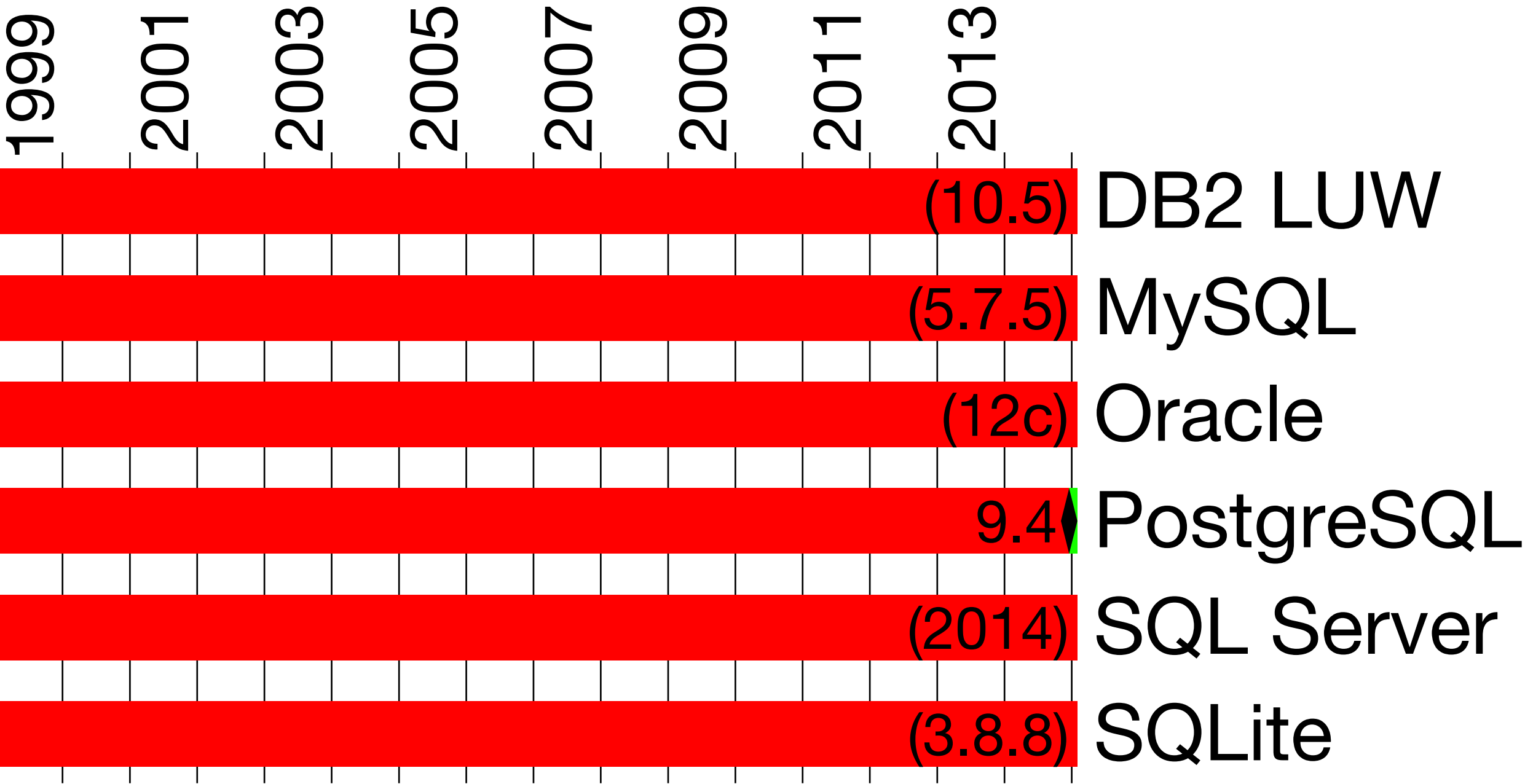
**SUM(sales) FILTER (WHERE MONTH = 2) FEB,**

**...**

**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
  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

---

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

~~WITH total\_salary\_by\_department~~ *WITH intermezzo*

```
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)
WHERE emp.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

---

**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

---

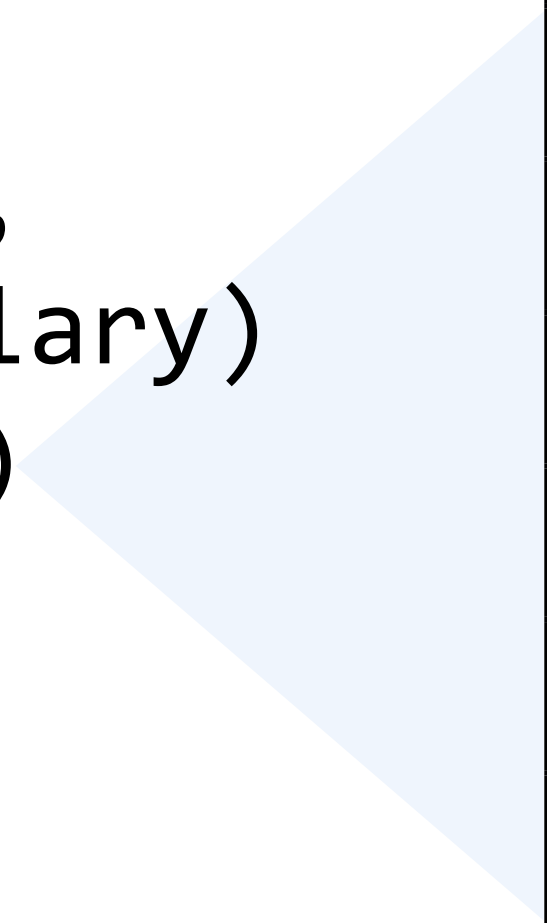
```
SELECT dep,  
       salary,  
       SUM(salary)  
       OVER ()  
FROM emp;
```

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;
```

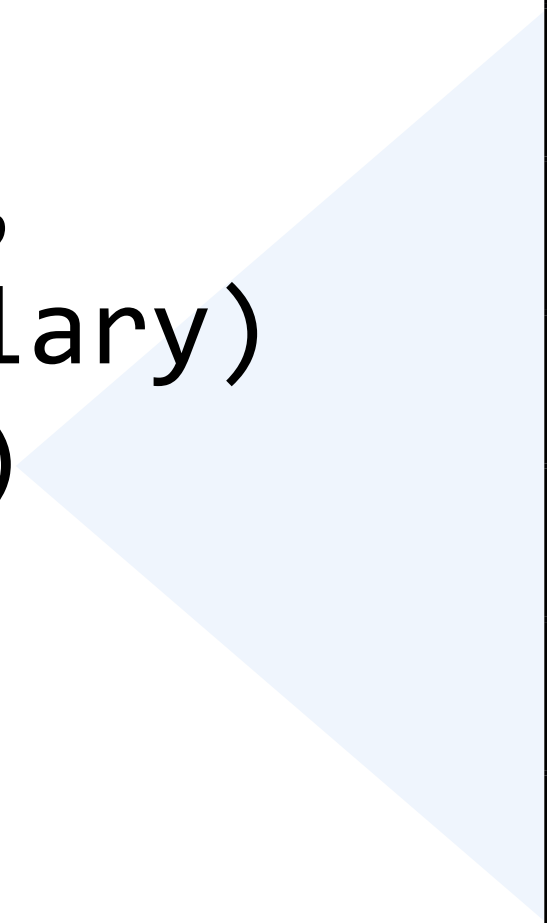


dep	salary
1	1000
22	1000
22	1000
333	1000
333	1000
333	1000

# 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 acct = ?  
          AND tx2.txid <= tx1.txid) bal  
FROM transactions tx1  
WHERE acct = ?  
ORDER BY txid
```

# OVER Before SQL:2003

---

Calculating a running total:

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

# OVER Before SQL:2003

---

Calculating a running total:

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

# OVER Before SQL:2003

---

Calculating a running total:

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



# OVER Before SQL:2003

---

Before SQL:2003 running totals were awkward:

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

# OVER Before SQL:2003

---

Before SQL:2003 running totals were awkward:

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

The only real answer was:

# OVER Before SQL:2003

---

Before SQL:2003 running totals were awkward:

- ▶ Requires a scalar sub-select or self-join
- ▶ Poor maintainability (reparative 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 acct = ?  
ORDER BY txid
```

# OVER Since SQL:2003

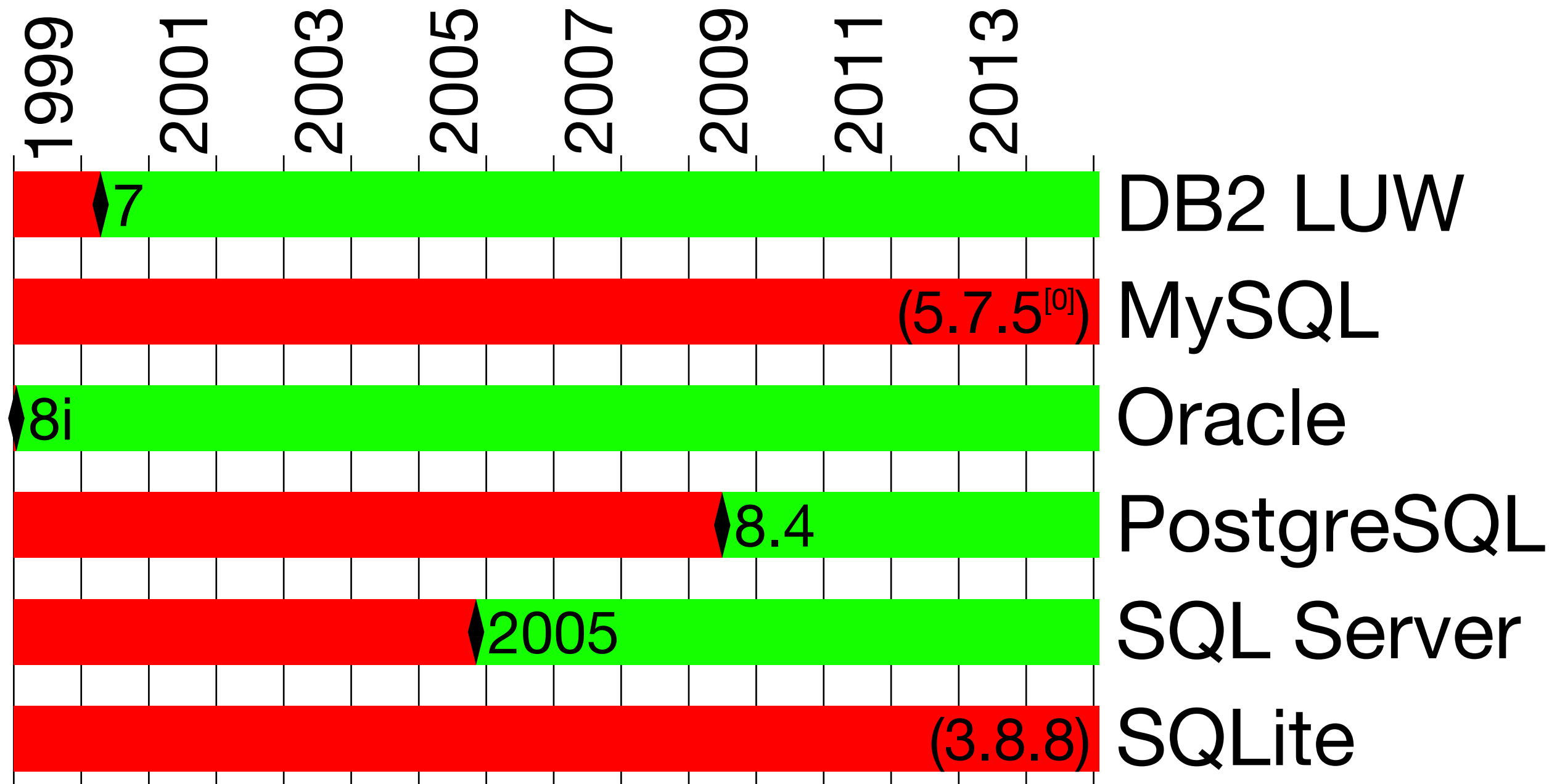
---

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)

---



<sup>[0]</sup> 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...

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

...and hypothetical-set functions to say which rank a hypothetical row would have:

```
SELECT RANK(123)
       WITHIN GROUP (ORDER BY val)
FROM data
```

# WITHIN GROUP Since SQL:2003

---

SQL:2003 introduced ordered-set functions...

*Median*

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

...and hypothetical-set functions to say which rank a hypothetical row would have:

```
SELECT RANK(123)
       WITHIN GROUP (ORDER BY val)
FROM data
```

# WITHIN GROUP Since SQL:2003

---

SQL:2003 introduced ordered-set functions...

*Median*

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

*Which value?*

...and hypothetical-set functions to say which rank a hypothetical row would have:

```
SELECT RANK(123)
       WITHIN GROUP (ORDER BY val)
FROM data
```

# WITHIN GROUP Since SQL:2003

---

SQL:2003 introduced ordered-set functions...

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

...and hypothetical-set functions to say which rank a hypothetical row would have:

```
SELECT RANK(123)
       WITHIN GROUP (ORDER BY val)
FROM data
```

# WITHIN GROUP Since SQL:2003

---

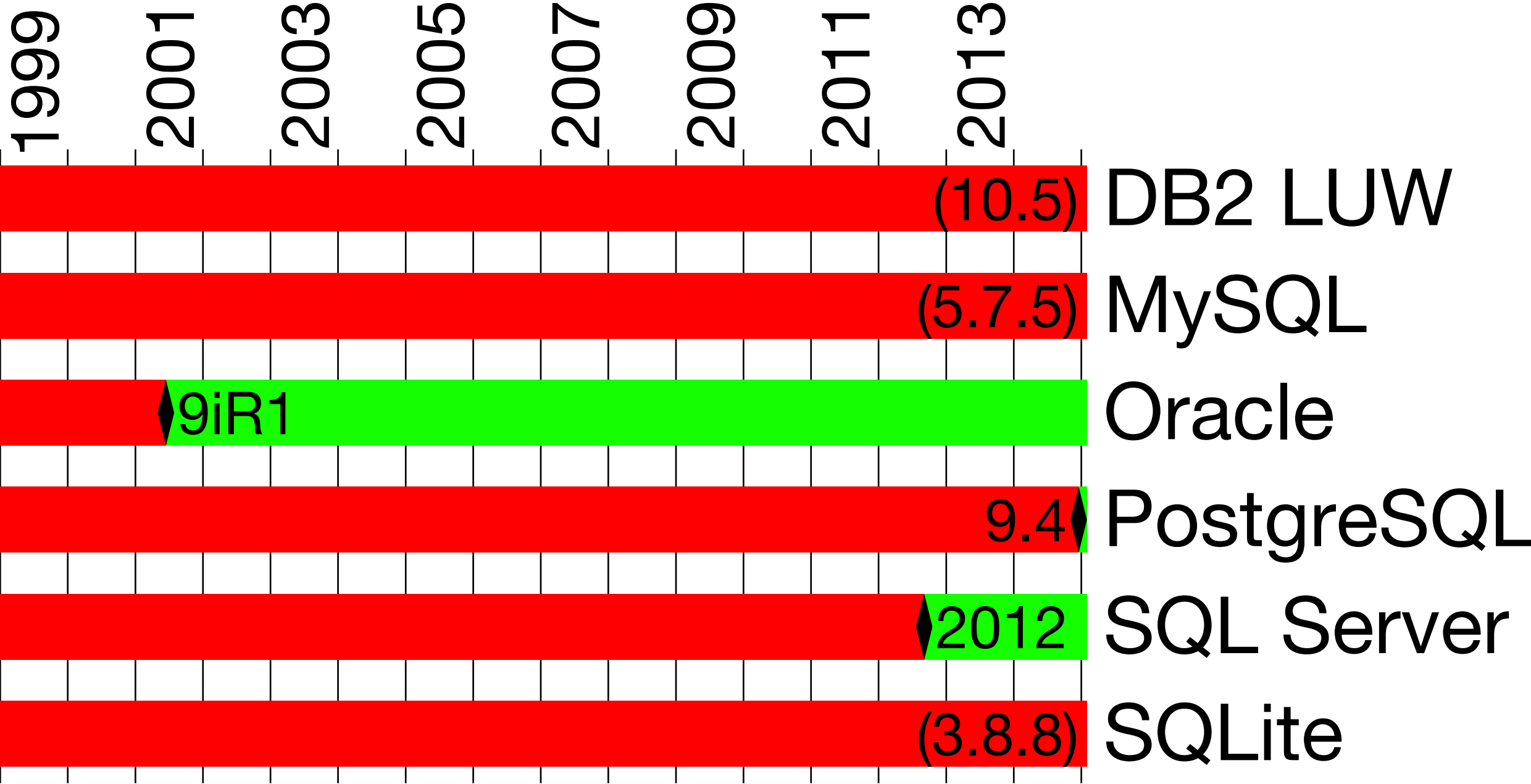
SQL:2003 introduced ordered-set functions...

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

...and hypothetical-set functions to say which rank a hypothetical row would have:

```
SELECT RANK(123)
       WITHIN GROUP (ORDER BY val)
FROM data
```

# WITHIN GROUP Availability



SQL: 2008

**OVER**



# OVER Before SQL:2008

---

Calculate the difference to a previous row:

# 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:

# 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 Since SQL:2008

---

SQL:2008 can access other rows directly:

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

Available functions:

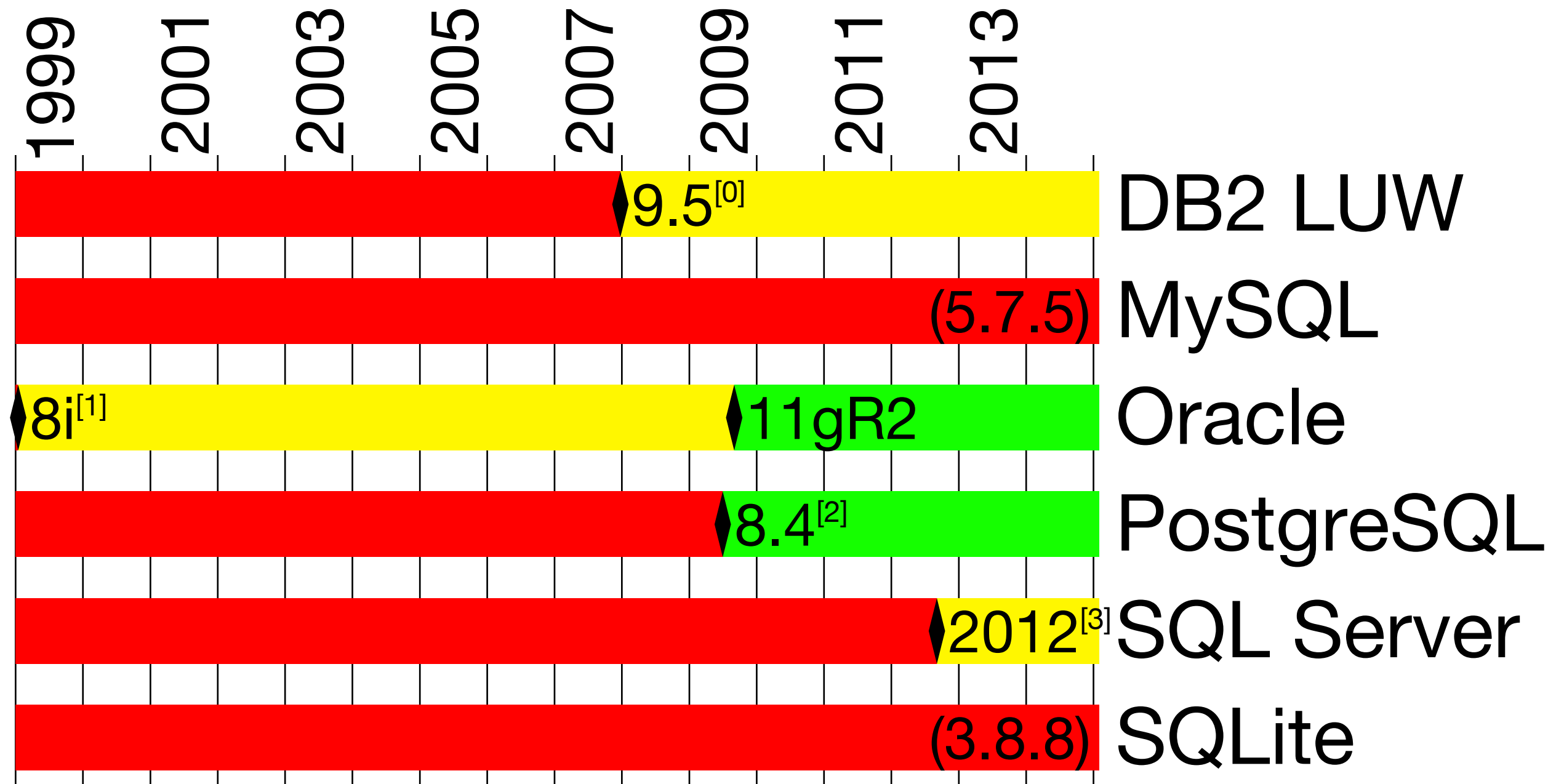
LEAD / LAG

FIRST\_VALUE / LAST\_VALUE

NTH\_VALUE(col, n) FROM FIRST/LAST  
RESPECT/IGNORE NULLS

*Not supported  
by PostgreSQL  
(as of 9.4)*

# OVER Availability (SQL:2008)



<sup>[0]</sup> No NTH\_VALUE as of DB2 LUW 10.5

<sup>[1]</sup> No NTH\_VALUE and IGNORE NULLS until Oracle release 11gR2

<sup>[2]</sup> No support for IGNORE NULLS and FROM LAST as of PostgreSQL 9.4

<sup>[3]</sup> No NTH\_VALUE as of SQL Server 2014



**FETCH FIRST**

# FETCH FIRST Before SQL:2008

---

Limit the number of selected rows:

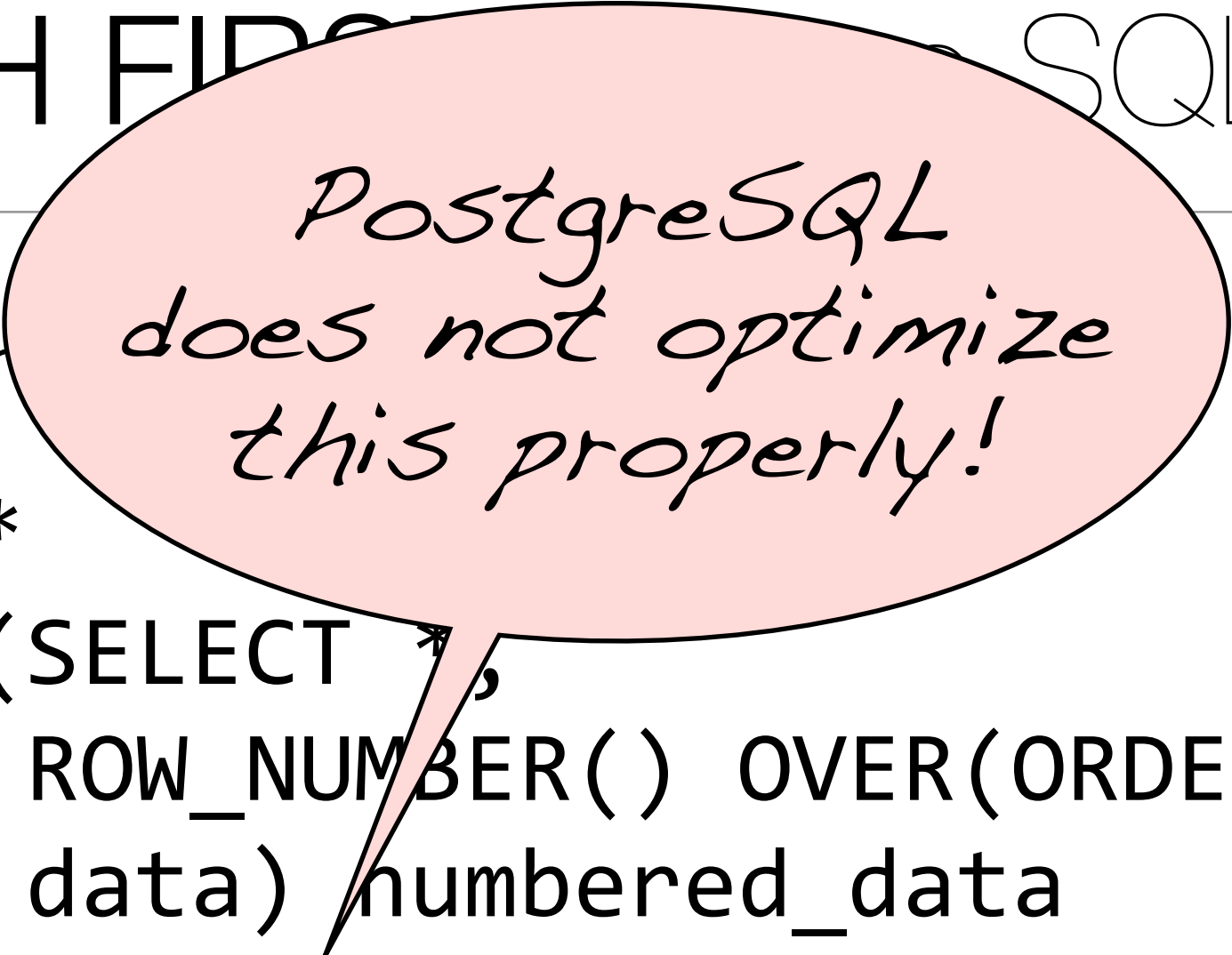
```
SELECT *  
  FROM (SELECT *,  
             ROW_NUMBER() OVER(ORDER BY x) rn  
        FROM data) numbered_data  
WHERE rn <=10
```

# FETCH FIRST SQL:2008

---

Limit the result

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



*PostgreSQL  
does not optimize  
this properly!*

# FETCH FIRST Before SQL:2008

---

Limit the number of selected rows:

```
SELECT *  
  FROM (SE  
        RO  
  FROM da  
 WHERE rn <=
```

ORDER BY x) rn

*Dammit!  
Let's take  
LIMIT  
(or TOP)*

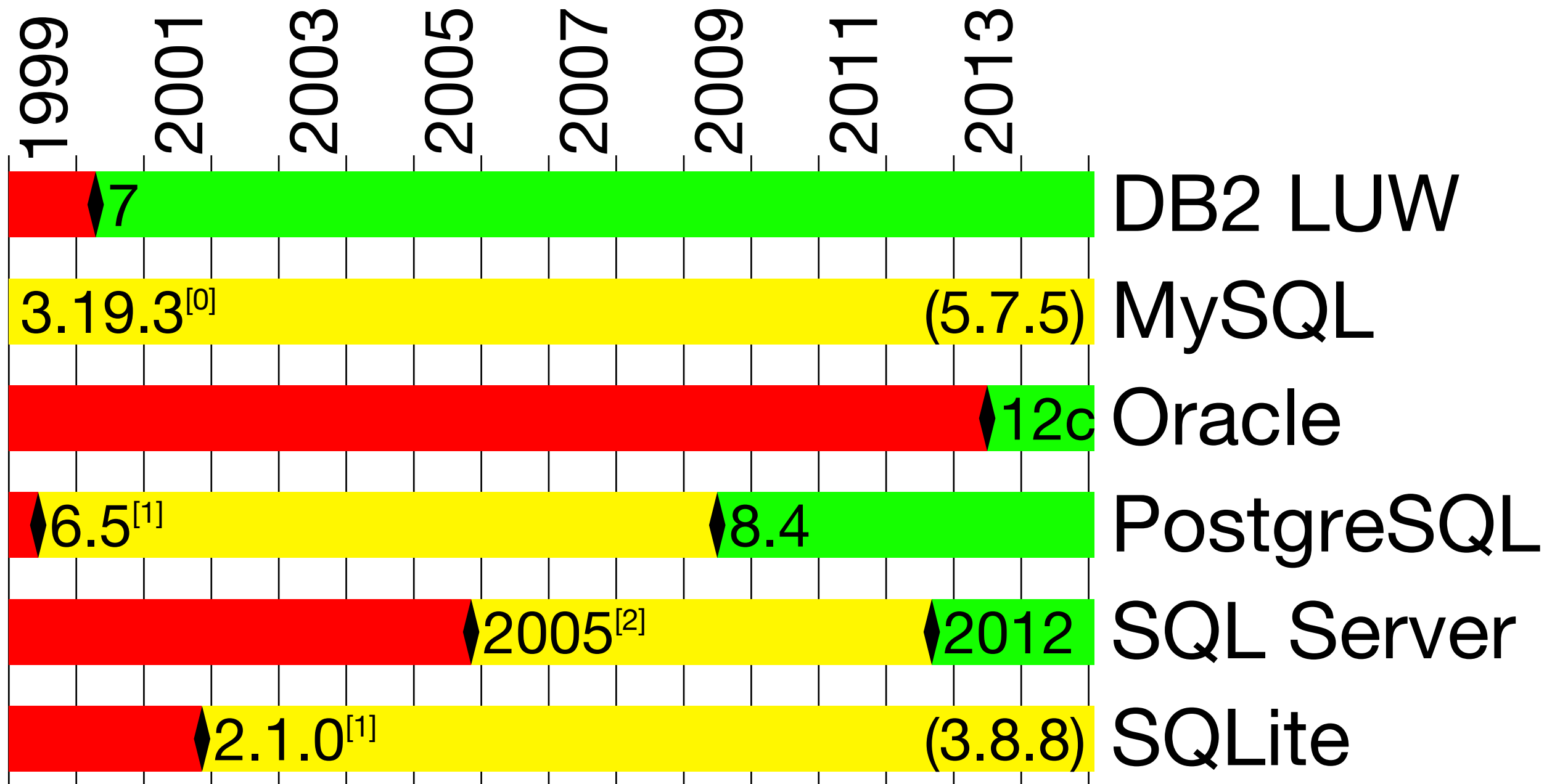
# FETCH FIRST Since SQL:2008

---

SQL:2008 has FETCH FIRST n ROWS ONLY:

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

# FETCH FIRST Availability



<sup>[0]</sup> Earliest mention of LIMIT. Probably inherited from mSQL

<sup>[1]</sup> Functionality available using LIMIT

<sup>[2]</sup> SELECT TOP n \* FROM...

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 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 is EVIL

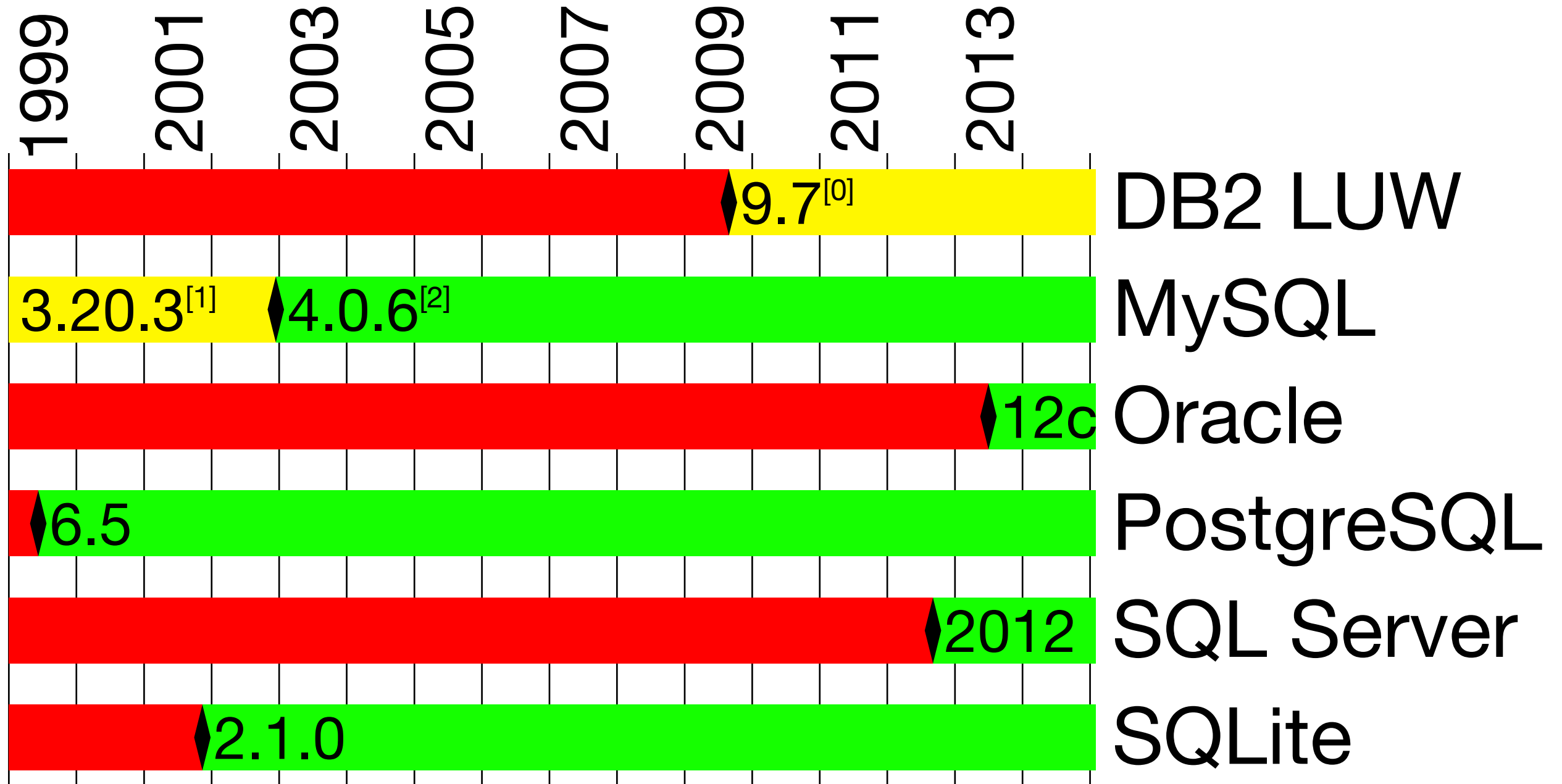
---

*Grab stickers  
and coasters!*



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

# OFFSET Availability (SQL:2011)



<sup>[0]</sup> Requires enabling the MySQL compatibility vector: `db2set DB2_COMPATIBILITY_VECTOR=MYS`

<sup>[1]</sup> `LIMIT [offset,] limit`: "With this it's easy to do a poor man's next page/previous page WWW application."

<sup>[2]</sup> The release notes say "Added PostgreSQL compatible LIMIT syntax"

**WITHOUT OVERLAPS**

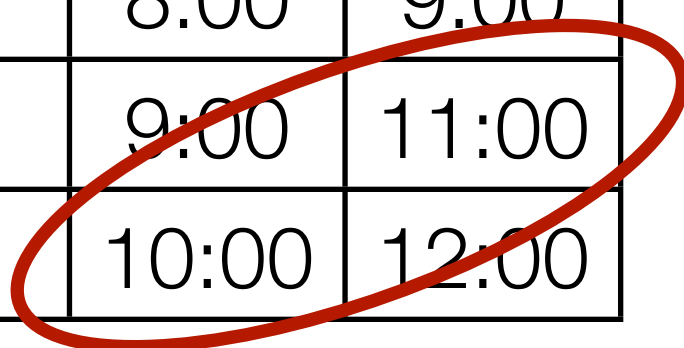
# WITHOUT OVERLAPS Before SQL:2011

---

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)**

# 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 far beyond **WITHOUT OVERLAPS**.

Please read these papers to get the idea:

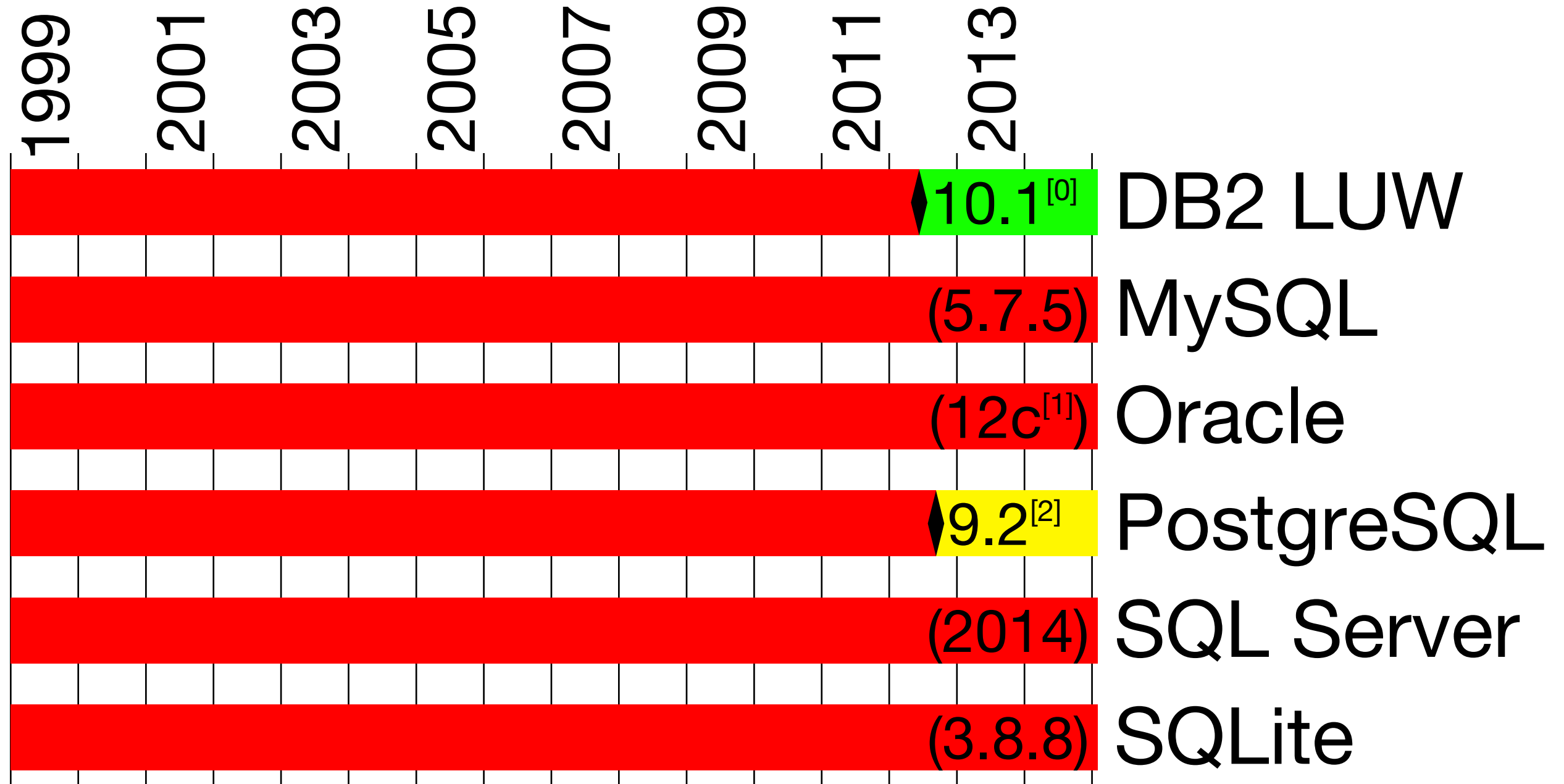
Temporal features in SQL:2011

[http://cs.ulb.ac.be/public/\\_media/teaching/infoh415/tempfeaturessql2011.pdf](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



<sup>[0]</sup> Minor differences: PERIOD without FOR; period name must be BUSINESS\_TIME

<sup>[1]</sup> Oracle 12c has partial temporal support, but no direct equivalent of WITHOUT OVERLAPS

<sup>[2]</sup> Functionality available using EXCLUDE constraints

# About Markus Winand

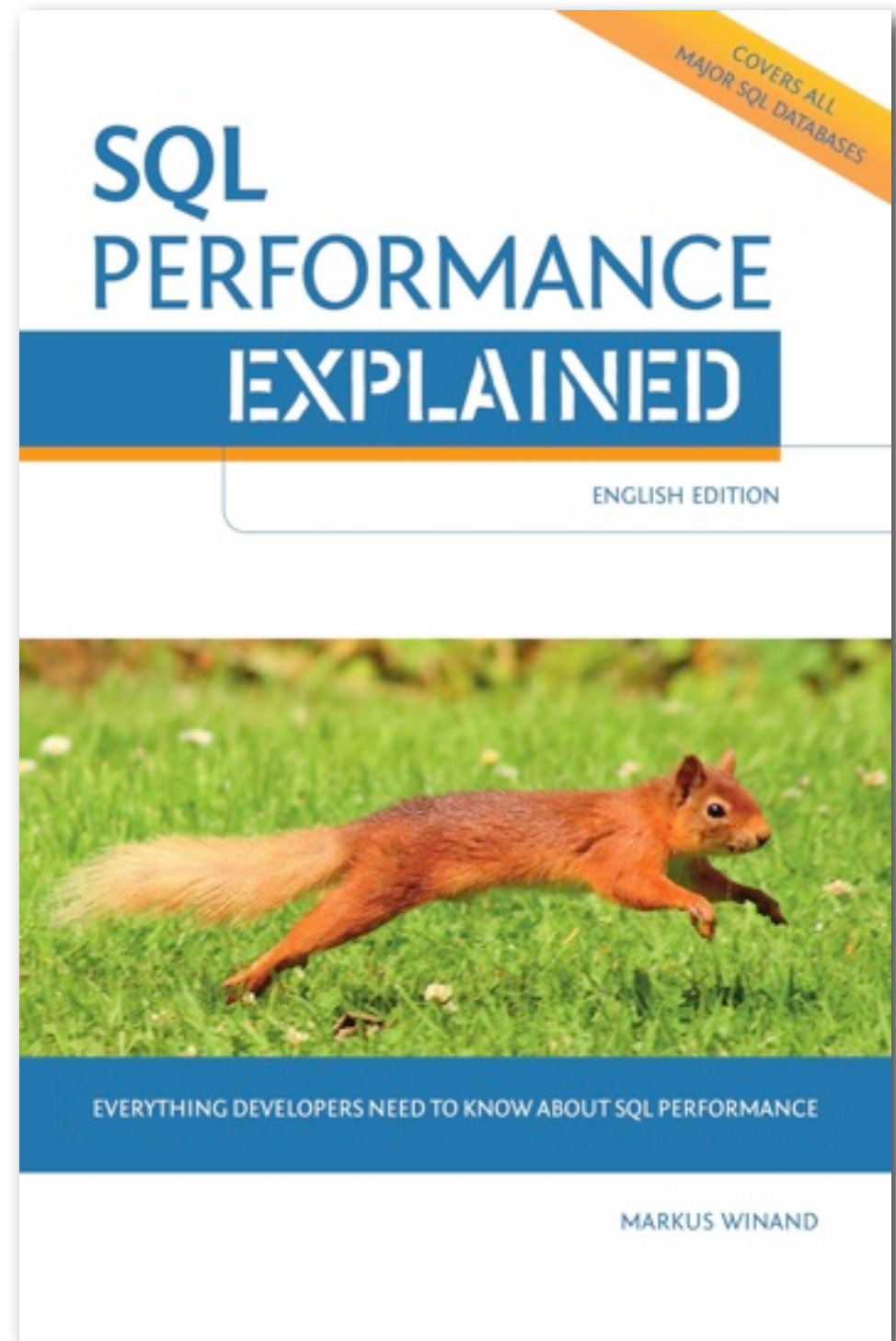
---

Tuning developers for  
high SQL performance

Training & tuning:  
[winand.at](http://winand.at)

Author of:  
SQL Performance Explained

Geeky blog:  
[use-the-index-luke.com](http://use-the-index-luke.com)



# About Markus Winand

---

[use-the-index-luke.com](https://use-the-index-luke.com)