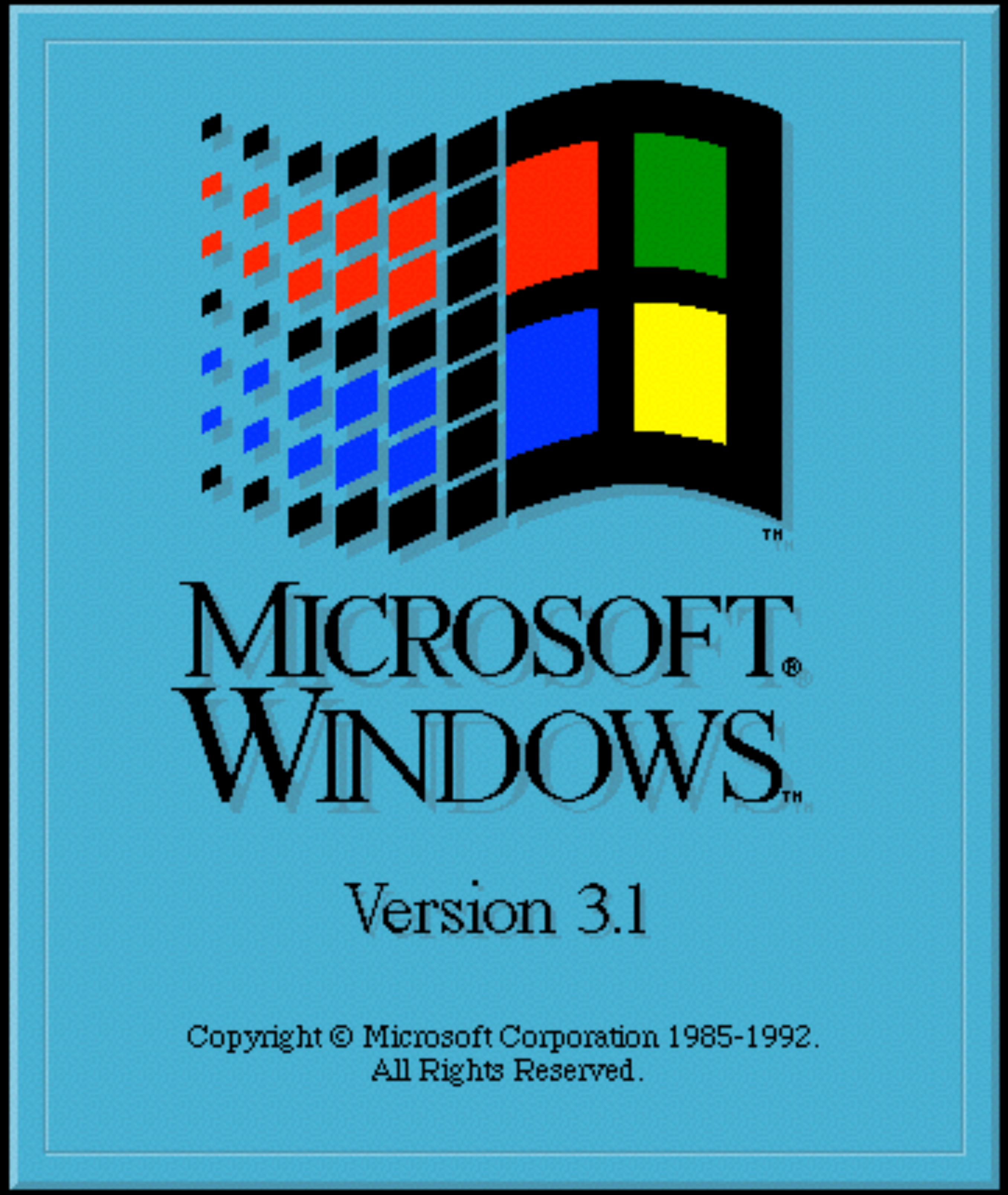
Still using
Windows 3.1?

So why stick to
SQL-92?



MICROSOFT® WINDOWS™

Version 3.1

Copyright © Microsoft Corporation 1985-1992.
All Rights Reserved.

@ModernSQL - http://modern-sql.com/
@MarkusWinand

SQL:1999

LATERAL

LATERAL

Before SQL:1999

Select-list sub-queries must be ***scalar***[0]:

(an atomic quantity that can hold only one value at a time[1])

```
SELECT …
     , (SELECT column_1
          FROM t1
         WHERE t1.x = t2.y
       ) AS c
  FROM t2
    …
```

[0] Neglecting row values and other workarounds here; [1] https://en.wikipedia.org/wiki/Scalar

LATERAL

Before SQL:1999

Select-list sub-queries must be *scalar*[0]:
(an atomic quantity that can hold only one value at a time[1])

```
SELECT …
     , (SELECT column_1, column_2
          FROM t1
         WHERE t1.x = t2.y
       ) AS c
  FROM t2
   …
```

More than one row? ⇒Runtime error!

More than one column? ⇒Syntax error

[0] Neglecting row values and other workarounds here; [1] https://en.wikipedia.org/wiki/Scalar

LATERAL Since SQL:1999

Lateral derived tables lift both limitations and can be correlated:

```
SELECT …
     , ldt.*
  FROM t2
  LEFT JOIN LATERAL (SELECT column_1, column_2
                       FROM t1
                      WHERE t1.x = t2.y
                    ) AS ldt
       ON (true)
    …
```

LATERAL

Since SQL:1999

Lateral derived tables lift both limitations and can be correlated:

```
SELECT …
     , ldt.*
  FROM t2
  LEFT JOIN LATERAL (SELECT column_1, column_2
                       FROM t1
                      WHERE t1.x = t2.y
                    ) AS ldt
       ON (true)
    …
```

"Derived table" means it's in the FROM/JOIN clause

Regular join semantics

Still "correlated"

LATERAL Use-Cases

- Top-N per group

  inside a lateral derived table `FETCH FIRST` (or `LIMIT`, `TOP`) applies per row from left tables.

  - Also useful to find most recent news from several subscribed topics (“multi-source top-N”).

```
FROM t
JOIN LATERAL (SELECT …
                FROM …
               WHERE t.c=…
               ORDER BY …
               LIMIT 10
```

http://use-the-index-luke.com/sql/partial-results/top-n-queries

LATERAL

Use-Cases

- Top-N per group

  inside a lateral derived table `FETCH FIRST` (or `LIMIT`, `TOP`) applies per row from left tables.

  - Also useful to find most recent news from several subscribed topics (“multi-source top-N”).

```
FROM t
JOIN LATERAL (SELECT …
                FROM …
               WHERE t.c=…
               ORDER BY …
               LIMIT 10
             ) derived_table
```

- Table function arguments

  (`TABLE` often implies `LATERAL`)

```
FROM t
JOIN TABLE (your_func(t.c))
```

LATERAL

In a Nutshell

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

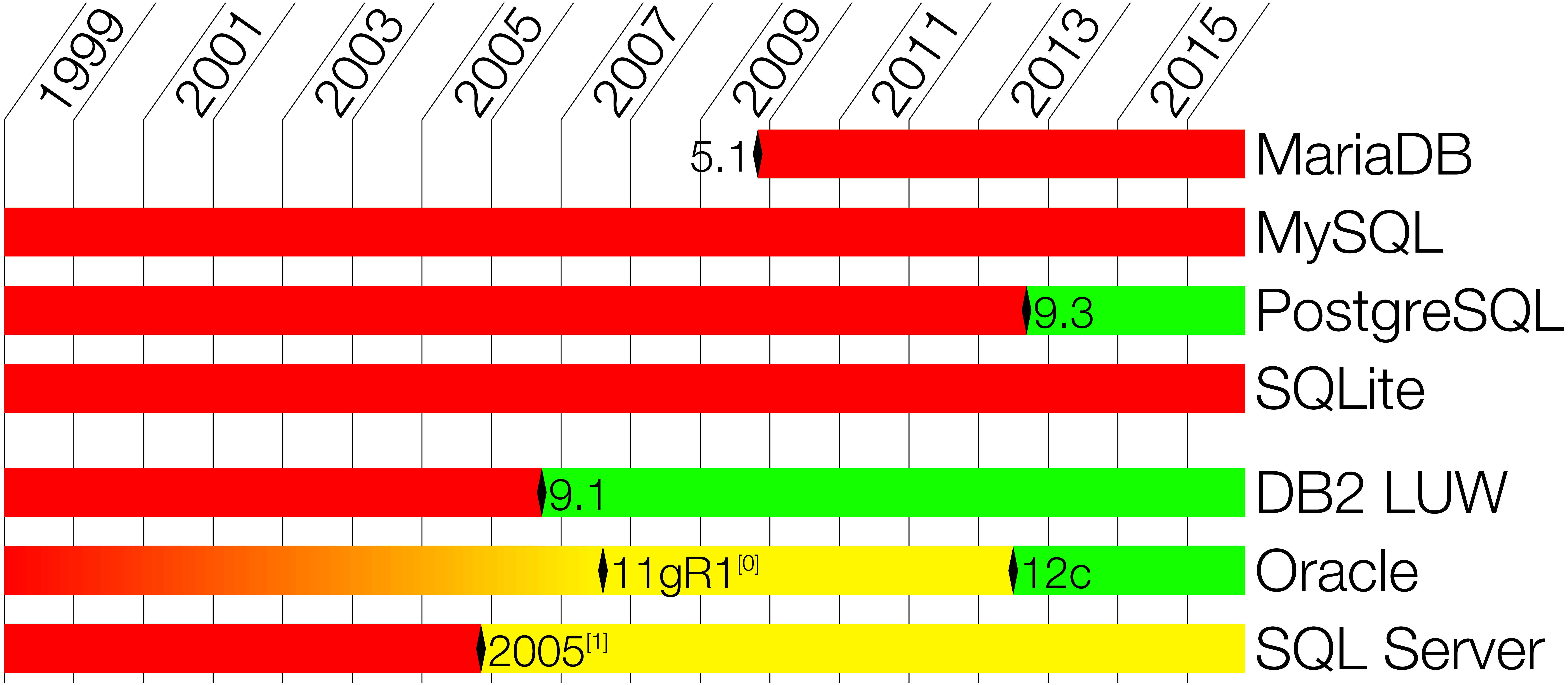
`LATERAL` plays well with outer and cross joins

`LATERAL` is great for Top-N subqueries

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

LATERAL

Availability



[0] Undocumented. Requires setting trace event 22829.

[1] `LATERAL` is not supported as of SQL Server 2016 but `[CROSS|OUTER] APPLY` can be used for the same effect.

GROUPING SETS

GROUPING SETS Before SQL:1999

Only one **GROUP BY** operation at a time:

Monthly revenue

```
SELECT year
     , month
     , sum(revenue)
  FROM tbl
 GROUP BY year, month
```

Yearly revenue

```
SELECT year

     , sum(revenue)
  FROM tbl
 GROUP BY year
```

GROUPING SETS Before SQL:1999

```
SELECT year
     , month
     , sum(revenue)
  FROM tbl
 GROUP BY year, month

SELECT year

     , sum(revenue)
  FROM tbl
 GROUP BY year
```

GROUPING SETS Before SQL:1999

```
SELECT year
     , month
     , sum(revenue)
  FROM tbl
 GROUP BY year, month
 UNION ALL
SELECT year
     , null
     , sum(revenue)
  FROM tbl
 GROUP BY year
```

GROUPING SETS

Since SQL:1999

```
SELECT year
     , month
     , sum(revenue)
  FROM tbl
 GROUP BY year, month
 UNION ALL
SELECT year
     , null
     , sum(revenue)
  FROM tbl
 GROUP BY year
```

```
SELECT year
     , month
     , sum(revenue)
  FROM tbl
 GROUP BY
       GROUPING SETS (
          (year, month)
        , (year)
       )
```

`GROUPING SETS` In a Nutshell

`GROUPING SETS` are multiple `GROUP BY`s in one go

`()` (empty brackets) build a group over all rows

`GROUPING` (function) disambiguates the meaning of `NULL`

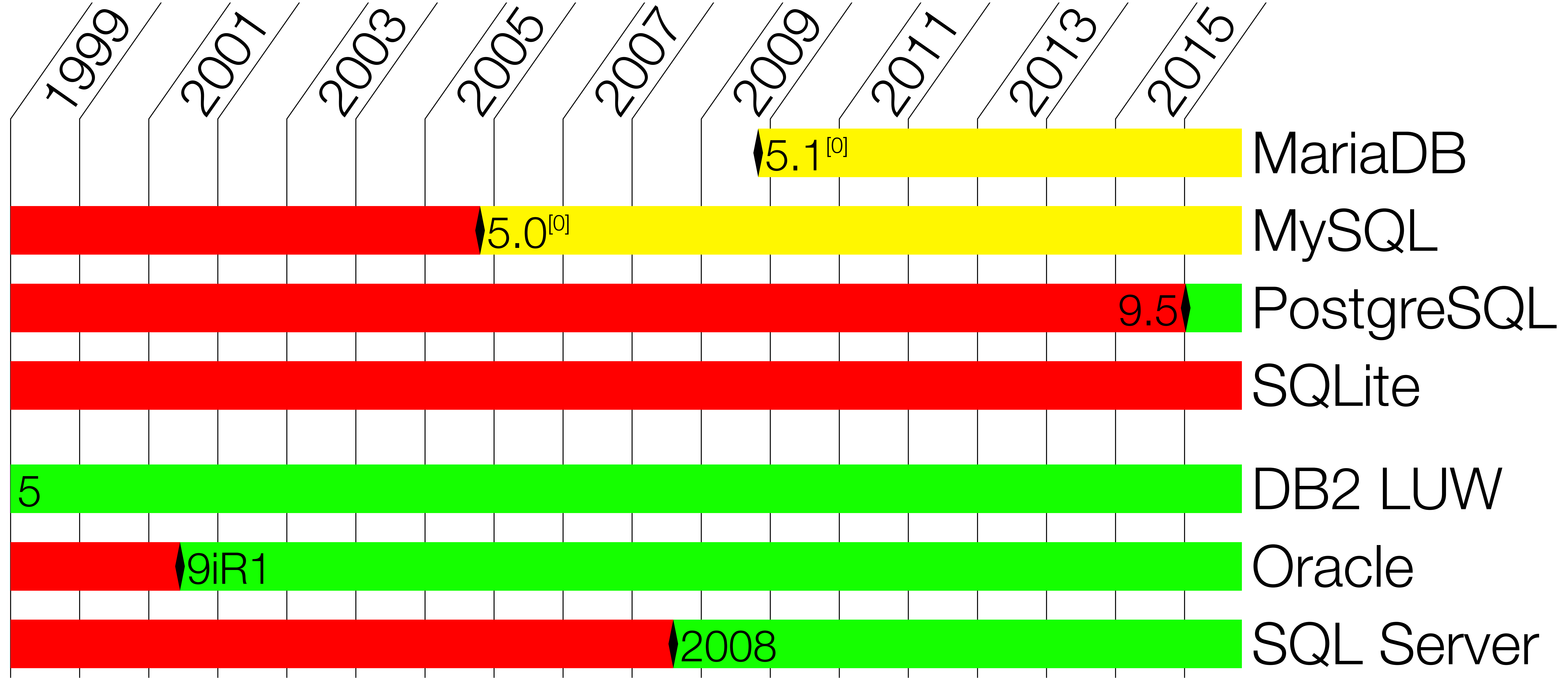
(was the grouped data `NULL` or is this column not currently grouped?)

Permutations can be created using `ROLLUP` and `CUBE`

(`ROLLUP(a,b,c)` = `GROUPING SETS ((a,b,c), (a,b),(a),())`

GROUPING SETS

Availability



[0]Only ROLLUP

`WITH`

(Common Table Expressions)

WITH (non-recursive) The Problem

Nested queries are hard to read:

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

WITH (non-recursive)

The Problem

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 (non-recursive) The Problem

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 (non-recursive)

The Problem

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 (non-recursive) The Problem

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 (non-recursive) Since SQL:1999

CTEs are statement-scoped views:

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

WITH (non-recursive) Since SQL:1999

CTEs are statement-scoped views:

Keyword

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

WITH (non-recursive) Since SQL:1999

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 (non-recursive)

Since SQL:1999

CTEs are statement-scoped views:

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

Definition

WITH (non-recursive)

Since SQL:1999

CTEs are statement-scoped views:

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

Introduces another CTE

Don't repeat WITH

WITH (non-recursive)

Since SQL:1999

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 (…)
   ),

 c (…)
```

May refer to previous CTEs

WITH (non-recursive)

Since SQL:1999

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

Third CTE

WITH (non-recursive) Since SQL:1999

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

 c (…)
AS (SELECT … FROM …)

SELECT …
  FROM b JOIN c ON (…)
```

No comma!

WITH (non-recursive)

Since SQL:1999

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

 c (…)
AS (SELECT … FROM …)
SELECT …
  FROM b JOIN c ON (…)
```

Main query

WITH (non-recursive)

Since SQL:1999

```
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` (non-recursive) Use-Cases

- Literate SQL

  http://modern-sql.com/use-case/literate-sql

  Organize SQL code to improve maintainability

- Assign column names

  http://modern-sql.com/use-case/naming-unnamed-columns

  to tables produced by `values` or `unnest`.

- Overload tables (for testing)

  http://modern-sql.com/use-case/unit-tests-on-transient-data

  `with` queries hide tables of the same name.

`WITH` (non-recursive) In a Nutshell

`WITH` are the "private methods" of SQL

`WITH` is a prefix to `SELECT`

`WITH` queries are only visible in the `SELECT` they precede

`WITH` in detail:

http://modern-sql.com/feature/with

WITH (non-recursive) PostgreSQL “issues”

In PostgreSQL **WITH** queries are “optimizer fences”:

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

WITH (non-recursive) PostgreSQL “issues”

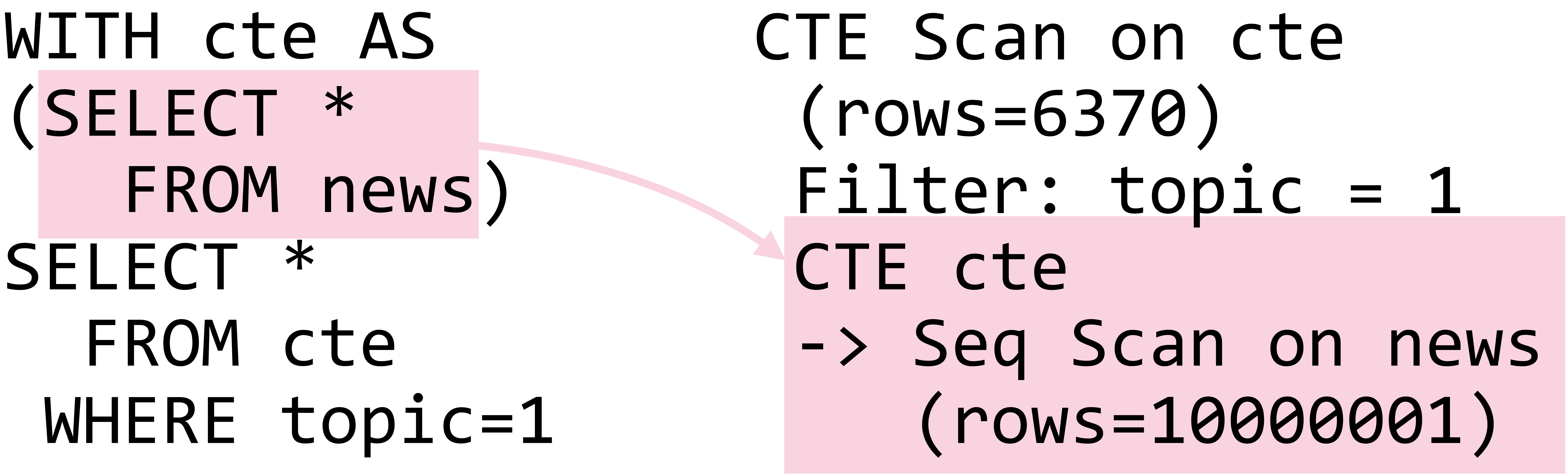
In PostgreSQL **WITH** queries are “optimizer fences”:

```
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` (non-recursive) PostgreSQL “issues”

In PostgreSQL **`WITH`** queries are “optimizer fences”:



```
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 (non-recursive) PostgreSQL “issues”

In PostgreSQL **WITH** queries are “optimizer fences”:

```
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)
```

CTE doesn't know about the outer filter

WITH (non-recursive) PostgreSQL “issues”

Views and derived tables support "predicate pushdown":

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

WITH (non-recursive) PostgreSQL “issues”

Views and derived tables support "predicate pushdown":

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

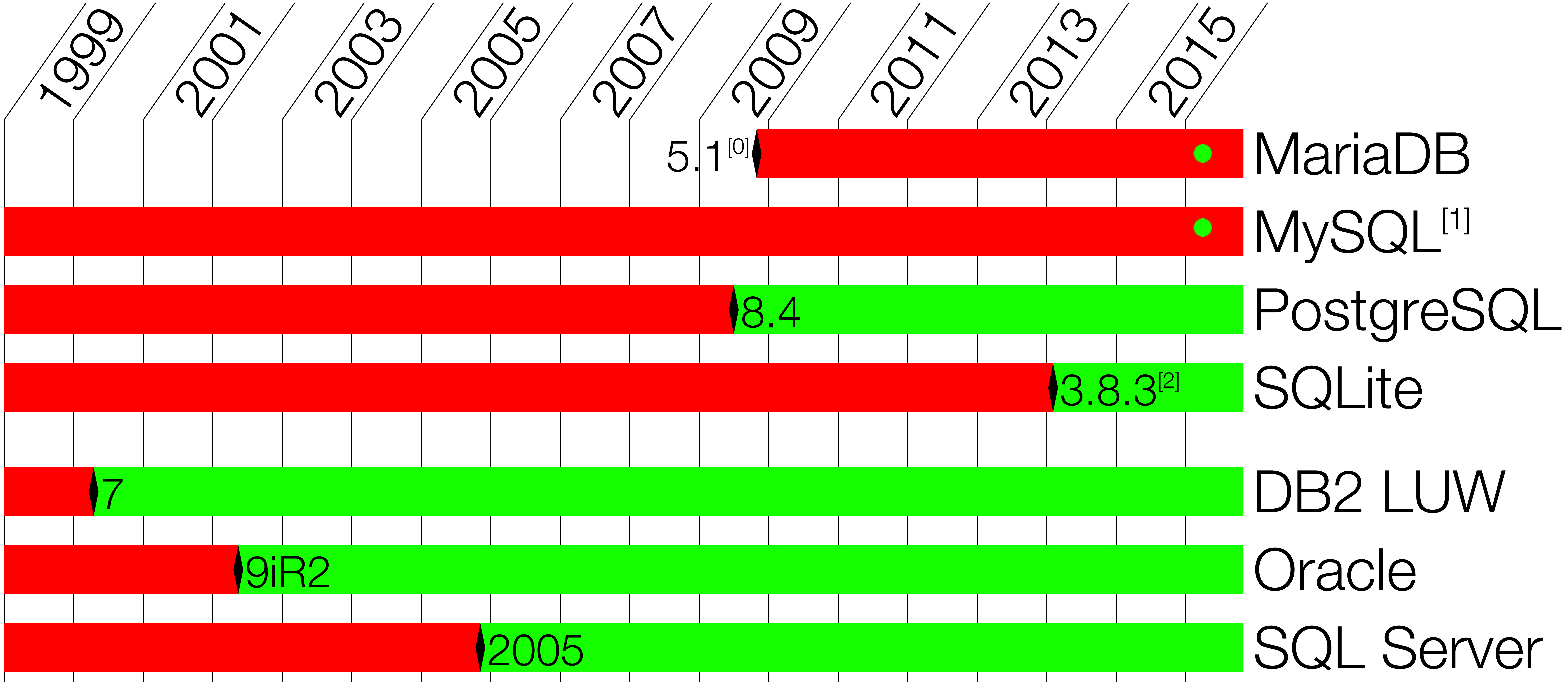
```
Bitmap Heap Scan
on news (rows=6370)
->Bitmap Index Scan
  on idx (rows=6370)
  Cond: topic=1
```

WITH (non-recursive) PostgreSQL Extension

PostgreSQL 9.1+ allows DML within **WITH**:

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

WITH (non-recursive) Availability



[0] Available MariaDB 10.2 alpha

[1] Announced for 8.0: http://www.percona.com/blog/2016/09/01/percona-live-europe-featured-talk-manyi-lu

[2] Only for top-level `SELECT` statements

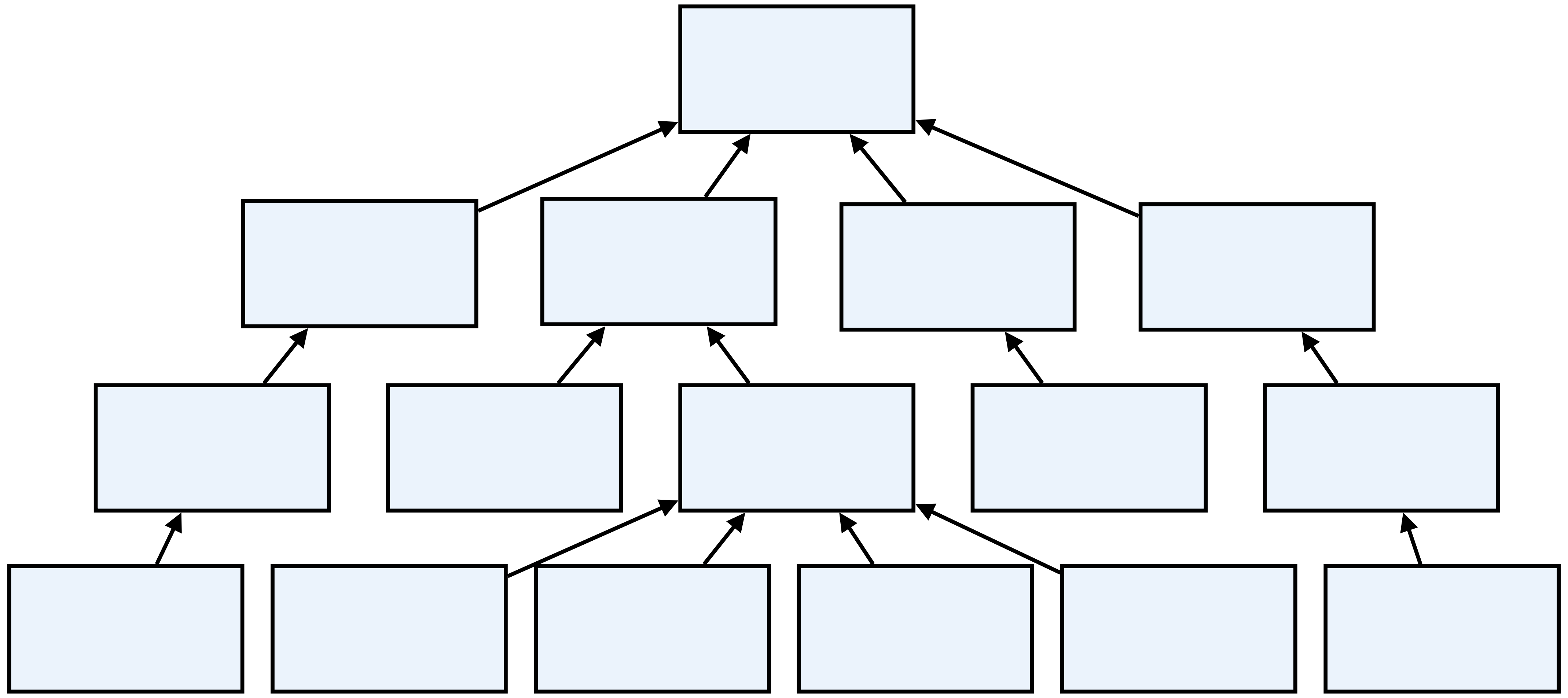
WITH RECURSIVE

(Common Table Expressions)

WITH RECURSIVE The Problem

Coping with hierarchies in the Adjacency List Model[0]

```
CREATE TABLE t (
   id NUMERIC NOT NULL,
   parent_id NUMERIC,
   …
   PRIMARY KEY (id)
)
```



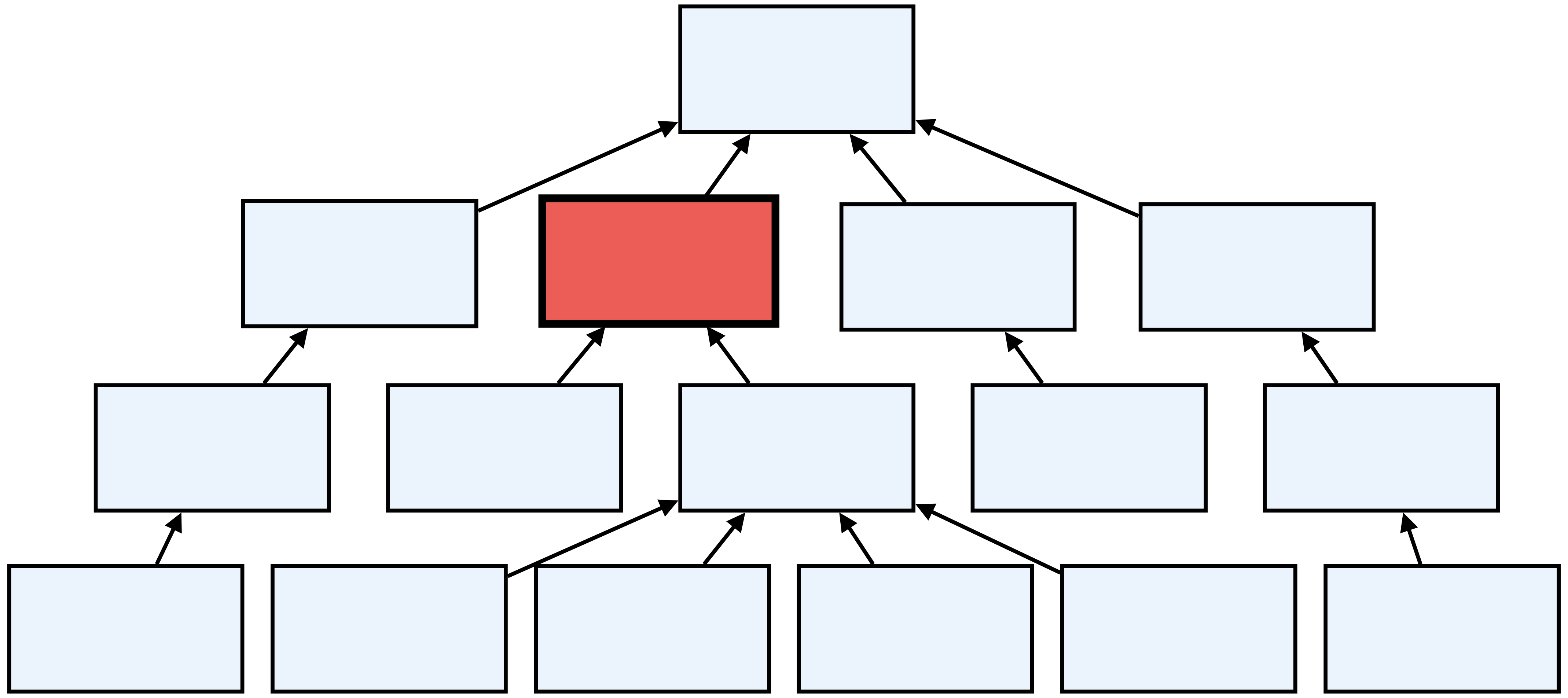
[0] Hierarchies implemented using a “parent id” — see “Joe Celko’s Trees and Hierarchies in SQL for Smarties”

WITH RECURSIVE

The Problem

Coping with hierarchies in the Adjacency List Model[0]

```
SELECT *
  FROM t AS d0
 WHERE d0.id = ?
```



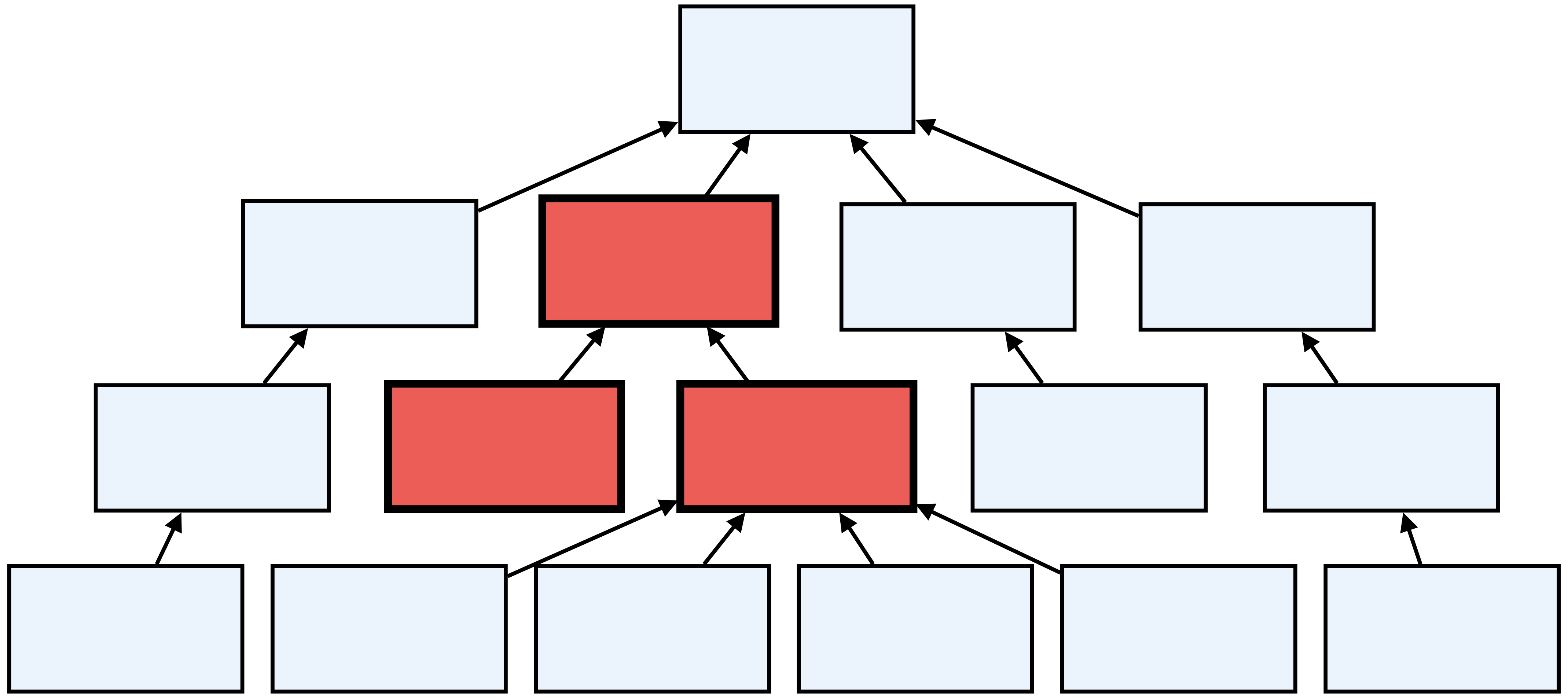
[0] Hierarchies implemented using a “parent id” — see “Joe Celko’s Trees and Hierarchies in SQL for Smarties”

WITH RECURSIVE

The Problem

Coping with hierarchies in the Adjacency List Model[0]

```
SELECT *
  FROM t AS d0
  LEFT JOIN t AS d1
    ON (d1.parent_id=d0.id)
 WHERE d0.id = ?
```



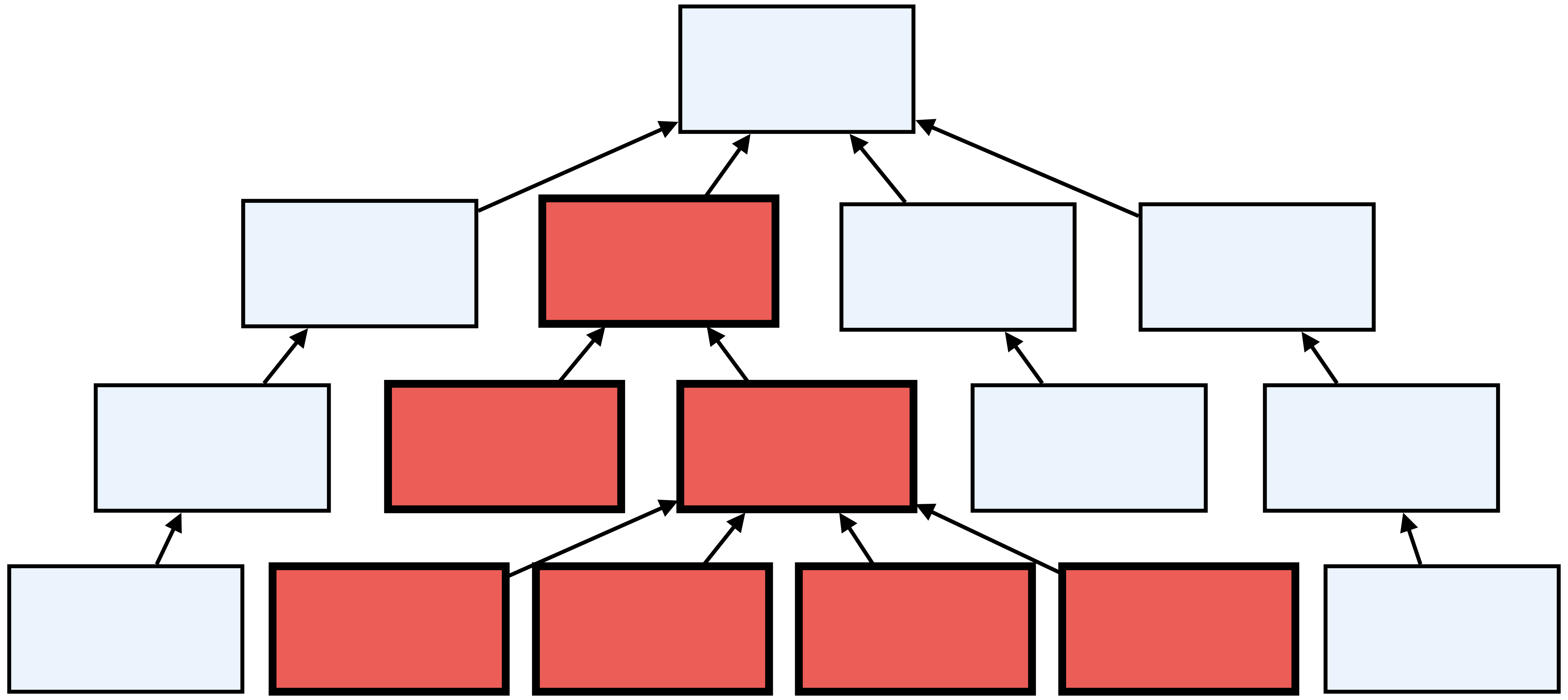
[0] Hierarchies implemented using a “parent id” — see “Joe Celko’s Trees and Hierarchies in SQL for Smarties”

WITH RECURSIVE

The Problem

Coping with hierarchies in the Adjacency List Model[0]

```
SELECT *
   FROM t AS d0
   LEFT JOIN t AS d1
      ON (d1.parent_id=d0.id)
   LEFT JOIN t AS d2
      ON (d2.parent_id=d1.id)
 WHERE d0.id = ?
```



[0] Hierarchies implemented using a “parent id” — see “Joe Celko’s Trees and Hierarchies in SQL for Smarties”

WITH RECURSIVE

Since SQL:1999

```
SELECT *
  FROM t AS d0
  LEFT JOIN t AS d1
    ON (d1.parent_id=d0.id)
  LEFT JOIN t AS d2
    ON (d2.parent_id=d1.id)
 WHERE d0.id = ?
```

```
WITH RECURSIVE
   d (id, parent, …) AS
     (SELECT id, parent, …
        FROM tbl
       WHERE id = ?
   UNION ALL
      SELECT id, parent, …
        FROM d
        LEFT JOIN tbl
          ON (tbl.parent=d.id)
     )
SELECT *
  FROM subtree
```

WITH RECURSIVE Since SQL:1999

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:1999

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:1999

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:1999

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:1999

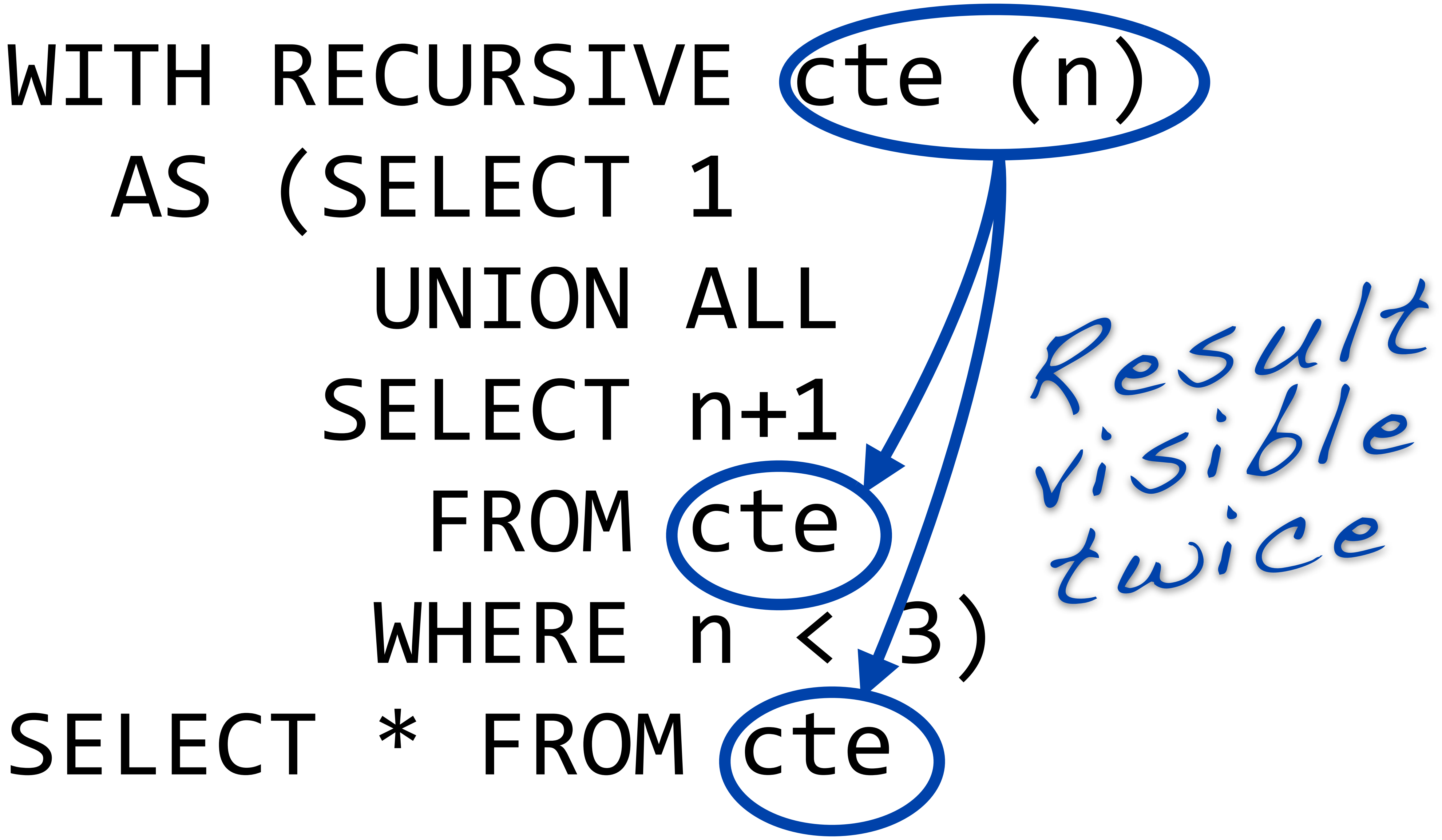
Recursive common table expressions may refer to themselves in the second leg of a **UNION [ALL]**:

```
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:1999

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

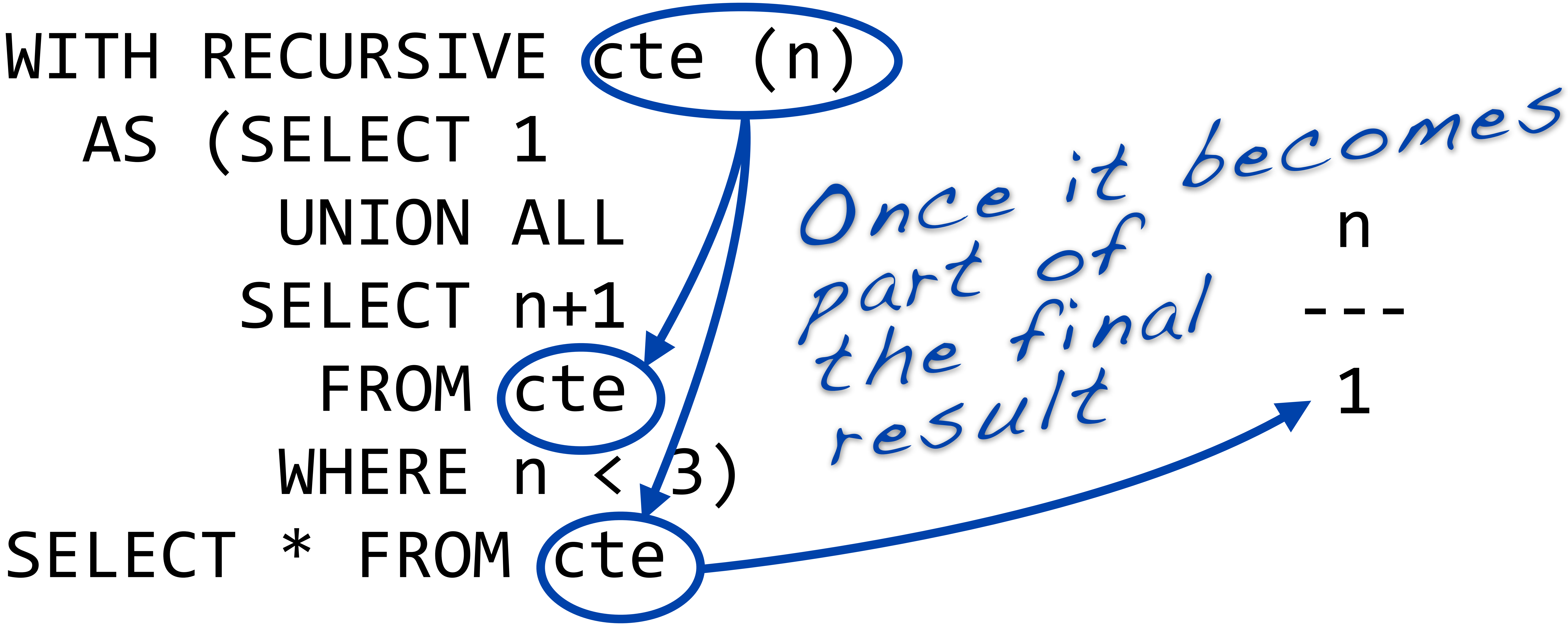


```
WITH RECURSIVE cte (n)
   AS (SELECT 1
        UNION ALL
       SELECT n+1
         FROM cte
        WHERE n < 3)
SELECT * FROM cte
```

WITH RECURSIVE

Since SQL:1999

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



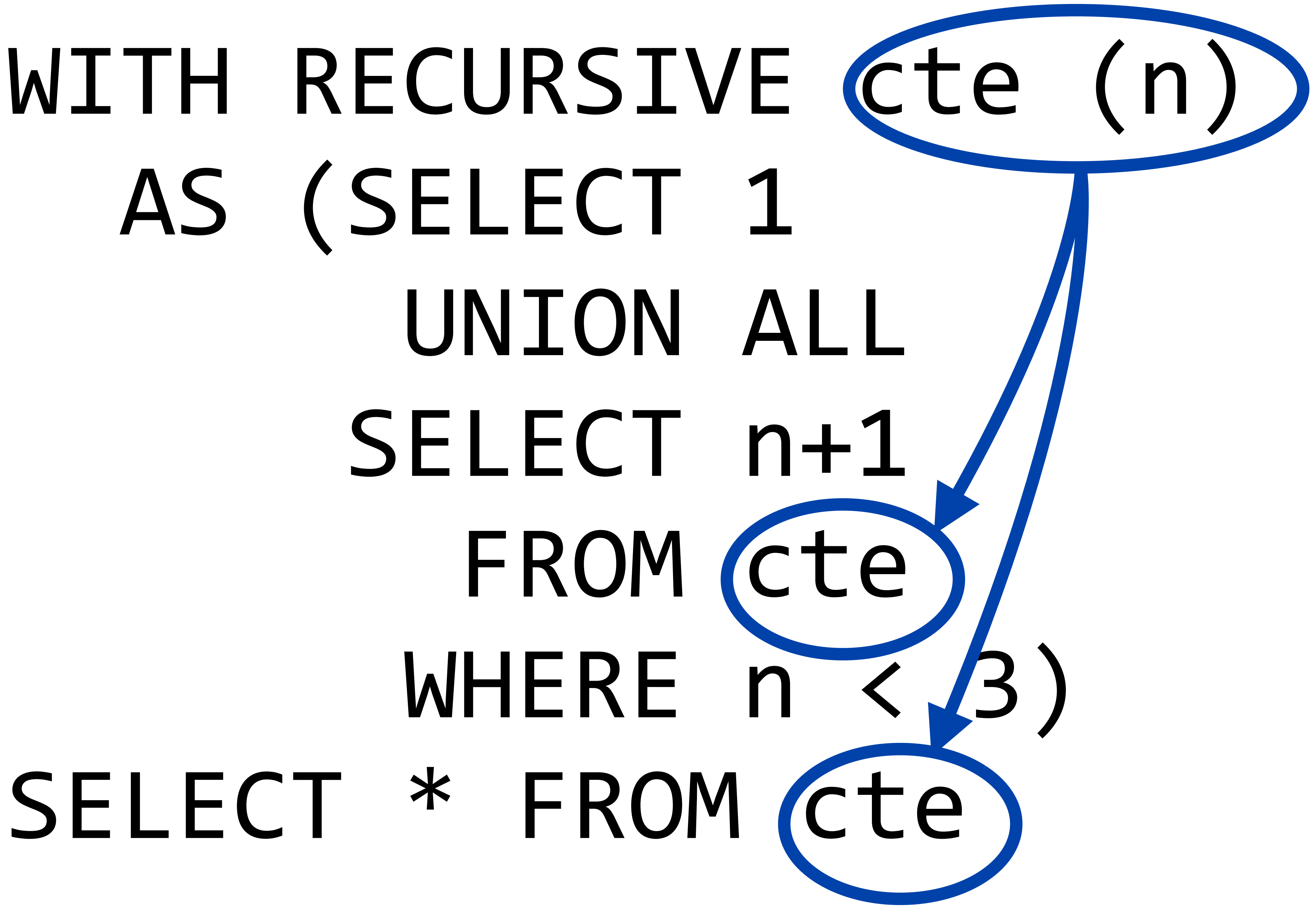
```
WITH RECURSIVE cte (n)
   AS (SELECT 1
         UNION ALL
        SELECT n+1
          FROM cte
         WHERE n < 3)
SELECT * FROM cte
```

Once it becomes part of the final result

```
 n
---
 1
```

WITH RECURSIVE Since SQL:1999

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:1999

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:1999

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
```

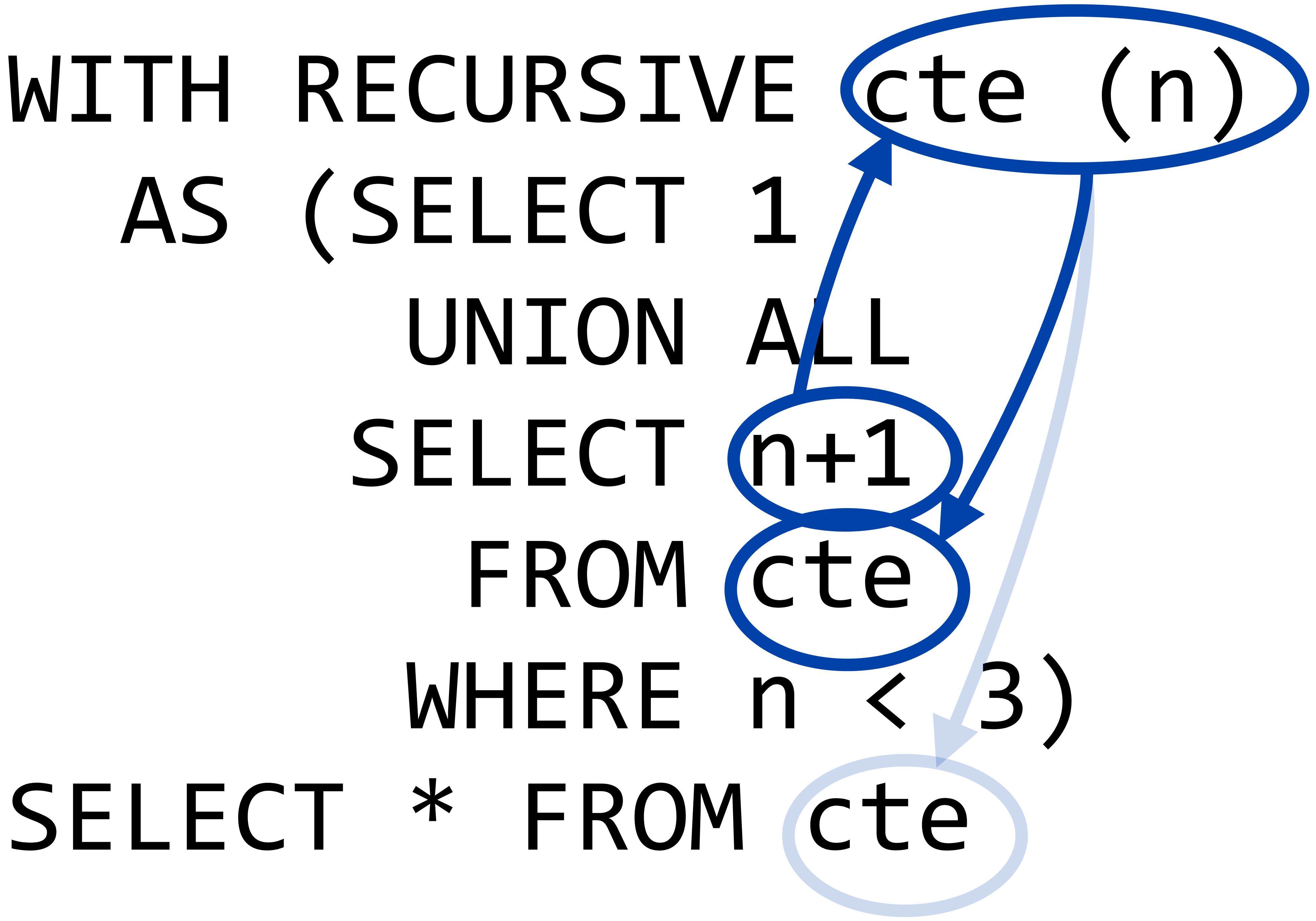
Result
sent there
again

```
 n
---
 1
```

WITH RECURSIVE

Since SQL:1999

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



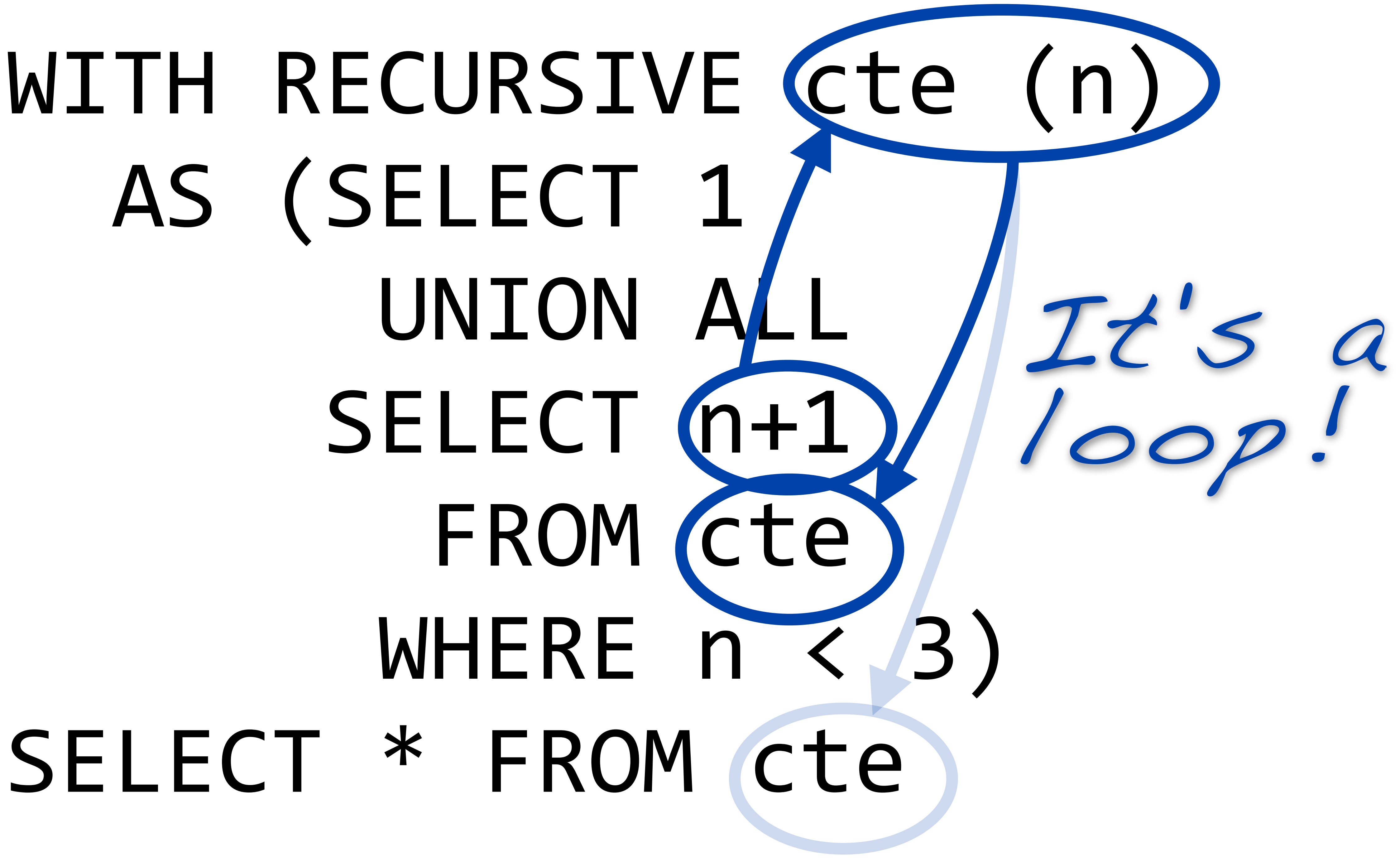
```
WITH RECURSIVE cte (n)
  AS (SELECT 1
       UNION ALL
      SELECT n+1
        FROM cte
       WHERE n < 3)
SELECT * FROM cte
```

```
 n
---
 1
```

WITH RECURSIVE

Since SQL:1999

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
```

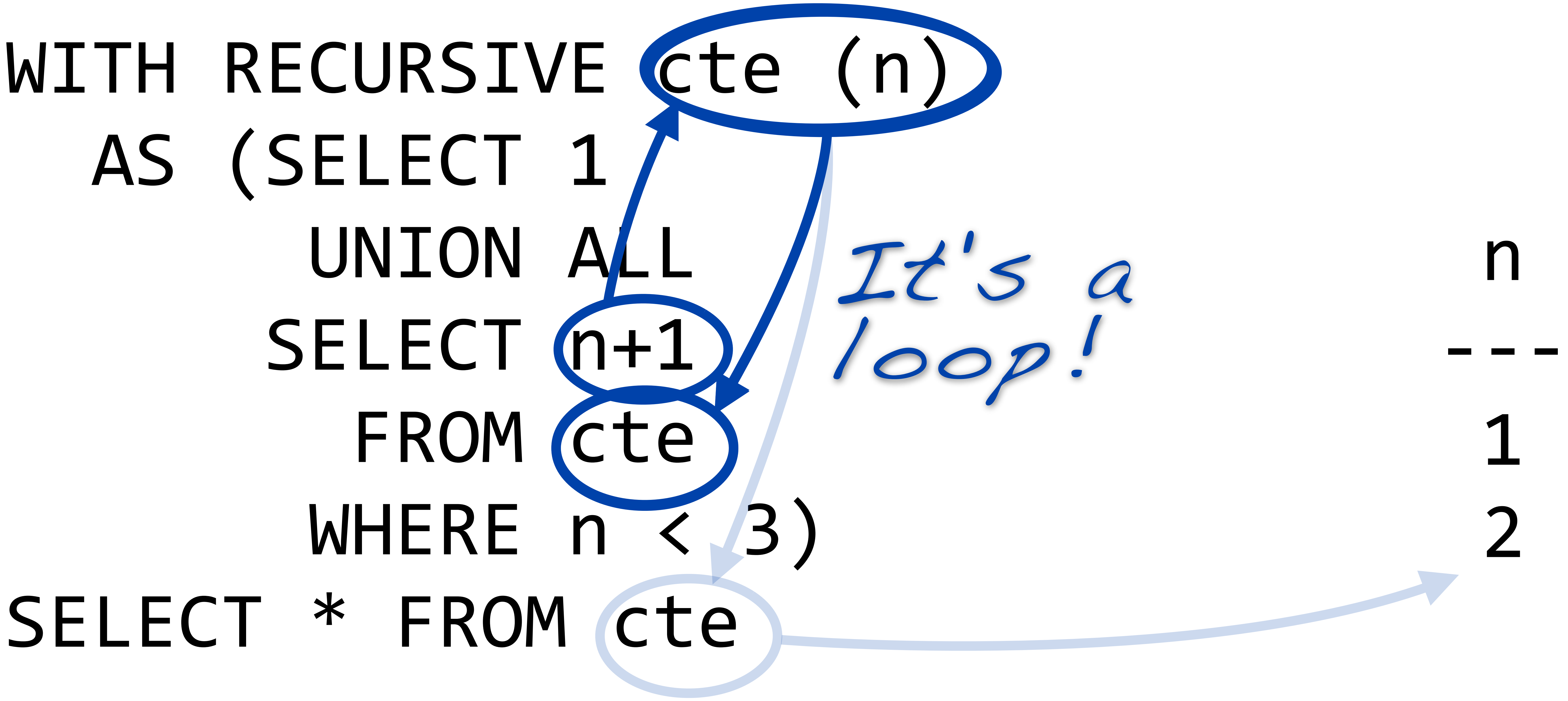
It's a loop!

```
 n
---
 1
```

WITH RECURSIVE

Since SQL:1999

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



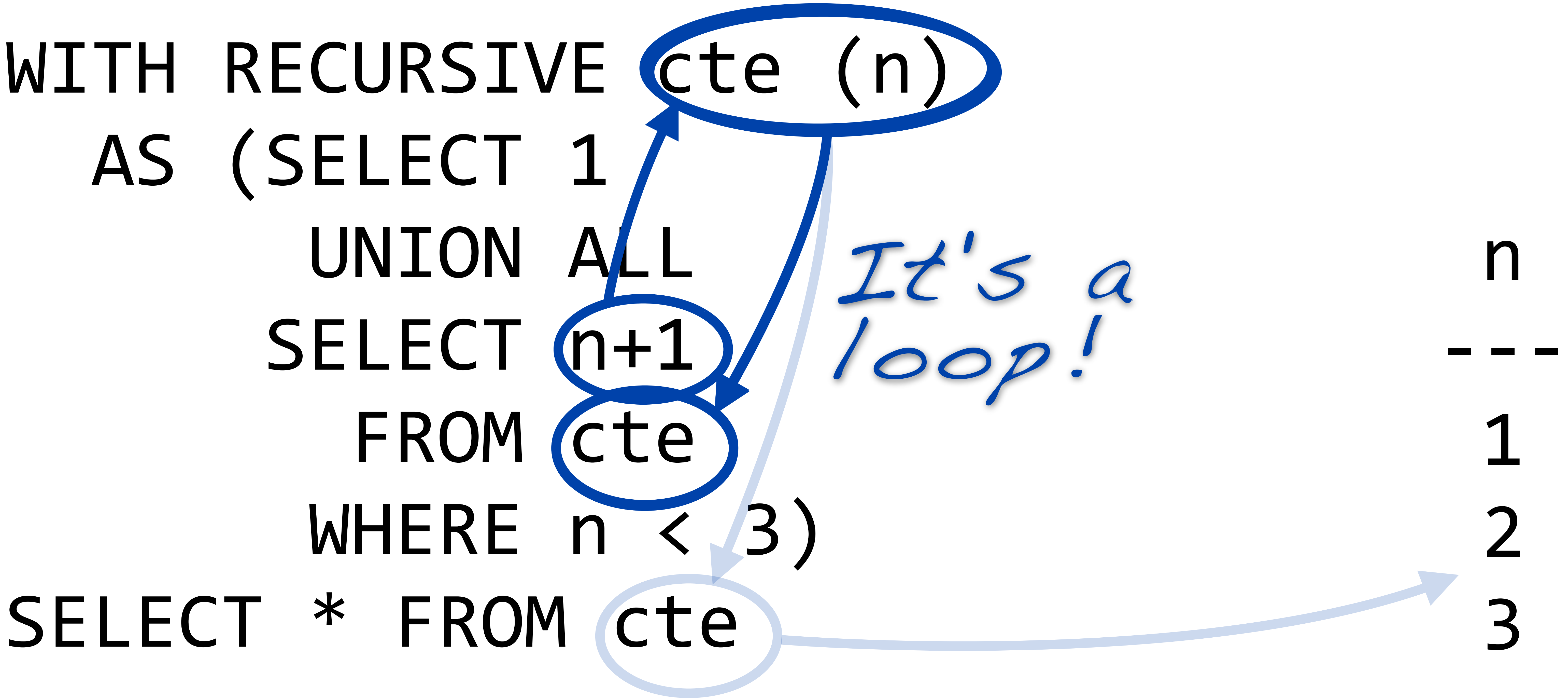
```
WITH RECURSIVE cte (n)
   AS (SELECT 1
         UNION ALL
        SELECT n+1
          FROM cte
         WHERE n < 3)
SELECT * FROM cte
```

It's a loop!

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

WITH RECURSIVE Since SQL:1999

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



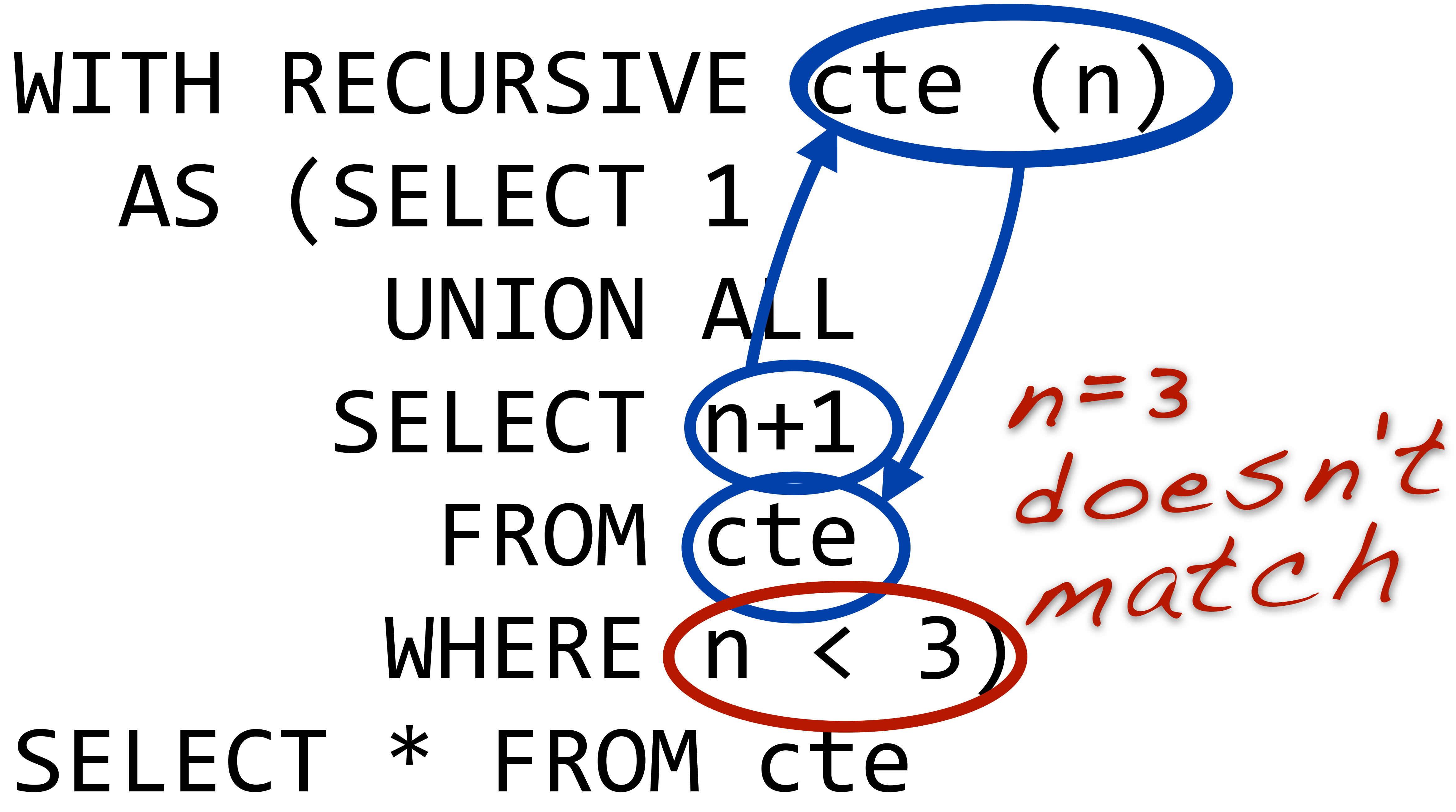
```
WITH RECURSIVE cte (n)
   AS (SELECT 1
        UNION ALL
       SELECT n+1
         FROM cte
        WHERE n < 3)
SELECT * FROM cte
```

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

WITH RECURSIVE

Since SQL:1999

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:1999

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

  As shown on previous slide

  To fill gaps (e.g., in time series), generate test data.

- Processing graphs

  http://aprogrammerwrites.eu/?p=1391

  Shortest route from person A to B in LinkedIn/Facebook/Twitter/…

  "[…] for certain classes of graphs, solutions utilizing relational database technology […] can offer performance superior to that of the dedicated graph databases." event.cwi.nl/grades2013/07-welc.pdf

- Finding distinct values

  http://wiki.postgresql.org/wiki/Loose_indexscan

  with *n*log(N)*† time complexity.

[…many more…]

† n … # distinct values, N … # of table rows. Suitable index required

`WITH RECURSIVE` In a Nutshell

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

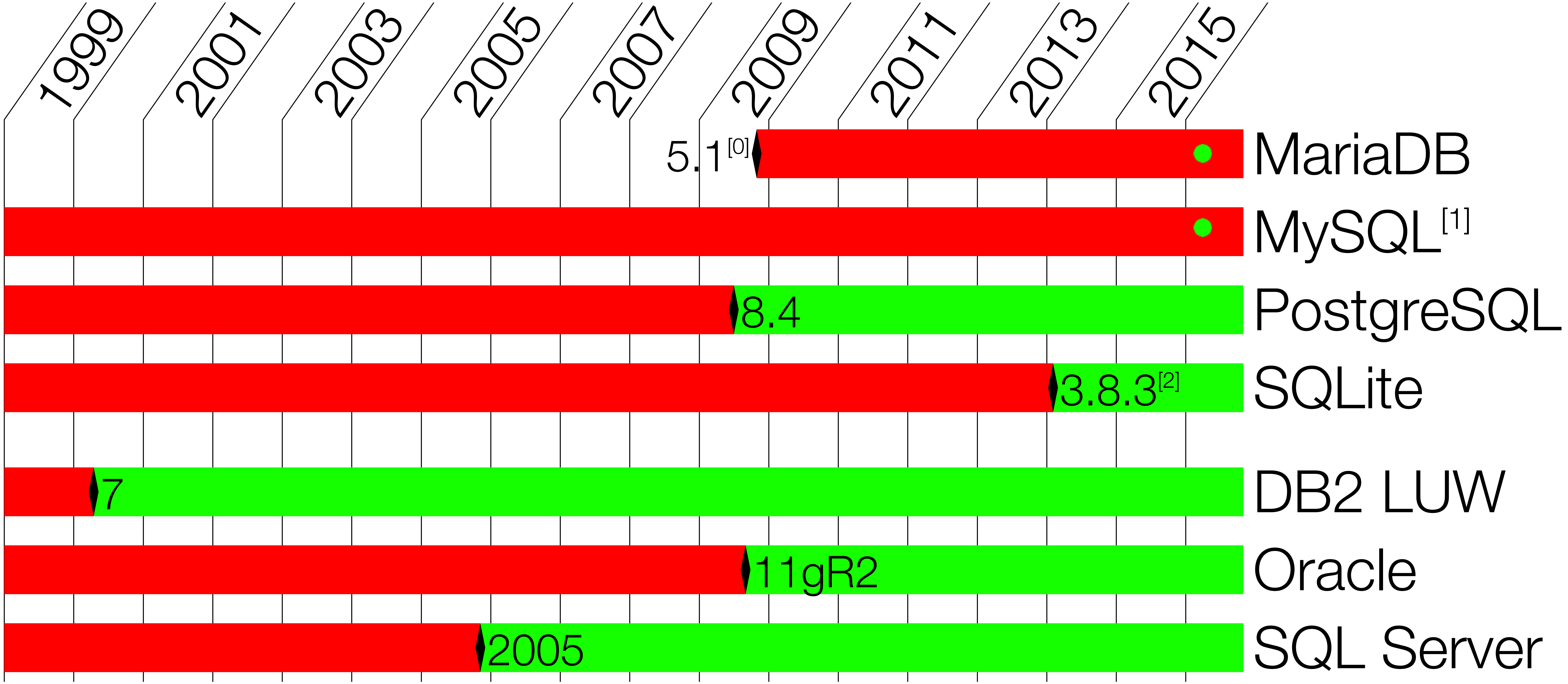
`WITH RECURSIVE` "supports" infinite loops

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

DB2, SQL Server & Oracle don’t even know the keyword **`RECURSIVE`**, but allow recursive CTEs anyway.

WITH RECURSIVE

Availability



[0] Expected in 10.2.2

[1] Announced for 8.0: http://www.percona.com/blog/2016/09/01/percona-live-europe-featured-talk-manyi-lu

[2] Only for top-level `SELECT` statements

SQL:2003

FILTER

FILTER

The Problem

Pivot table: Years on the Y axis, month on X:

```
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

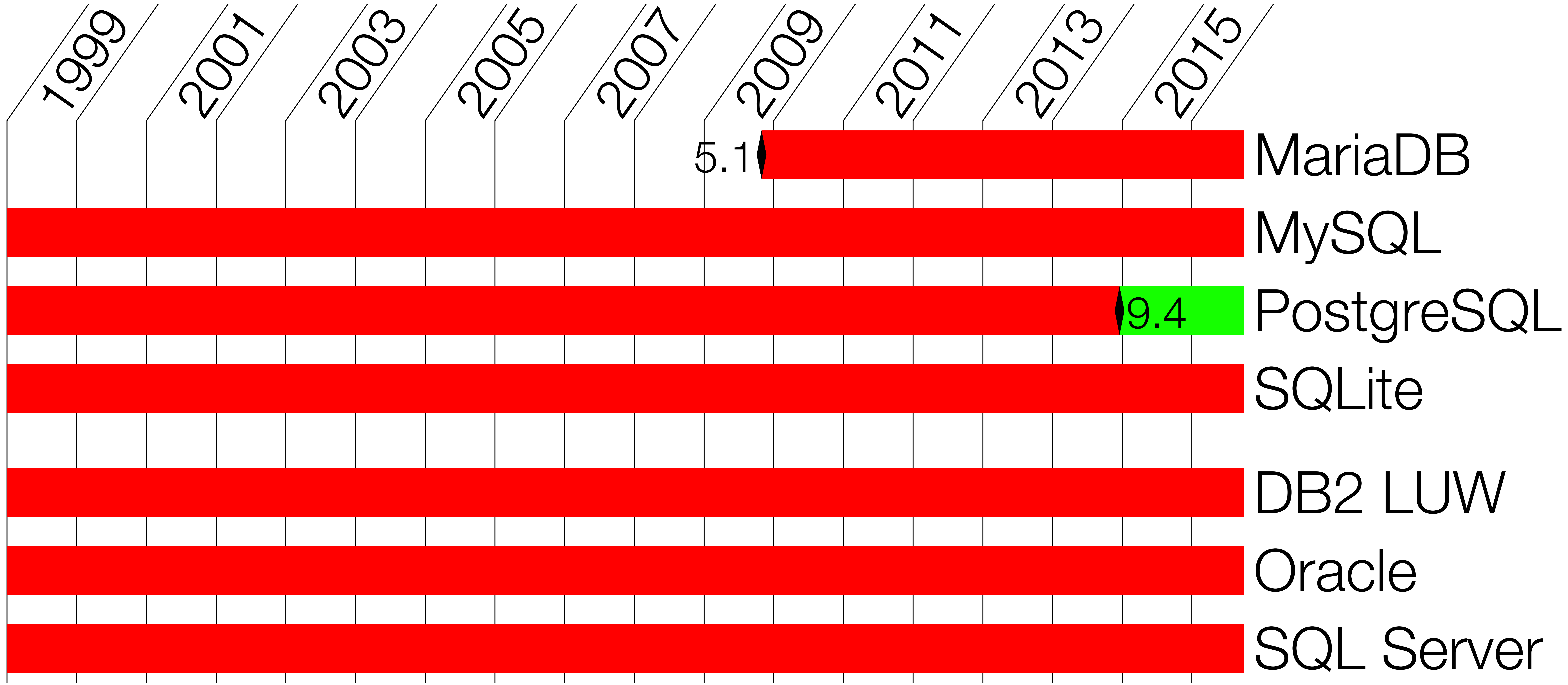
Since SQL:2003

SQL:2003 allows `FILTER (WHERE…)` after aggregates:

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

FILTER

Availability



OVER

and

PARTITION BY

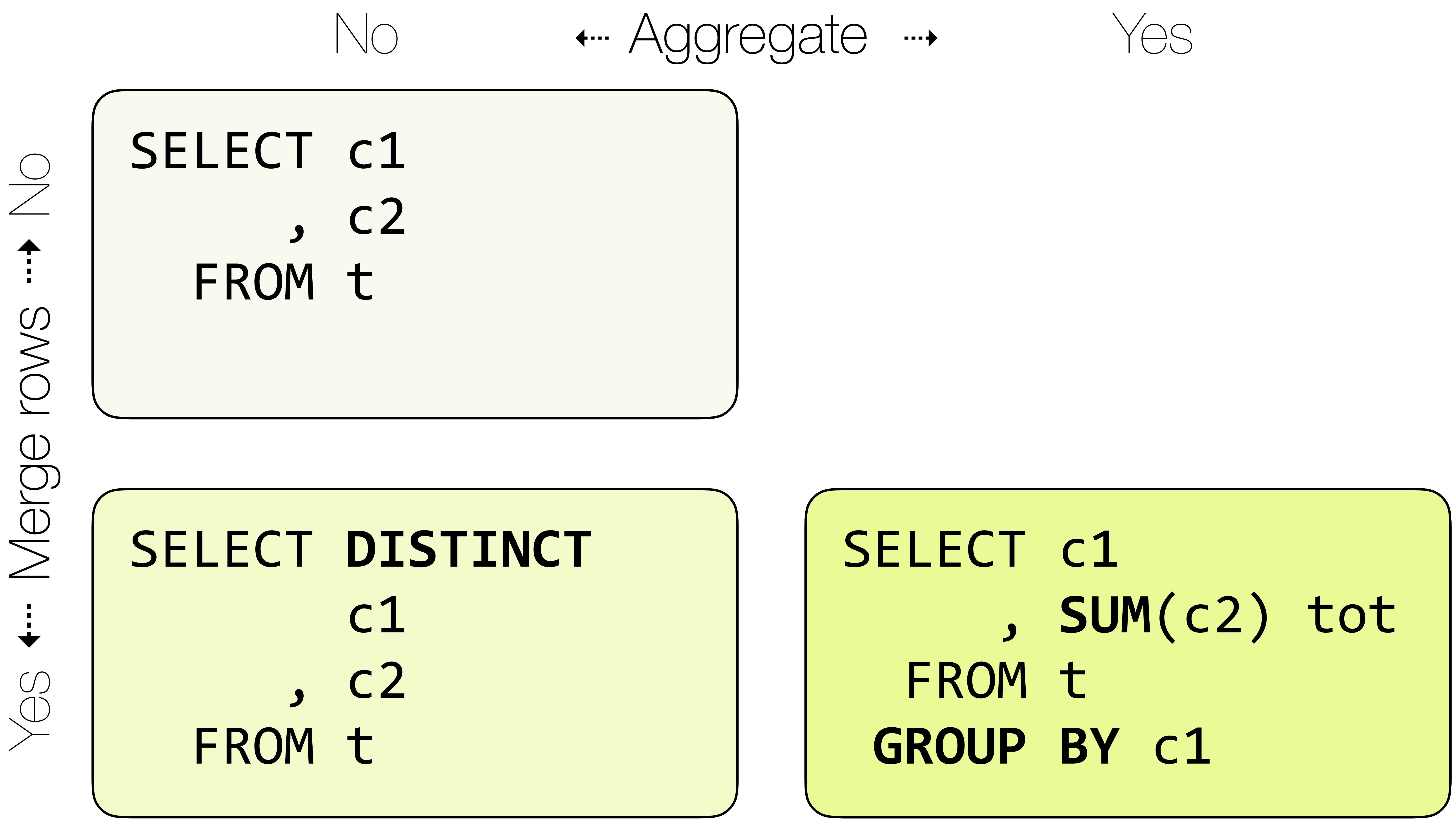
`OVER` (PARTITION BY) The Problem

Two distinct concepts could not be used independently:

- Merge rows with the same key properties
  - **`GROUP BY`** to specify key properties
  - **`DISTINCT`** to use full row as key
- Aggregate data from related rows
  - Requires **`GROUP BY`** to segregate the rows
  - **`COUNT`**, **`SUM`**, **`AVG`**, **`MIN`**, **`MAX`** to aggregate grouped rows

OVER (PARTITION BY)

The Problem

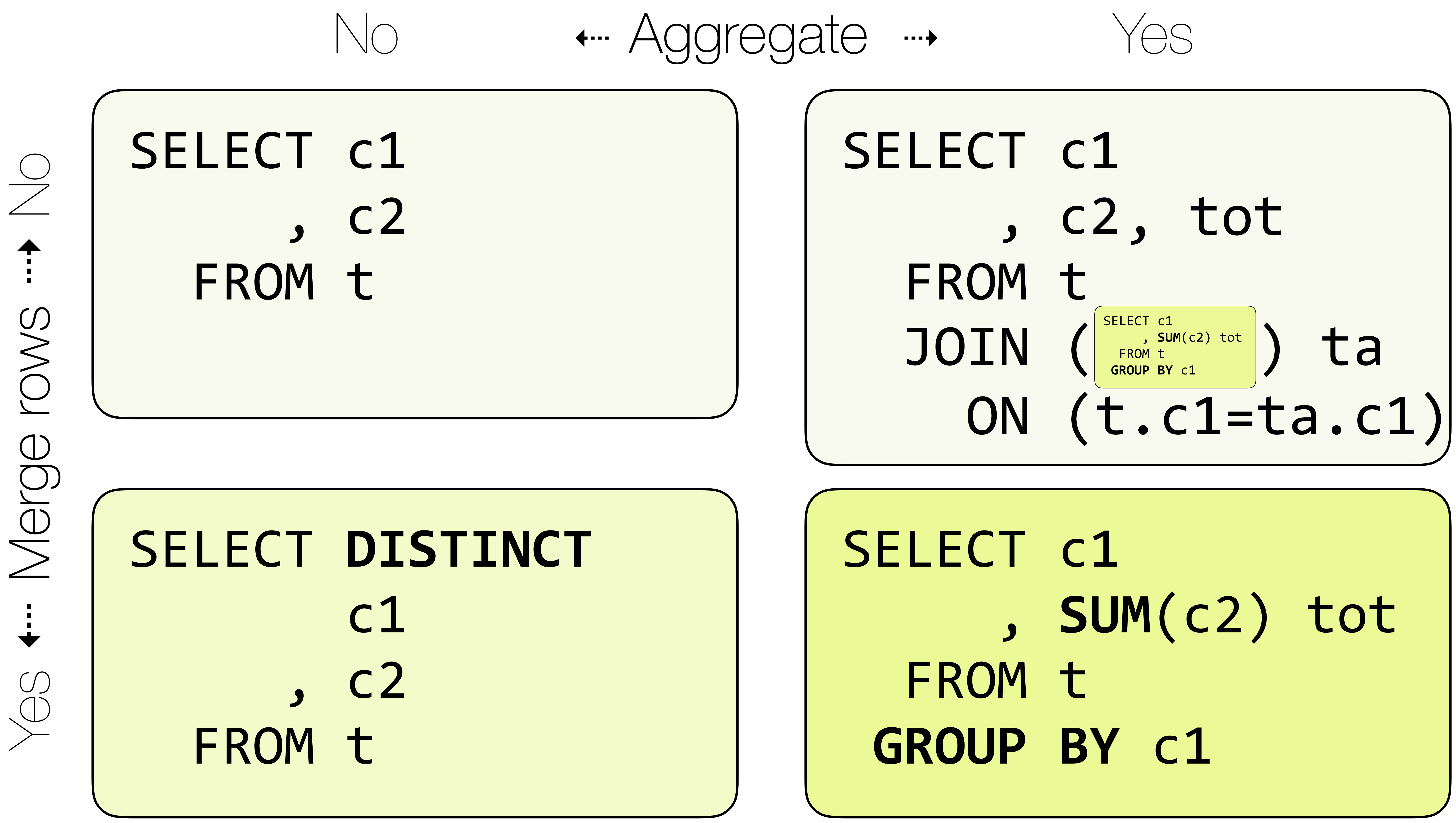


```
SELECT c1
     , c2
  FROM t
```

```
SELECT DISTINCT
       c1
     , c2
  FROM t
```

```
SELECT c1
     , SUM(c2) tot
  FROM t
 GROUP BY c1
```

OVER (PARTITION BY) The Problem



```
SELECT c1
     , c2
  FROM t
```

```
SELECT c1
     , c2, tot
  FROM t
  JOIN (SELECT c1
             , SUM(c2) tot
          FROM t
         GROUP BY c1) ta
    ON (t.c1=ta.c1)
```

```
SELECT DISTINCT
       c1
     , c2
  FROM t
```

```
SELECT c1
     , SUM(c2) tot
  FROM t
 GROUP BY c1
```

OVER (PARTITION BY)

The Problem

No ⇠ Aggregate ⇢ Yes

Yes ⇠ Merge rows ⇢ No

```
SELECT c1
     , c2
  FROM t
```

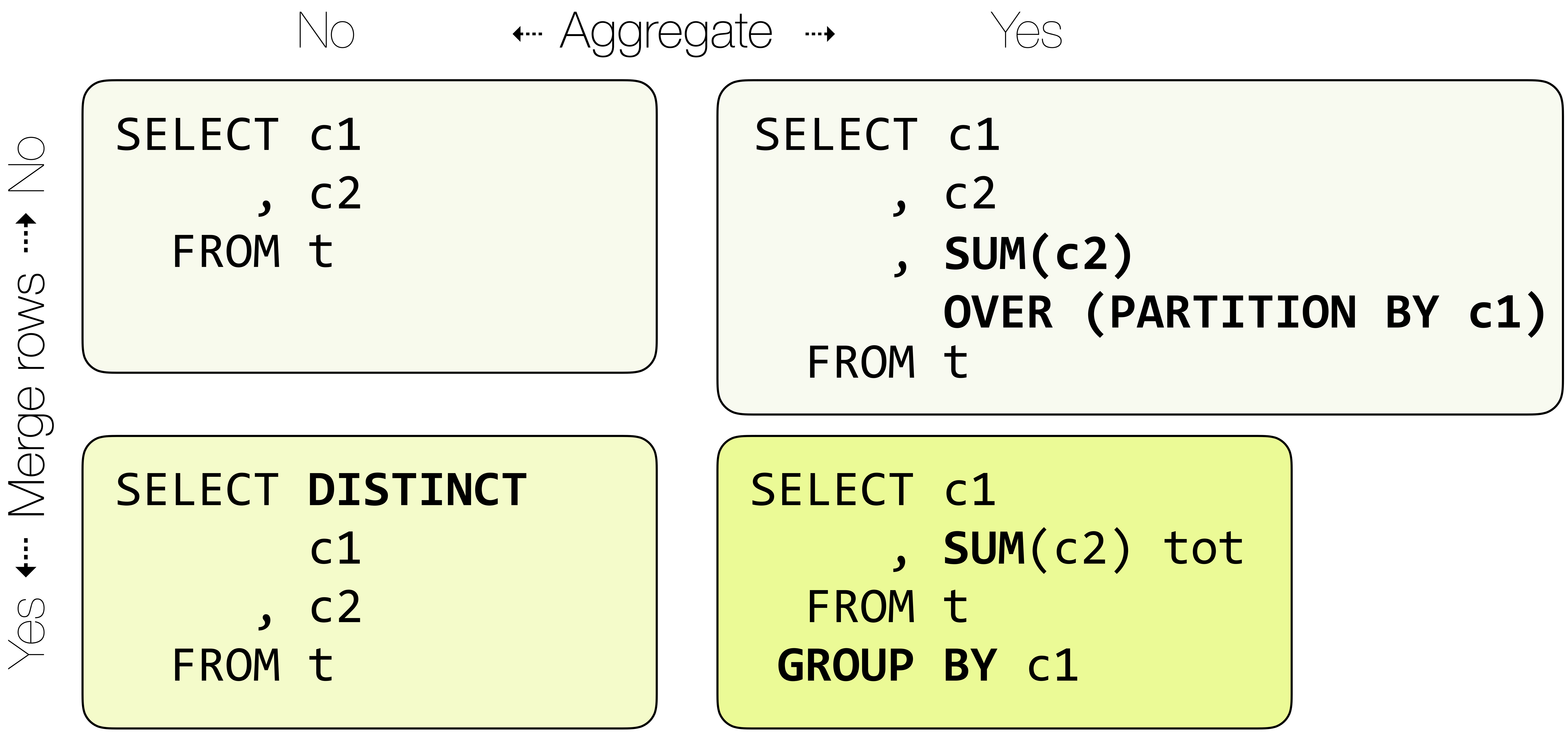
```
SELECT
  F
  JO             ta
                a.c1)
```

```
SELECT DISTINCT
       c1
     , c2
  FROM t
```

```
SELECT c1
     , SUM(c2) tot
  FROM t
 GROUP BY c1
```

OVER (PARTITION BY)

Since SQL:2003



```
SELECT c1
     , c2
  FROM t
```

```
SELECT c1
     , c2
     , SUM(c2)
       OVER (PARTITION BY c1)
  FROM t
```

```
SELECT DISTINCT
       c1
     , c2
  FROM t
```

```
SELECT c1
     , SUM(c2) tot
  FROM t
 GROUP BY c1
```

OVER (PARTITION BY) How it works

```
SELECT dep,
       salary


  FROM emp
```

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

OVER (PARTITION BY)

How it works

```
SELECT dep,
       salary,


  FROM emp
```

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

OVER (PARTITION BY) 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 (PARTITION BY)

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 (PARTITION BY) 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 (PARTITION BY)

How it works

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

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

OVER (PARTITION BY)

How it works

```
SELECT dep,
       salary,
       SUM(salary)
       OVER(PARTITION BY dep)
  FROM emp
```

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

OVER

and

ORDER BY

(Framing & Ranking)

OVER (ORDER BY)

The Problem

```
SELECT id,
       value
  FROM transactions t
```

id	value
1	+10
2	+20
3	-10
4	+50
5	-30
6	-20

OVER (ORDER BY)

The Problem

```
SELECT id,
       value,
       (SELECT SUM(value)
          FROM transactions t2
         WHERE t2.id <= t.id)
  FROM transactions t
```

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY)

The Problem

```
SELECT id,
       value,
       (SELECT SUM(value)
           FROM transactions t2
          WHERE t2.id <= t.id)
  FROM transactions t
```

Range segregation (<=)
not possible with
GROUP BY or
PARTITION BY

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY)

Since SQL:2003

```
SELECT id,
       value,
       SUM(value)
       OVER (
        ORDER BY id




       )
  FROM transactions t
```

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY)

Since SQL:2003

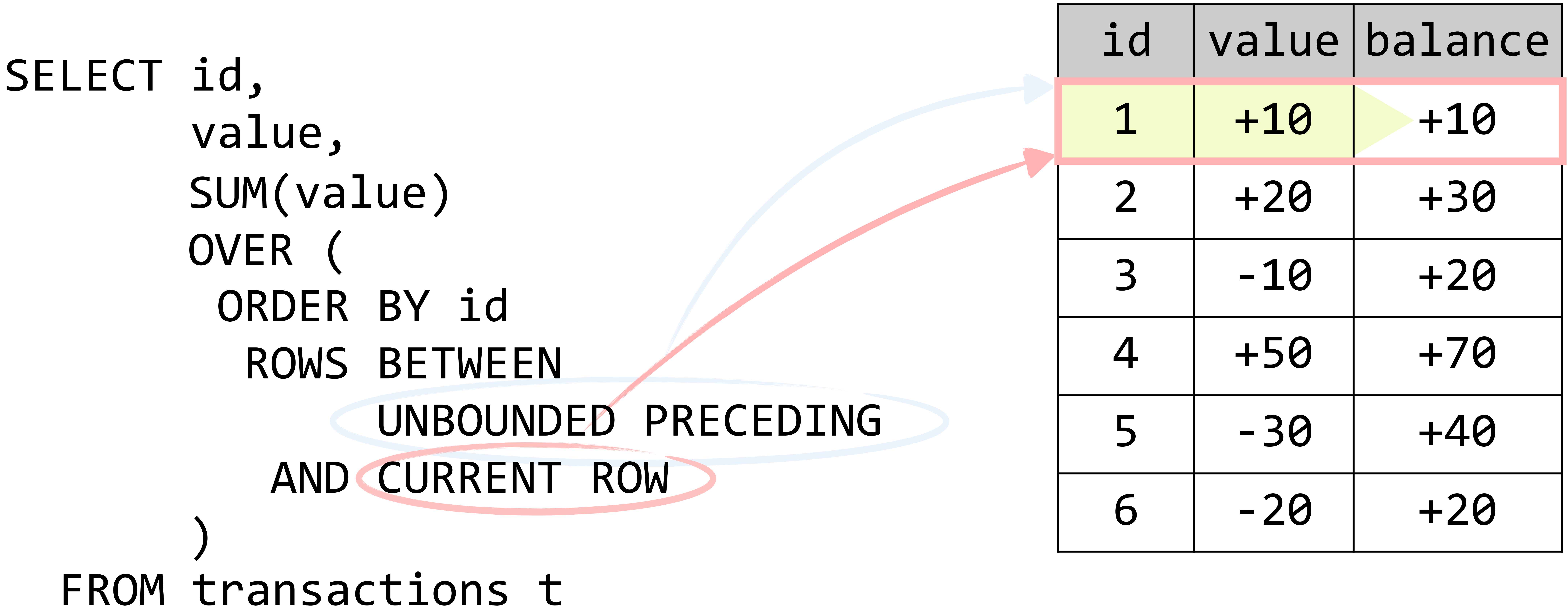
```
SELECT id,
       value,
       SUM(value)
       OVER (
        ORDER BY id
         ROWS BETWEEN
              UNBOUNDED PRECEDING

       )
  FROM transactions t
```

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY)

Since SQL:2003



```
SELECT id,
       value,
       SUM(value)
       OVER (
        ORDER BY id
         ROWS BETWEEN
              UNBOUNDED PRECEDING
          AND CURRENT ROW
       )
  FROM transactions t
```

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY)

Since SQL:2003

```
SELECT id,
       value,
       SUM(value)
       OVER (
        ORDER BY id
         ROWS BETWEEN
              UNBOUNDED PRECEDING
          AND CURRENT ROW
       )
  FROM transactions t
```

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY)

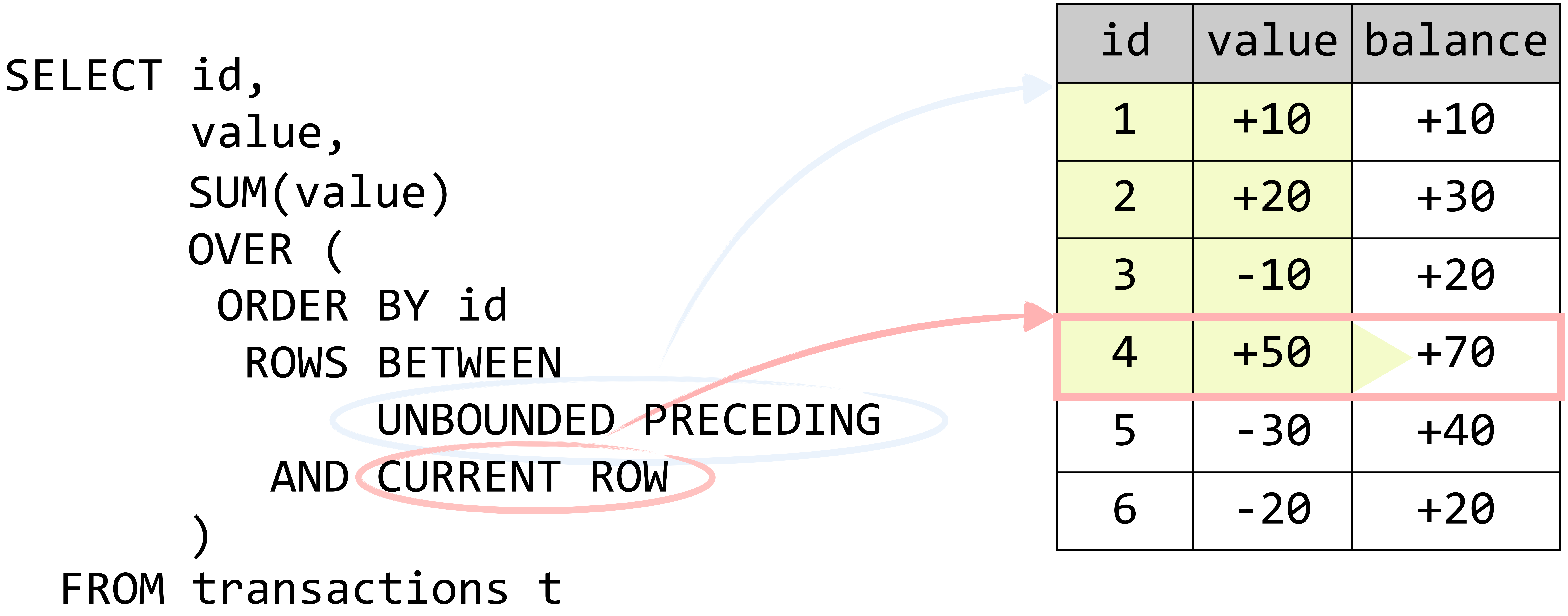
Since SQL:2003

```
SELECT id,
       value,
       SUM(value)
       OVER (
        ORDER BY id
         ROWS BETWEEN
              UNBOUNDED PRECEDING
          AND CURRENT ROW
       )
  FROM transactions t
```

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY)

Since SQL:2003

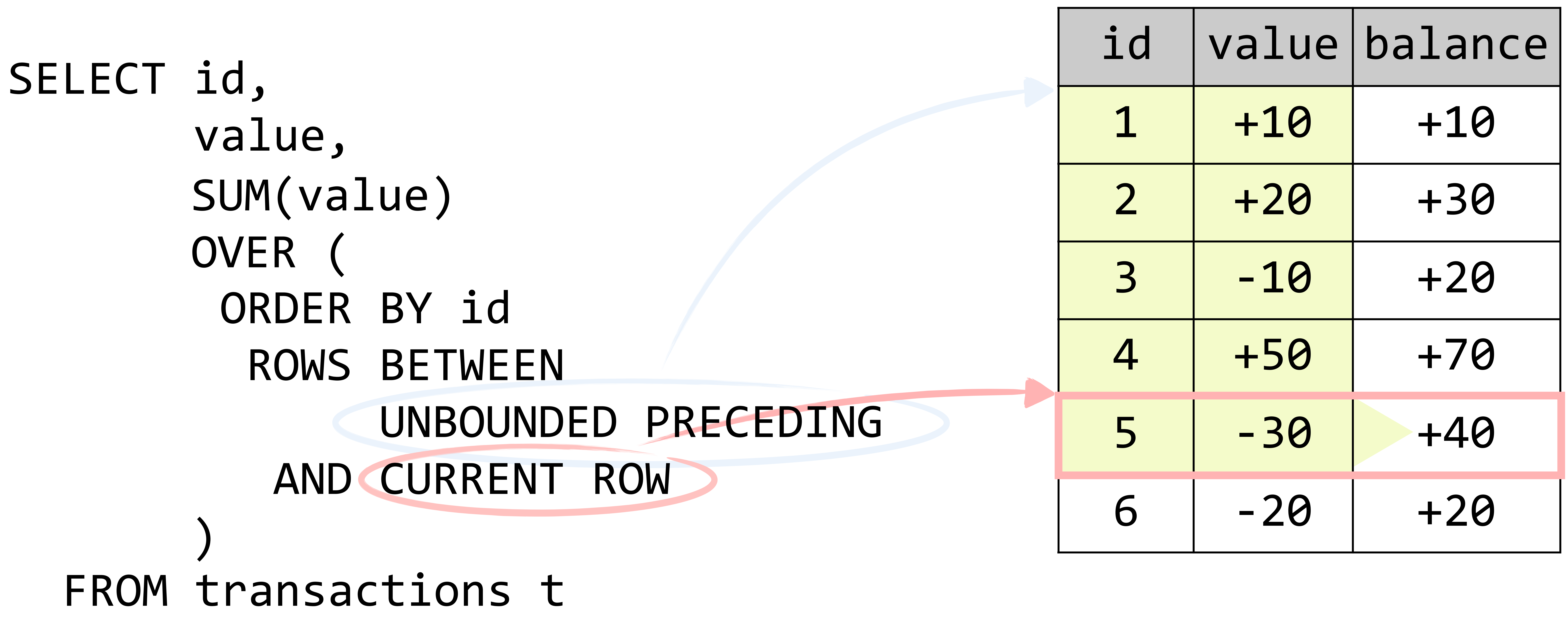


```
SELECT id,
       value,
       SUM(value)
       OVER (
        ORDER BY id
         ROWS BETWEEN
              UNBOUNDED PRECEDING
          AND CURRENT ROW
       )
  FROM transactions t
```

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY)

Since SQL:2003



```
SELECT id,
       value,
       SUM(value)
       OVER (
        ORDER BY id
         ROWS BETWEEN
              UNBOUNDED PRECEDING
          AND CURRENT ROW
       )
  FROM transactions t
```

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY)

Since SQL:2003

```
SELECT id,
       value,
       SUM(value)
       OVER (
        ORDER BY id
         ROWS BETWEEN
              UNBOUNDED PRECEDING
          AND CURRENT ROW
       )
  FROM transactions t
```

id	value	balance
1	+10	+10
2	+20	+30
3	-10	+20
4	+50	+70
5	-30	+40
6	-20	+20

OVER (ORDER BY) Since SQL:2003

```
SELECT id,
       value,
       SUM(value)
       OVER (PARTITION BY acnt
        ORDER BY id
         ROWS BETWEEN
              UNBOUNDED PRECEDING
          AND CURRENT ROW
       )
  FROM transactions t
```

acnt	id	value	balance
1	1	+10	+10
22	2	+20	+20
22	3	-10	+10
333	4	+50	+50
333	5	-30	+20
333	6	-20	0

OVER (ORDER BY) Since SQL:2003

With **OVER (ORDER BY n)** a new type of functions make sense:

n	ROW_NUMBER	RANK	DENSE_RANK	PERCENT_RANK	CUME_DIST
1	1	1	1	0	0.25
2	2	2	2	0.33…	0.75
3	3	2	2	0.33…	0.75
4	4	4	3	1	1

OVER (SQL:2003) Use Cases

- Aggregates without GROUP BY
- Running totals, moving averages

```
AVG(…) OVER(ORDER BY …
             ROWS BETWEEN 3 PRECEDING
                      AND 3 FOLLOWING) moving_avg
```

- Ranking
  - Top-N per Group

```
SELECT *
  FROM (SELECT ROW_NUMBER()
               OVER(PARTITION BY … ORDER BY …) rn
             , t.*
          FROM t) numbered_t
WHERE rn <= 3
```

- Avoiding self-joins

[… many more …]

`OVER` (SQL:2003) In a Nutshell

`OVER` may follow any aggregate function

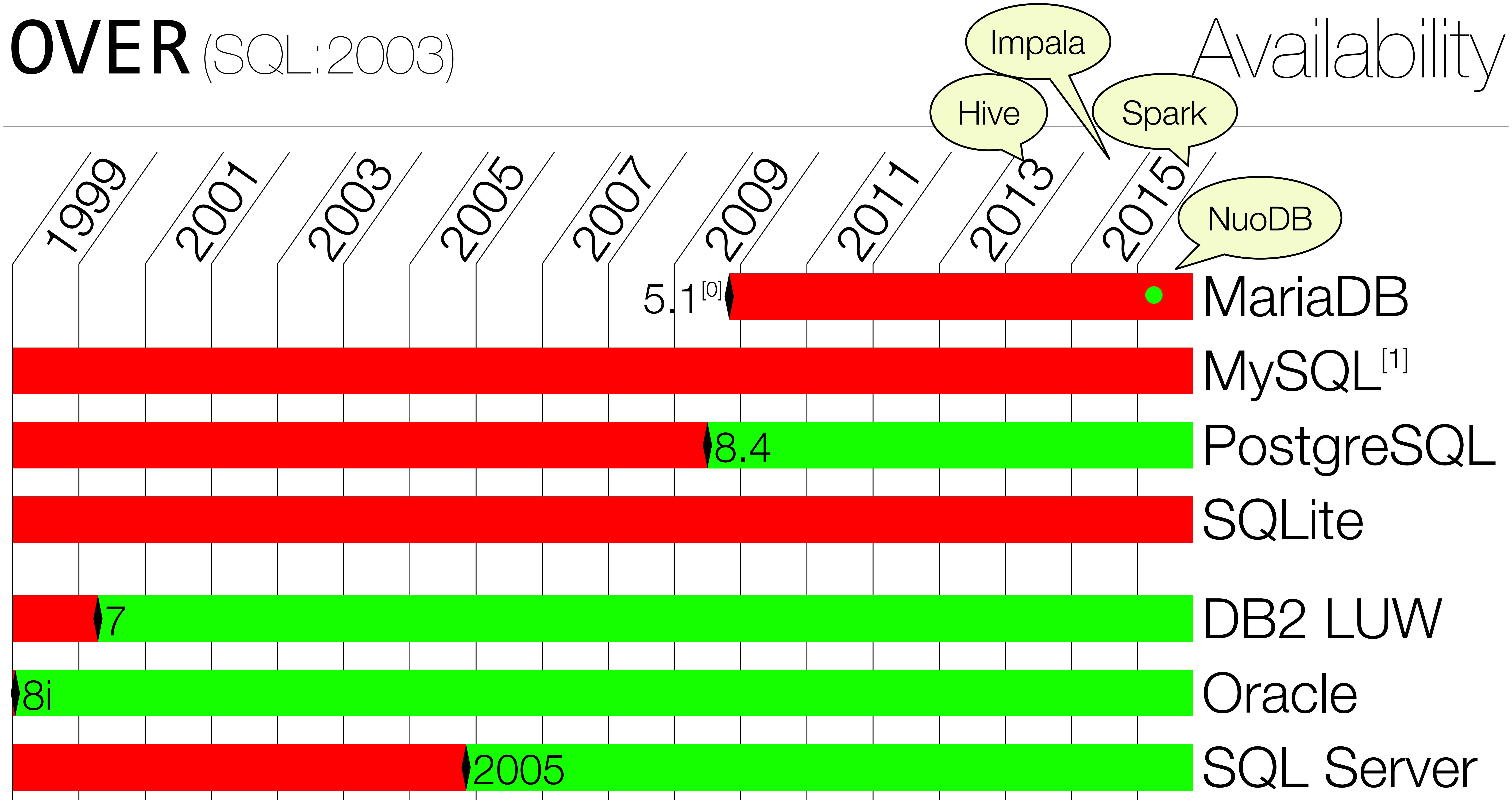
`OVER` defines which rows are visible at each row

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

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

`OVER(ORDER BY … BETWEEN)` segregates using `<, >`

OVER (SQL:2003) Availability



[0]Available MariaDB 10.2 alpha

[1]On the roadmap: http://www.slideshare.net/ManyiLu/optimizer-percona-liveams2015/47

WITHIN GROUP

WITHIN GROUP

The Problem

Grouped rows cannot be ordered prior aggregation.

(how to get the middle value (median) of a set)

```
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 d3)
```

WITHIN GROUP

The Problem

Grouped rows cannot be ordered prior aggregation.

(how to get the middle value (median) of a set)

```
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 d3)
```

Number rows

Pick middle one

WITHIN GROUP

The Problem

Grouped rows cannot be ordered prior aggregation.

(how to get the middle value (median) of a set)

```
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 d3)
```

Number rows

Pick middle one

WITHIN GROUP

The Problem

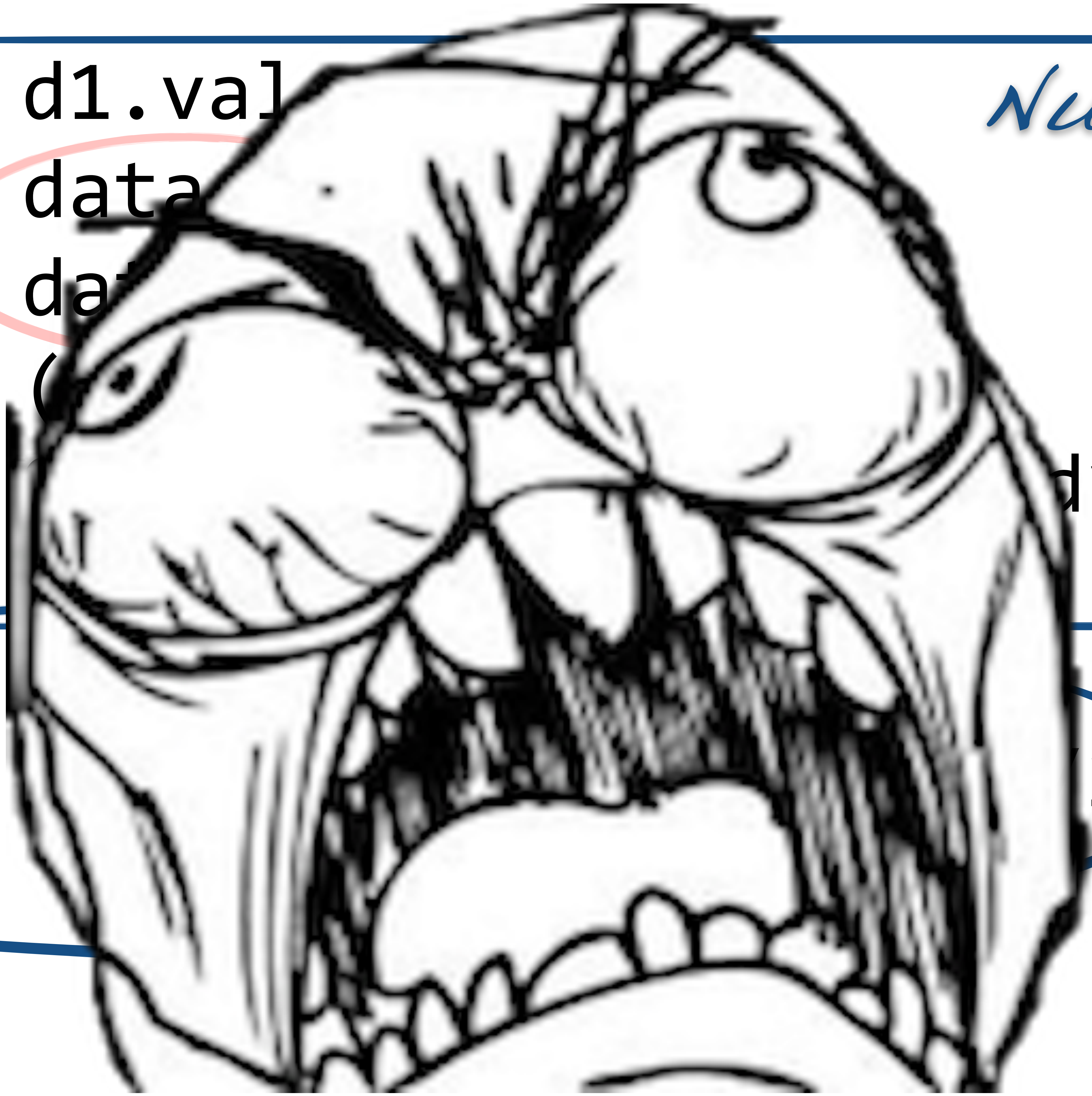
Grouped rows cannot be ordered prior aggregation.

(how to get the middle value (median) of a set)

```
SELECT d1.val
  FROM data
  JOIN da
    ON (
                             d1.id<d2.id))
 GROUP
HAVING
                              2)
```

Number rows

Pick middle one



WITHIN GROUP Since 2013

SQL:2003 introduced ordered set functions:

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

Median

Which value?

WITHIN GROUP Since 2013

SQL:2003 introduced ordered set functions:

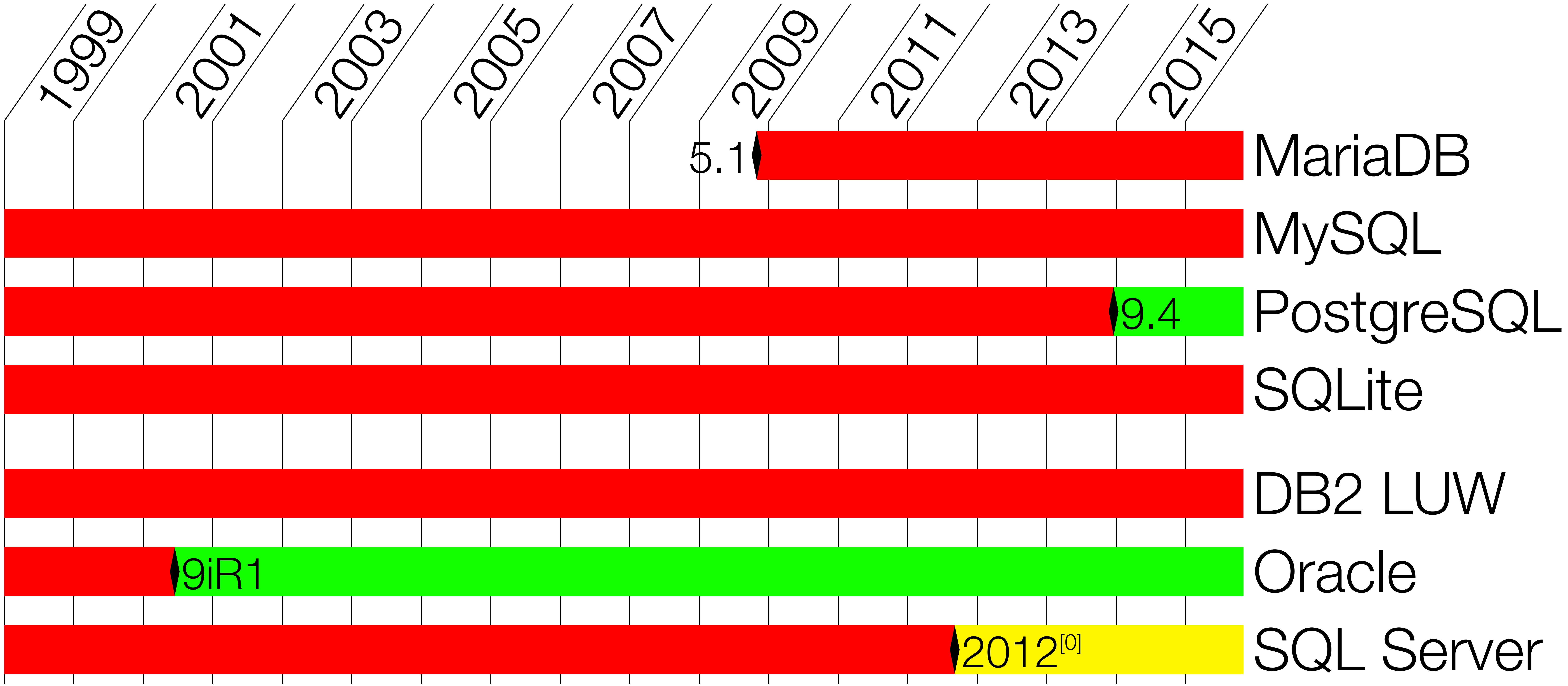
```
SELECT PERCENTILE_DISC(0.5)
       WITHIN GROUP (ORDER BY val)
  FROM data
```

…and hypothetical set-functions:

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

WITHIN GROUP

Availability

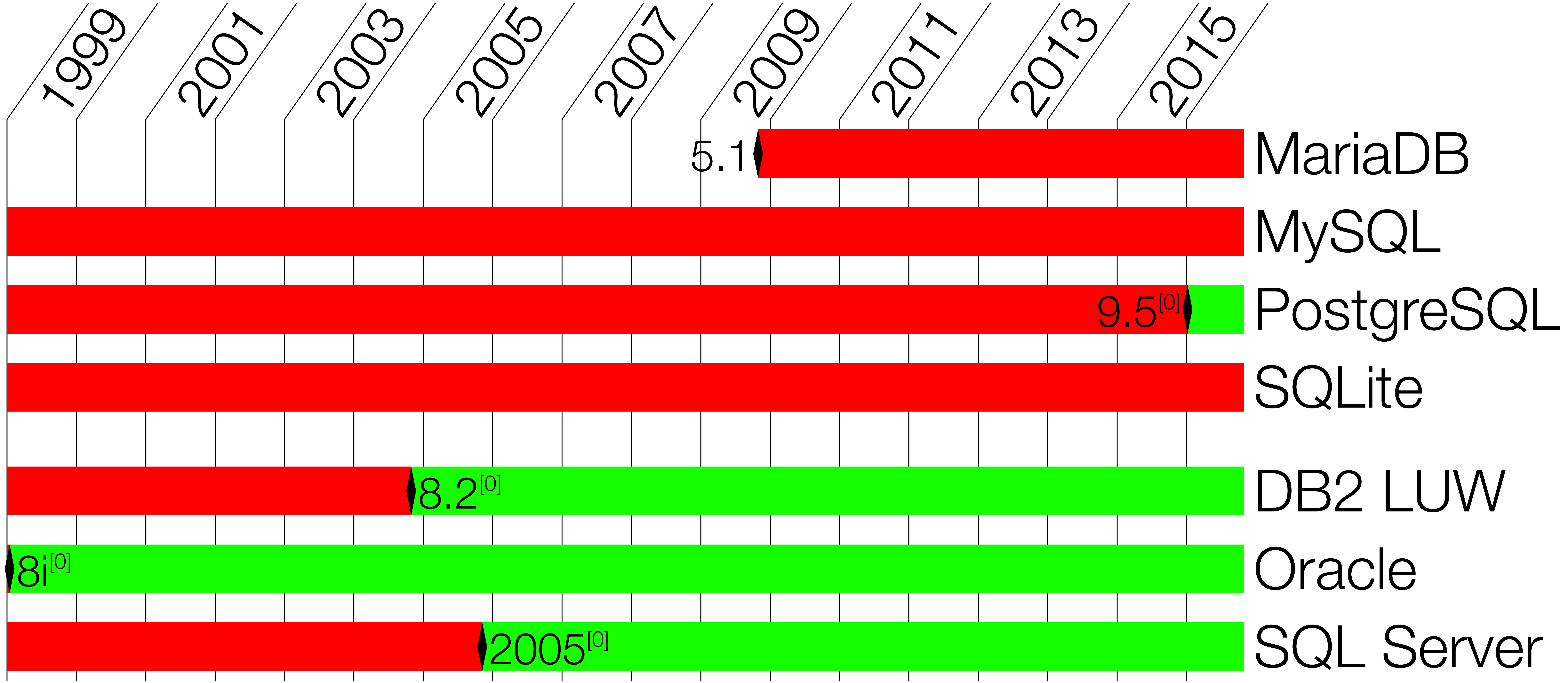


[0] Only as window function (OVER required). Feature request 728969 closed as "won't fix"

TABLESAMPLE

TABLESAMPLE

Availability



[0]Not for derived tables

SQL:2008

FETCH FIRST

FETCH FIRST

The Problem

Limit the result to a number of rows.

(`LIMIT`, `TOP` and `ROWNUM` are all proprietary)

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

SQL:2003 introduced `ROW_NUMBER()` to number rows.

But this still requires wrapping to limit the result.

And how about databases not supporting `ROW_NUMBER()`?

FETCH FIRST

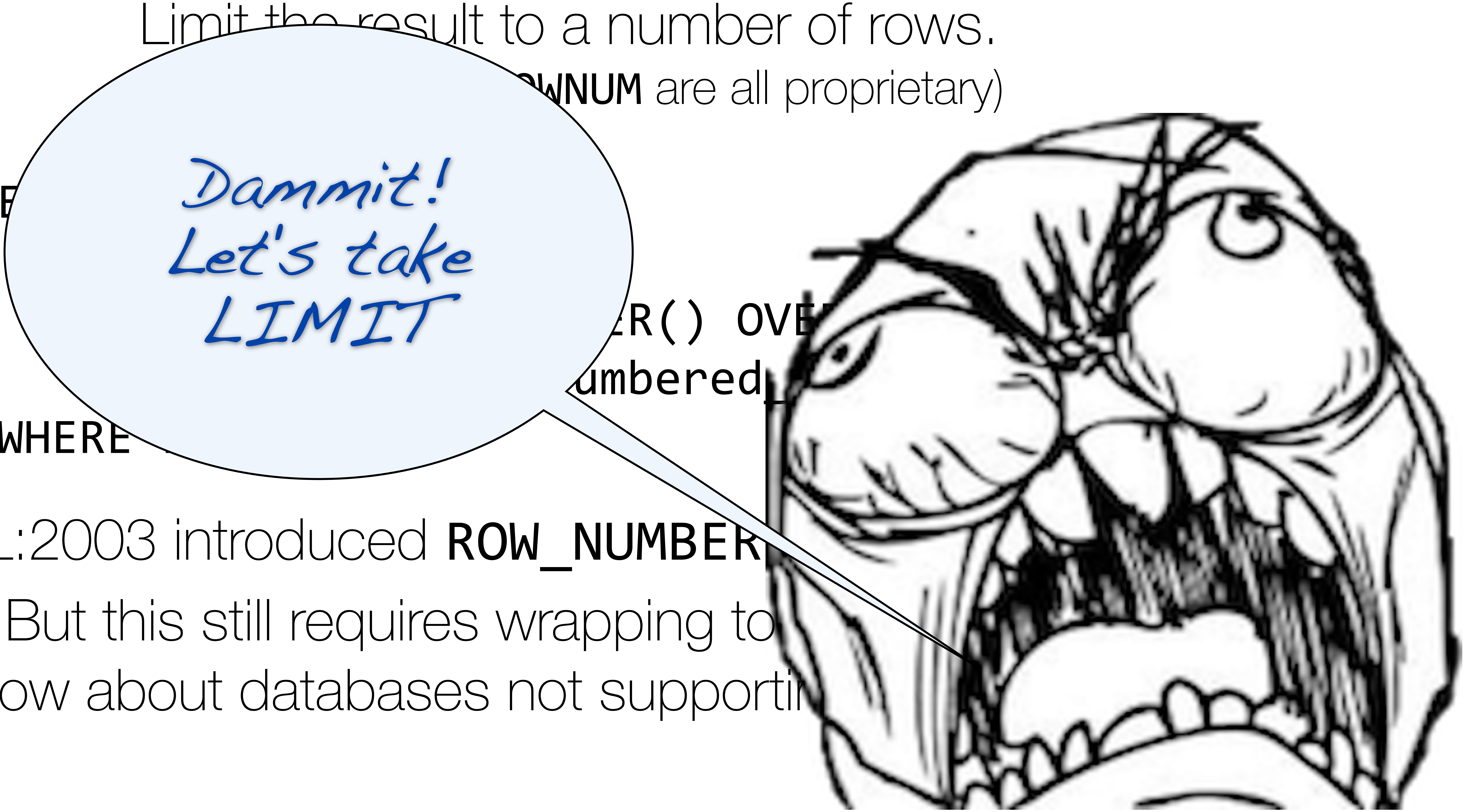
The Problem

Limit the result to a number of rows.

OWNUM are all proprietary)

```
SE

                              ER() OVE
                             umbered_
 WHERE
```



SQL:2003 introduced **ROW_NUMBER**

But this still requires wrapping to

And how about databases not supporti

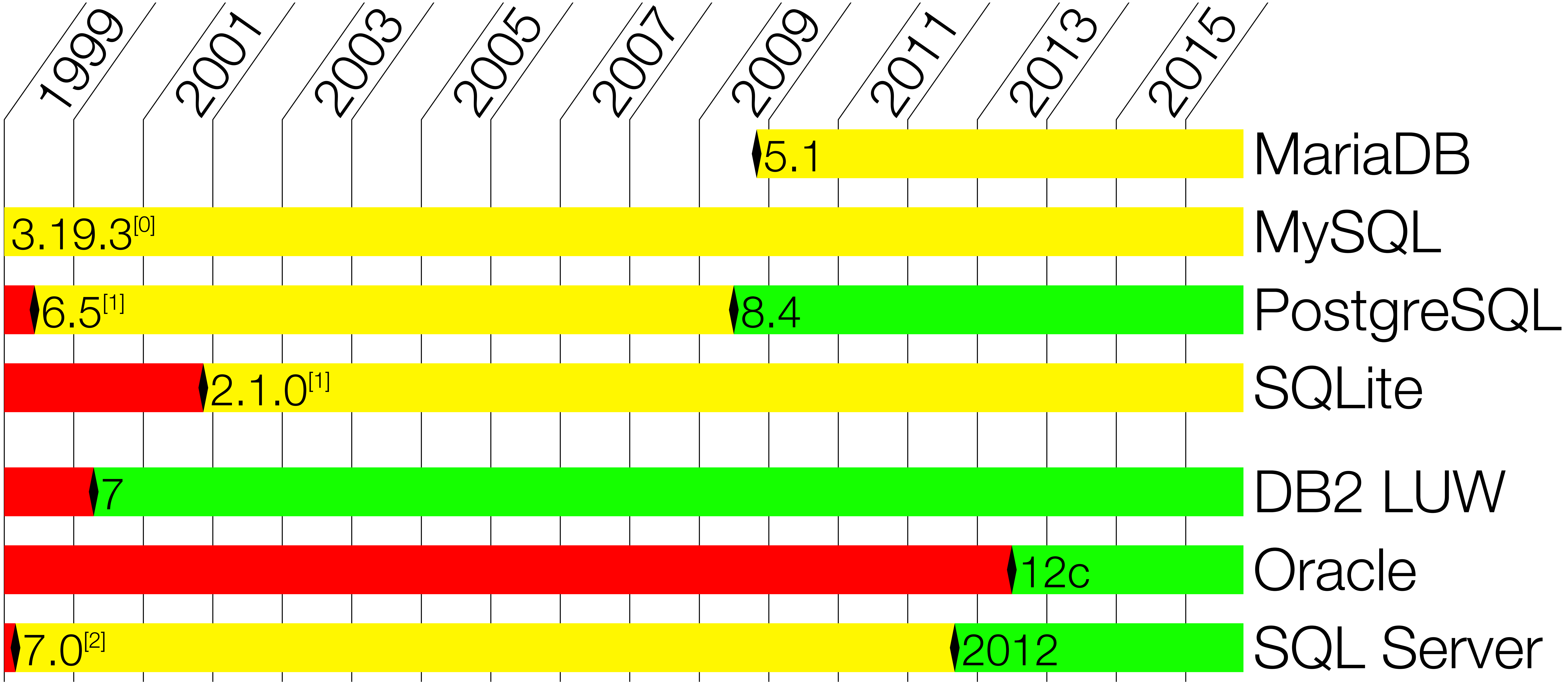
FETCH FIRST Since SQL:2008

SQL:2008 introduced the `FETCH FIRST … ROWS ONLY` clause:

```
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

SQL:2011

OFFSET

OFFSET

The Problem

How to fetch the rows after a limit?

(pagination anybody?)

```
SELECT *
  FROM (SELECT *
             , ROW_NUMBER() OVER(ORDER BY x) rn
          FROM data) numbered_data
 WHERE rn > 10 and rn <= 20
```

OFFSET Since SQL:2011

SQL:2011 introduced **OFFSET**, unfortunately!

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

`OFFSET` Since SQL:2011

SC `FSET`, unfortunately!

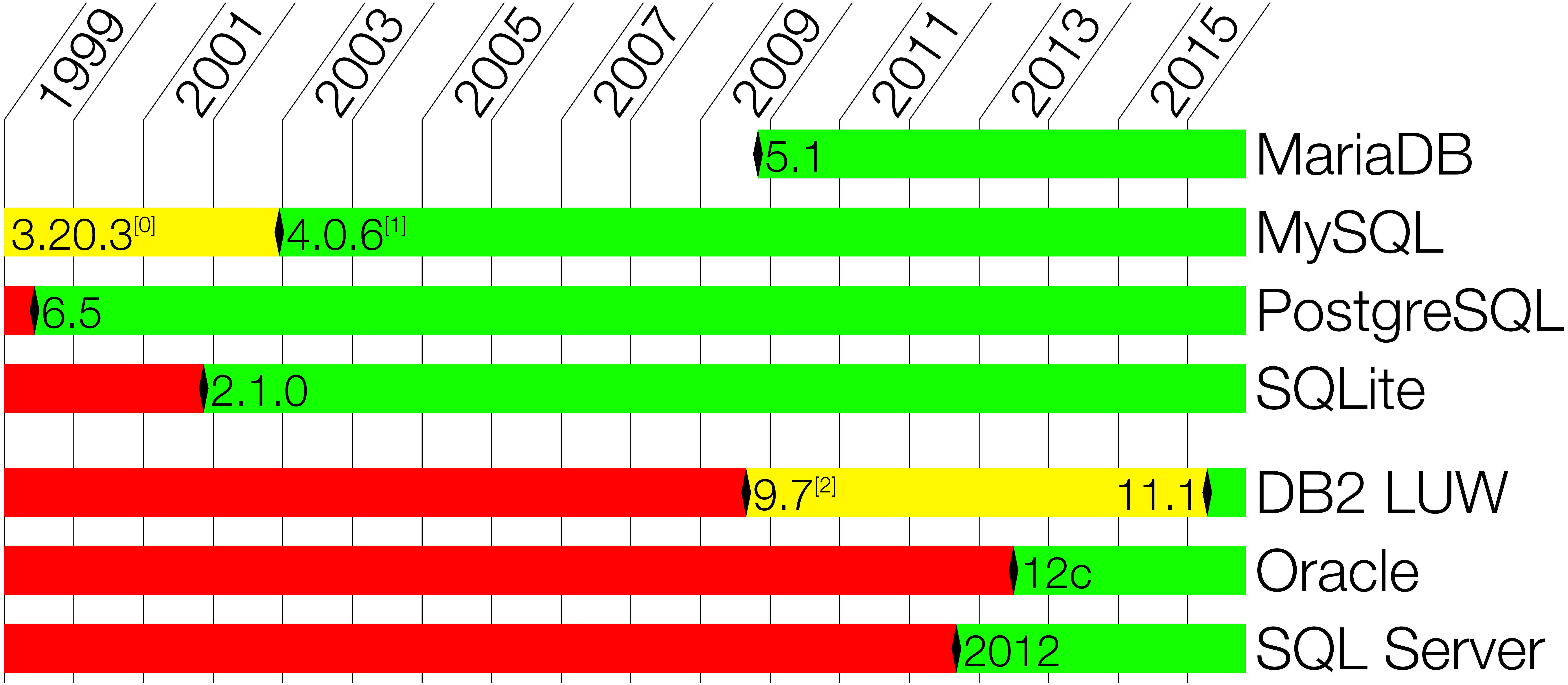
`10 ROWS`

Grab coasters & stickers!

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

OFFSET

Since SQL:2011



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

[1] The release notes say "Added PostgreSQL compatible `LIMIT` syntax"

[2] Requires enabling the MySQL compatibility vector: `db2set DB2_COMPATIBILITY_VECTOR=MYS`

OVER

OVER (SQL:2011) The Problem

Direct access of other rows of the same window is not possible.

(E.g., calculate the difference to the previous rows)

OVER (SQL:2011) The Problem

Direct access of other rows of the same window is not possible.

(E.g., calculate the difference to the previous rows)

```
SELECT *
  FROM t
```

balance	…
50	…
90	…
70	…
30	…

OVER (SQL:2011) The Problem

Direct access of other rows of the same window is not possible.

(E.g., calculate the difference to the previous rows)

```
SELECT *
     , ROW_NUMBER() OVER(ORDER BY x) rn
  FROM t
```

balance	…	rn
50	…	1
90	…	2
70	…	3
30	…	4

OVER (SQL:2011)

The Problem

Direct access of other rows of the same window is not possible.

(E.g., calculate the difference to the previous rows)

```
WITH numbered_t AS (SELECT *
                         , ROW_NUMBER() OVER(ORDER BY x) rn
                      FROM t)
SELECT curr.*


  FROM      numbered_t curr
```

curr		
balance	…	rn
50	…	1
90	…	2
70	…	3
30	…	4

OVER (SQL:2011) The Problem

Direct access of other rows of the same window is not possible.

(E.g., calculate the difference to the previous rows)

```
WITH numbered_t AS (SELECT *
                         , ROW_NUMBER() OVER(ORDER BY x) rn
                      FROM t)
SELECT curr.*



  FROM      numbered_t curr
  LEFT JOIN numbered_t prev
    ON (                   )
```

curr			prev		
balance	…	rn	balance	…	rn
50	…	1	50	…	1
90	…	2	90	…	2
70	…	3	70	…	3
30	…	4	30	…	4

OVER (SQL:2011)

The Problem

Direct access of other rows of the same window is not possible.

(E.g., calculate the difference to the previous rows)

```
WITH numbered_t AS (SELECT *
                         , ROW_NUMBER() OVER(ORDER BY x) rn
                      FROM t)
SELECT curr.*



  FROM      numbered_t curr
  LEFT JOIN numbered_t prev
    ON (curr.rn = prev.rn+1)
```

curr			prev		
balance	…	rn	balance	…	rn
50	…	1			
90	…	2	50	…	1
70	…	3	90	…	2
30	…	4	70	…	3

OVER (SQL:2011) The Problem

Direct access of other rows of the same window is not possible.

(E.g., calculate the difference to the previous rows)

```
WITH numbered_t AS (SELECT *
                         , ROW_NUMBER() OVER(ORDER BY x) rn
                      FROM t)
SELECT curr.*
     , curr.balance
       - COALESCE(prev.balance,0)
  FROM      numbered_t curr
  LEFT JOIN numbered_t prev
    ON (curr.rn = prev.rn+1)
```

curr			prev			
balance	…	rn	balance	…	rn	
50	…	1				+50
90	…	2	50	…	1	+40
70	…	3	90	…	2	-20
30	…	4	70	…	3	-40

OVER (SQL:2011) Since SQL:2011

SQL:2011 introduced **LEAD**, **LAG**, **NTH_VALUE**, … for that:

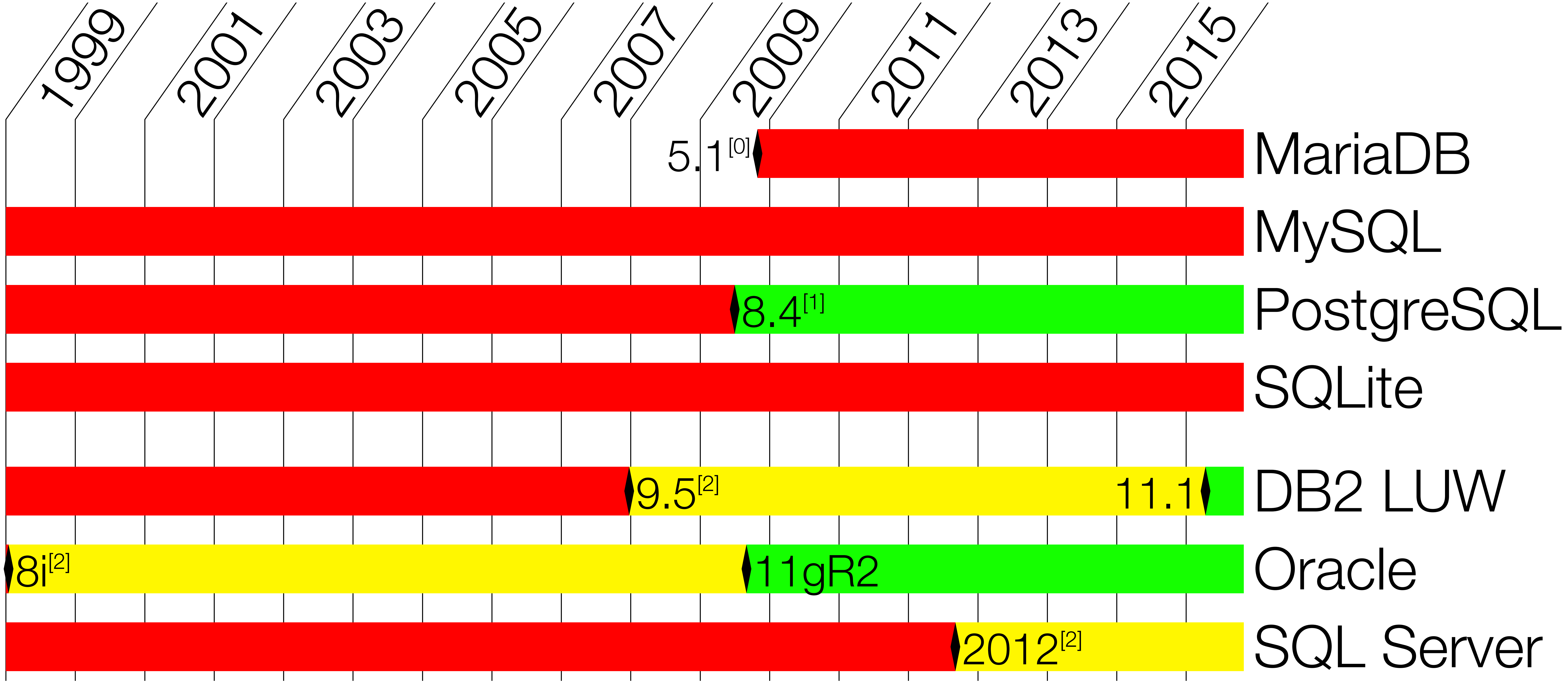
```
SELECT *, balance
        - COALESCE( LAG(balance)
                     OVER(ORDER BY x)
                   , 0)
  FROM t
```

Available functions:

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

OVER (LEAD, LAG, …)

Since SQL:2011



[0]Not yet available in MariaDB 10.2.2 (alpha). MDEV-8091
[1]No `IGNORE NULLS` and `FROM LAST` as of PostgreSQL 9.6
[2]No `NTH_VALUE`

Temporal Tables

(Time Traveling)

Temporal Tables

The Problem

```
INSERT
UPDATE
DELETE
```

are

DESTRUCTIVE

Temporal Tables

Since SQL:2011

Table can be system versioned, application versioned or both.

```
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
```

Temporal Tables

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	

Temporal Tables

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

Temporal Tables 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.

Temporal Tables

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 **FOR … 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

Temporal Tables

The Problem

It isn’t possible to define constraints to 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

Temporal Tables Since SQL:2011

SQL:2011 provides means to cope with temporal tables:

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

Temporal support in SQL:2011 goes way further.

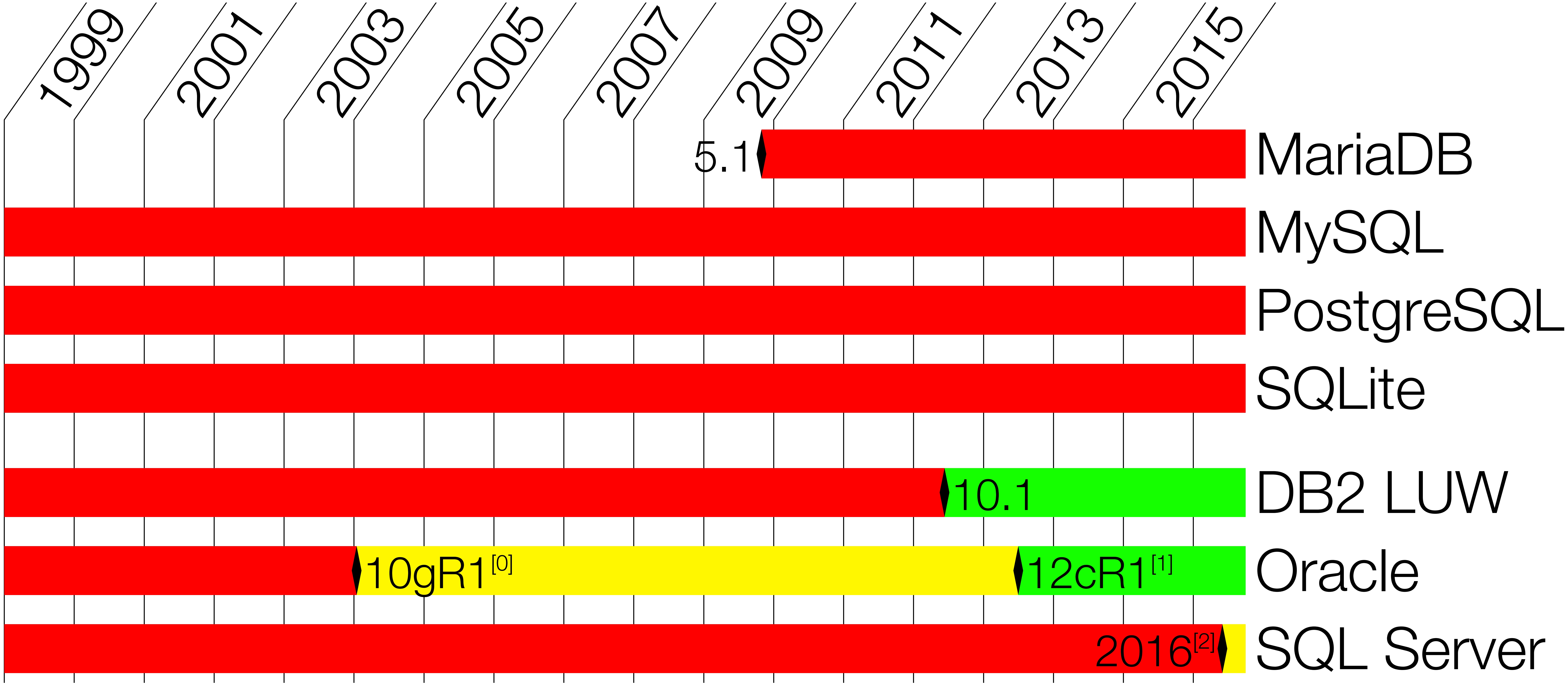
Please read this paper to get the idea:

Temporal features in SQL:2011

http://cs.ulb.ac.be/public/_media/teaching/infoh415/tempfeaturessql2011.pdf

Temporal Tables

Since SQL:2011



[0] Limited system versioning via Flashback

[1] Limited application versioning added (e.g. no WITHOUT OVERLAPS)

[2] Only system versioning

SQL:2016

(released: 2016-12-14)

MATCH_RECOGNIZE

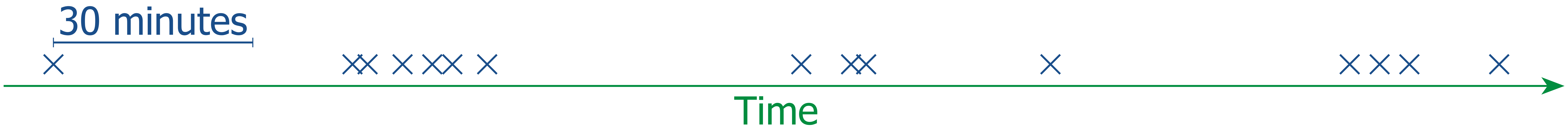
(Row Pattern Matching)

Row Pattern Matching

Example: Logfile

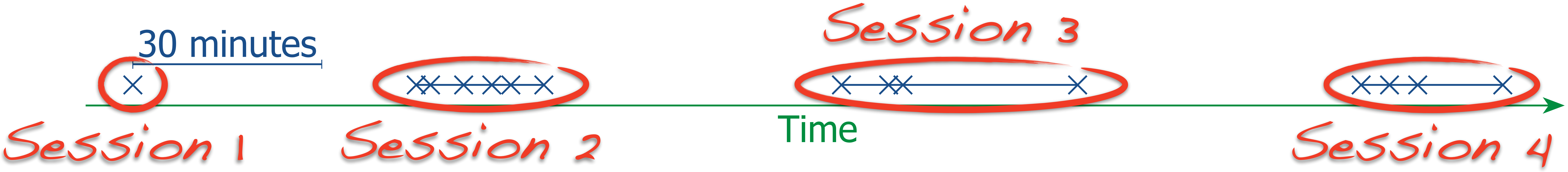
Row Pattern Matching

Example: Logfile



Row Pattern Matching

Example: Logfile



Example problem:

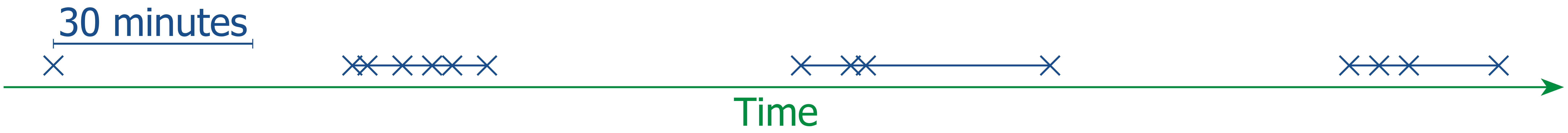
- Average session duration

Two approaches:

- Row pattern matching
- Start-of-group tagging

Row Pattern Matching

Since SQL:2016



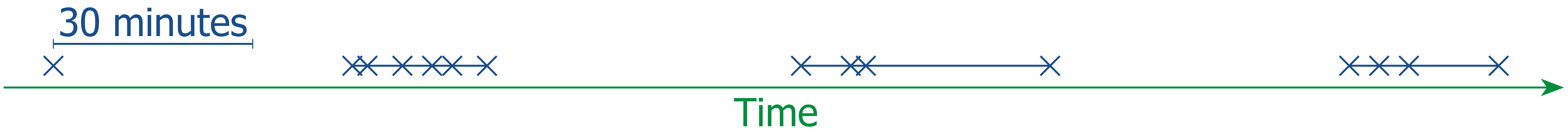
```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( new cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL '30' minute
```

define continuation

Oracle doesn't support avg on intervals — query doesn't work as shown

Row Pattern Matching

Since SQL:2016



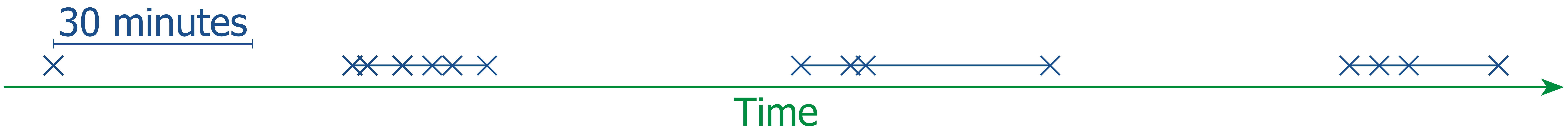
```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( new cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL '30' minute
```

undefined pattern variable: matches any row

Oracle doesn’t support avg on intervals — query doesn’t work as shown

Row Pattern Matching

Since SQL:2016



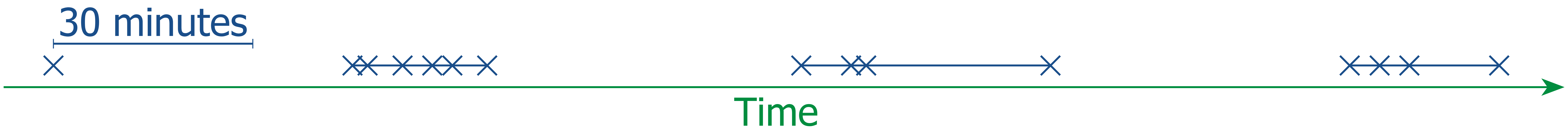
```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS
        ONE ROW PER MATCH
        PATTERN ( new cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL '30' minute
       ) t
```

any number of "cont" rows

Oracle doesn't support avg on intervals — query doesn't work as shown

Row Pattern Matching

Since SQL:2016



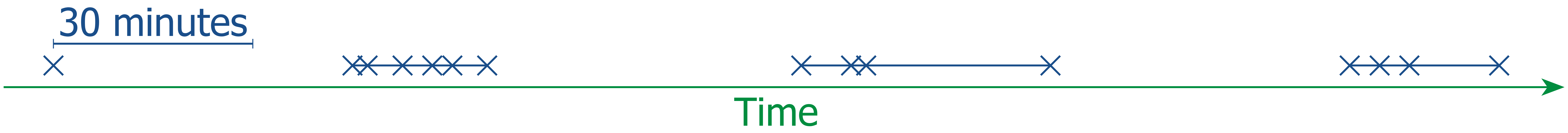
```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( new cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL '30' minute
       ) t
```

Very much like GROUP BY

Oracle doesn’t support avg on intervals — query doesn’t work as shown

Row Pattern Matching

Since SQL:2016



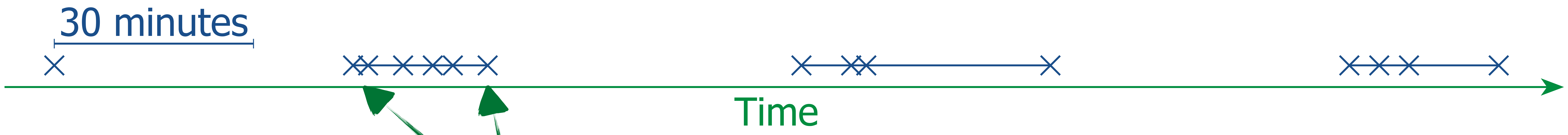
```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( new cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL '30' minute
       ) t
```

Very much like SELECT

Oracle doesn’t support avg on intervals — query doesn’t work as shown

Row Pattern Matching

Since SQL:2016

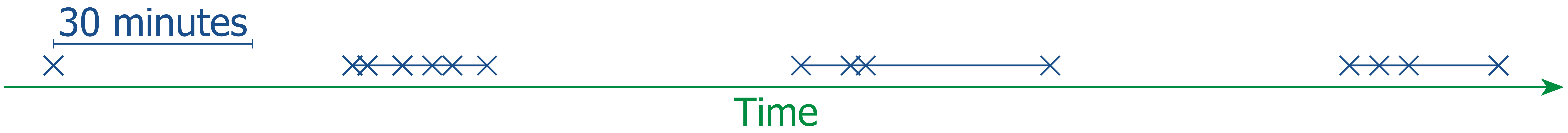


```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( new cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL '30' minute
       ) t
```

Oracle doesn’t support avg on intervals — query doesn’t work as shown

Row Pattern Matching

Since SQL:2016

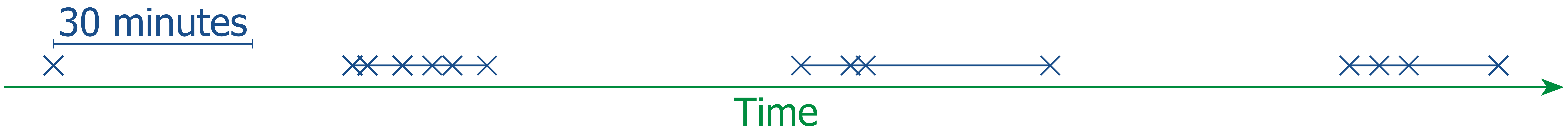


```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( new cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL '30' minute
       ) t
```

Oracle doesn’t support avg on intervals — query doesn’t work as shown

Row Pattern Matching

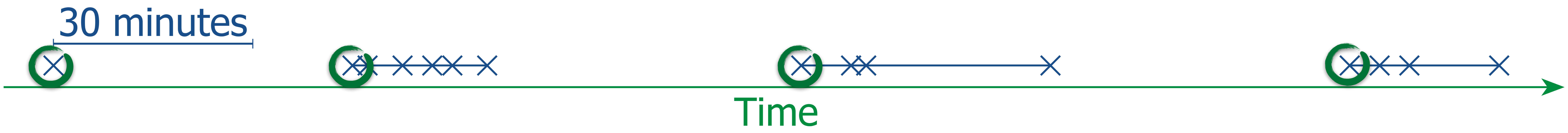
Before SQL:2016



Now, let’s try using window functions

Row Pattern Matching

Before SQL:2016

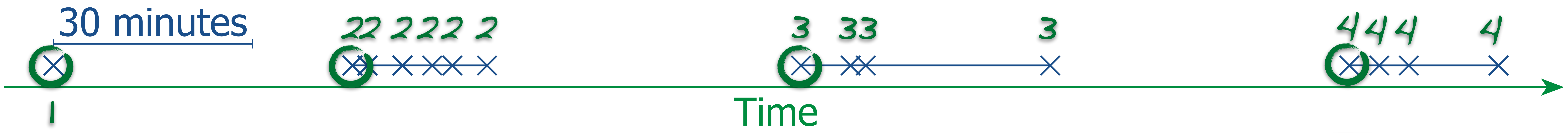


```
SELECT count(*) sessions, avg(duration) avg_duration
  FROM (SELECT MAX(ts) - MIN(ts) duration
          FROM (SELECT ts, COUNT(grp_start) OVER(ORDER BY ts) session_no
                  FROM (SELECT ts, CASE WHEN ts >= LAG( ts, 1, DATE’1900-01-1' )
                                                   OVER( ORDER BY ts )
                                                   + INTERVAL '30' minute
                                        THEN 1
                                    END grp_start
                          FROM log
                       ) tagged
               ) numbered
         GROUP BY session_no
       ) grouped
```

Start-of-group tags

Row Pattern Matching

Before SQL:2016

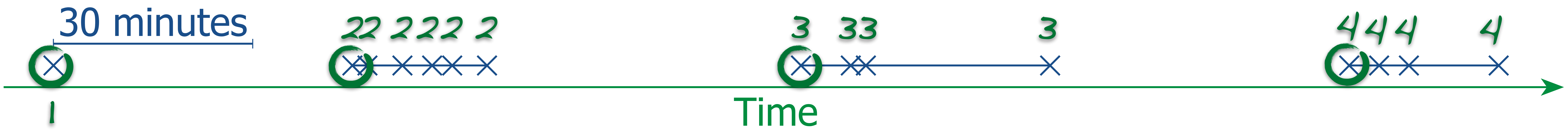


```
SELECT count(*) sessions, avg(duration) avg_duration
  FROM (SELECT MAX(ts) - MIN(ts) duration
          FROM (SELECT ts, COUNT(grp_start) OVER(ORDER BY ts) session_no
                  FROM (SELECT ts, CASE WHEN ts >= LAG( ts, 1, DATE’1900-01-1' )
                                                   OVER( ORDER BY ts )
                                                   + INTERVAL '30' minute
                                        THEN 1
                                    END grp_start
                          FROM log
                       ) tagged
               ) numbered
         GROUP BY session_no
       ) grouped
```

number sessions

Row Pattern Matching

Before SQL:2016



```
SELECT count(*) sessions, avg(duration) avg_duration
  FROM (SELECT MAX(ts) - MIN(ts) duration
          FROM (SELECT ts, COUNT(grp_start) OVER(ORDER BY ts) session_no
                  FROM (SELECT ts, CASE WHEN ts >= LAG( ts, 1, DATE’1900-01-1' )
                                                    OVER( ORDER BY ts )
                                                  + INTERVAL '30' minute
                                         THEN 1
                                     END grp_start
                          FROM log
                       ) tagged
               ) numbered
         GROUP BY session_no
       ) grouped
```

Row Pattern Matching

Since SQL:2016

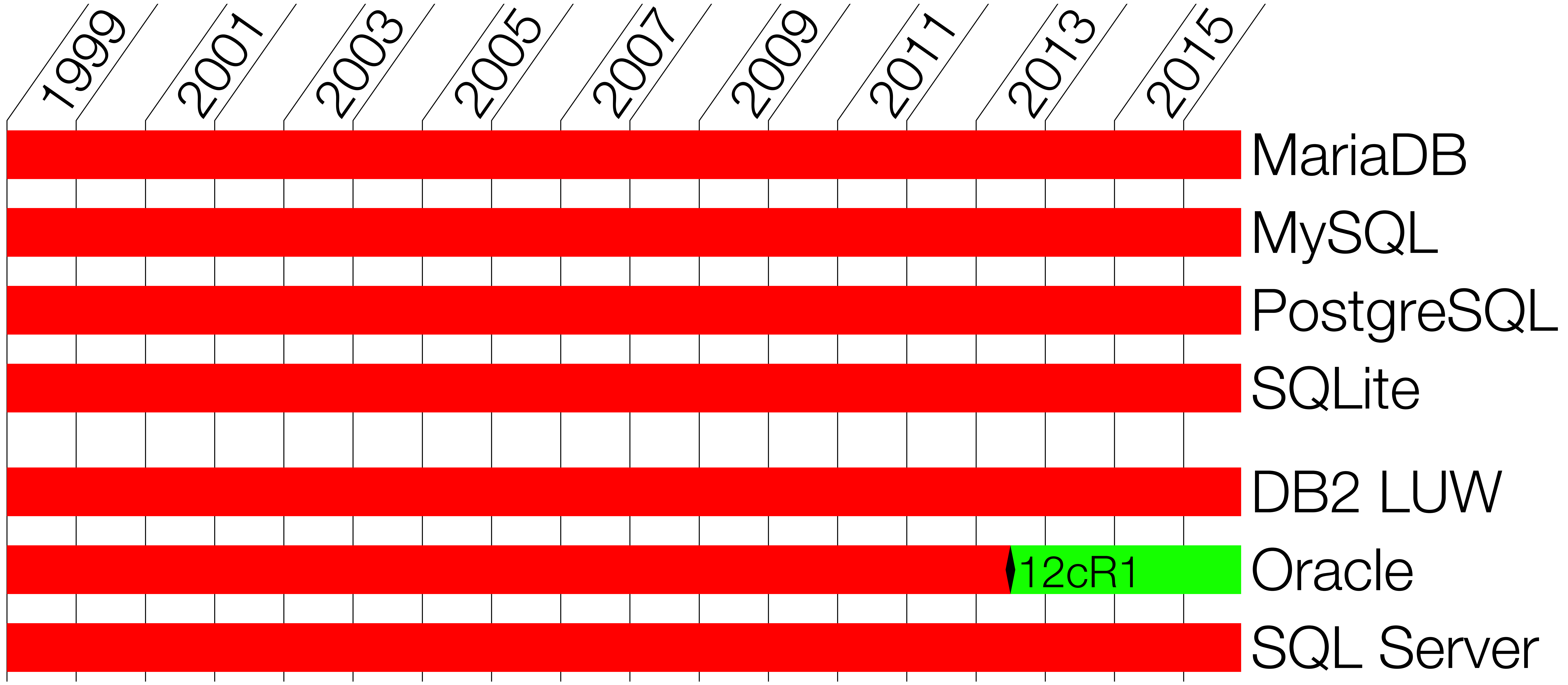
Row Pattern Matching

@MarkusWinand

https://www.slideshare.net/MarkusWinand/row-pattern-matching-in-sql2016

Row Pattern Matching

Availability



LIST_AGG

LIST_AGG

Since SQL:2016

grp	val
1	B
1	A
1	C
2	X

LIST_AGG

Since SQL:2016

grp	val
1	B
1	A
1	C
2	X

```
SELECT grp
     , LIST_AGG(val, ', ')
       WITHIN GROUP (ORDER BY val)
  FROM t
 GROUP BY grp
```

LIST_AGG

Since SQL:2016

grp	val
1	B
1	A
1	C
2	X

```
SELECT grp
     , LIST_AGG(val, ', ')
       WITHIN GROUP (ORDER BY val)
  FROM t
 GROUP BY grp
```

grp	val
1	A, B, C
2	X

LIST_AGG

Since SQL:2016

grp	val
1	B
1	A
1	C
2	X

```
SELECT grp
     , LIST_AGG(val, ', ')
       WITHIN GROUP (ORDER BY val)
  FROM t
 GROUP BY grp
```

grp	val
1	A, B, C
2	X

Default

Default

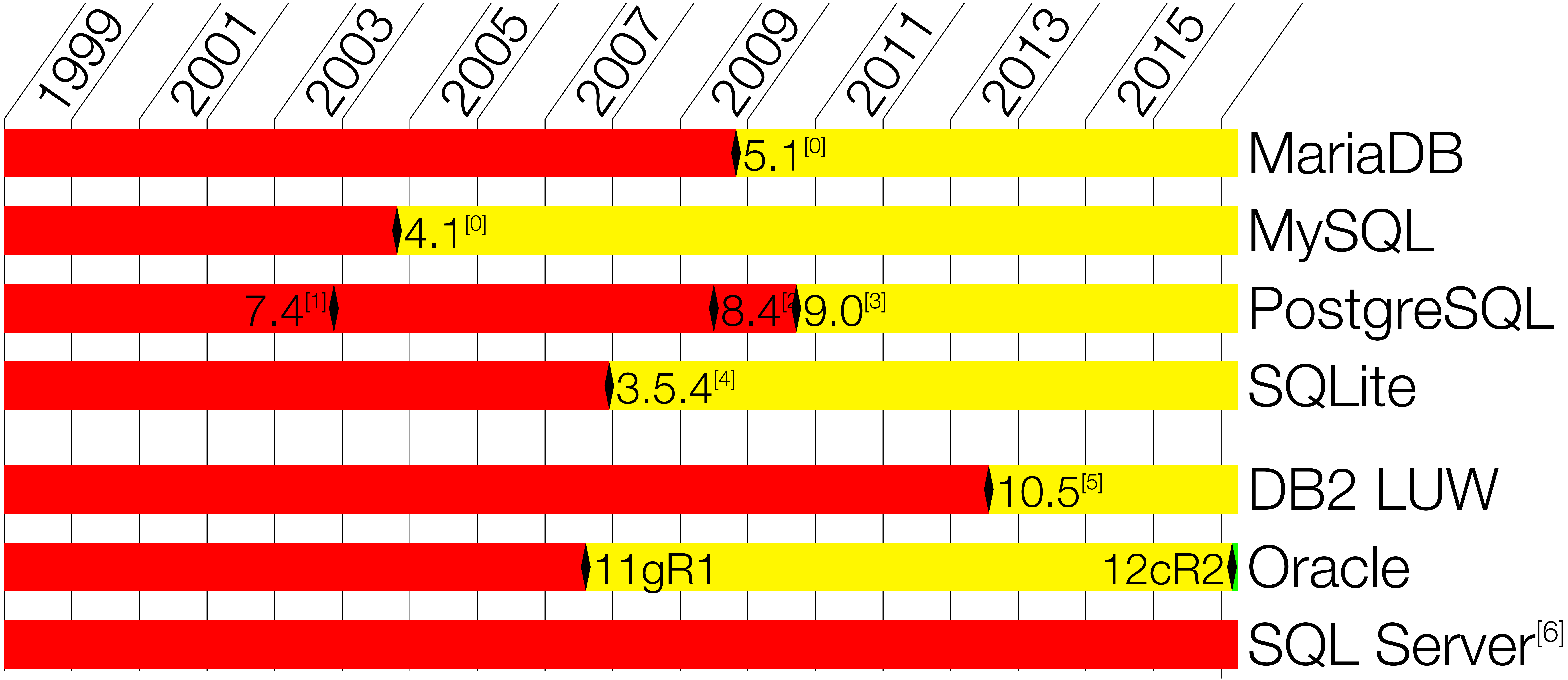
```
LIST_AGG(val, ', ' ON OVERFLOW ERROR)

LIST_AGG(val, ', ' ON OVERFLOW TRUNCATE '...' WITHOUT COUNT) → 'A, B, ...'

LIST_AGG(val, ', ' ON OVERFLOW TRUNCATE '...' WITH COUNT) → 'A, B, ...(1)'
```

LIST_AGG

Availability



[0] group_concat

[1] array_to_string

[2] array_agg

[3] string_agg

[4] group_concat w/o ORDER BY

[5] No ON OVERFLOW clause

[6] string_agg announced for vNext

Also new in SQL:2016

JSON

DATE FORMAT

POLYMORPHIC TABLE FUNCTIONS

About @MarkusWinand

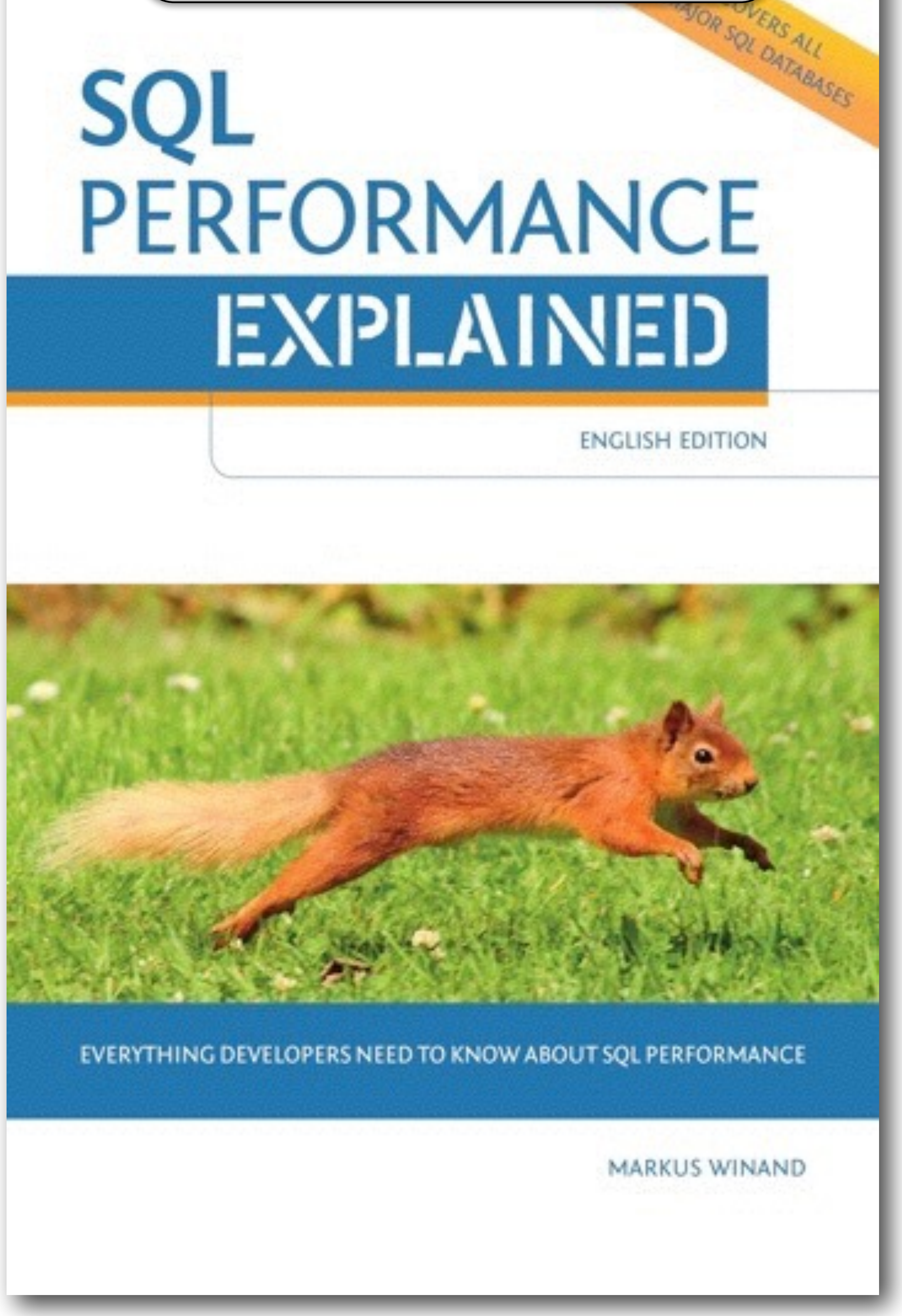
- Training for Developers
  - SQL Performance (Indexing)
  - Modern SQL
  - On-Site or Online
- SQL Tuning
  - Index-Redesign
  - Query Improvements
  - On-Site or Online

http://winand.at/

About @MarkusWinand

€0,-

€10-30



OVERS ALL
AJOR SQL DATABASES

SQL PERFORMANCE EXPLAINED

ENGLISH EDITION

EVERYTHING DEVELOPERS NEED TO KNOW ABOUT SQL PERFORMANCE

MARKUS WINAND

sql-performance-explained.com

About @MarkusWinand

@ModernSQL

http://modern-sql.com