
How PostgreSQL's
SQL dialect stays ahead
of its competitors
@MarkusWinand

I'm sorry!


I couldn't make it
to PgConf.EU in 2014!

Don't repeat my mistake.


Come to
PgConf.EU 2018
in Lisbon!
Oct 23-26.


How PostgreSQL's
SQL dialect stays ahead
of its competitors
@MarkusWinand

PostgreSQL 9.5

2016-01-07

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
 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

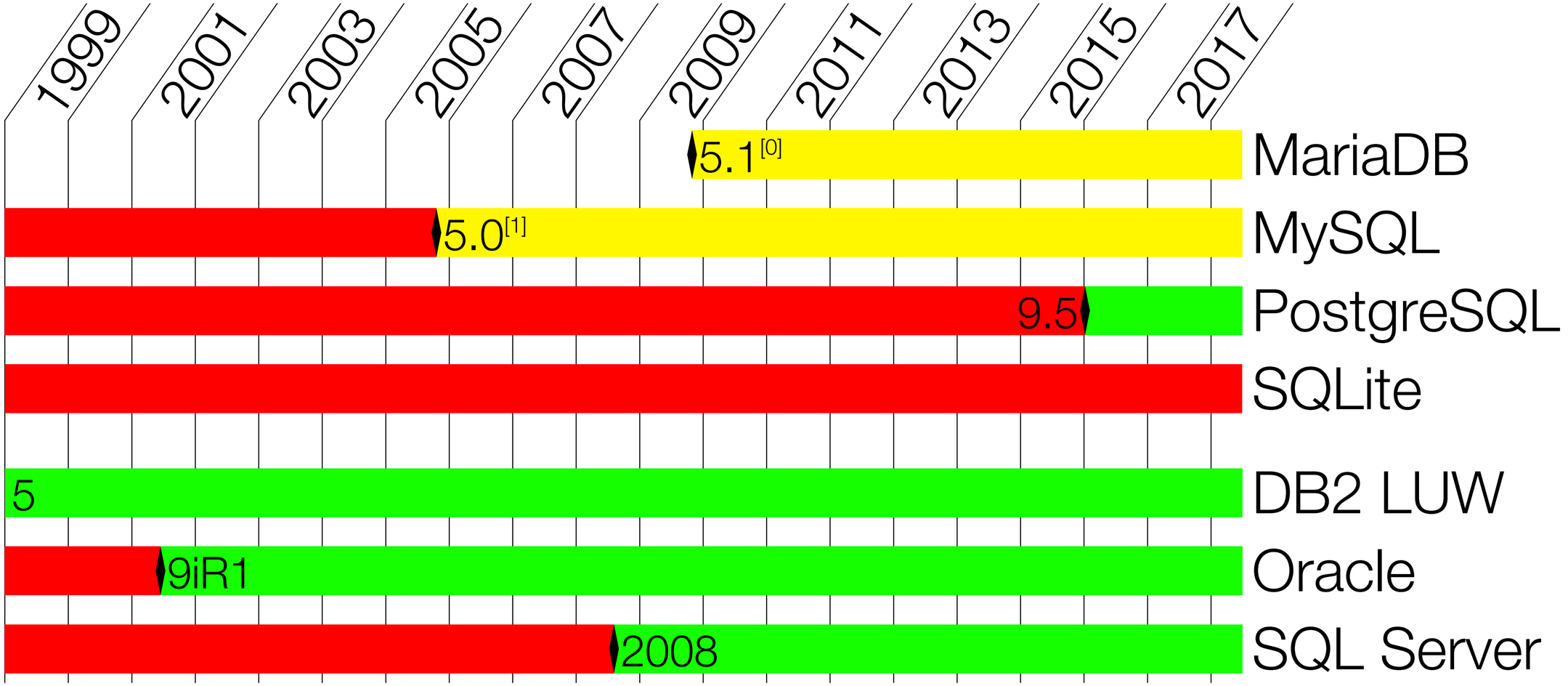
() (empty parenthesis) build a group over all rows

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

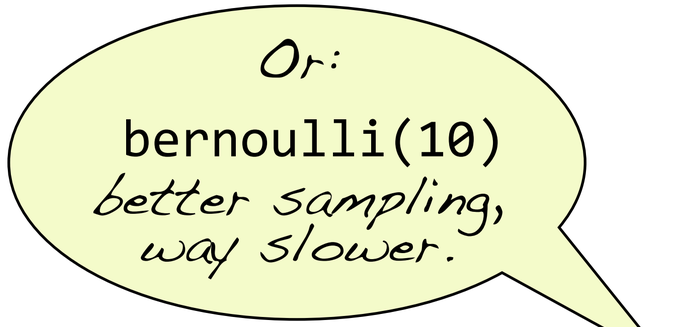


[0]Only ROLLUP (properitery syntax).
[1]Only ROLLUP (properitery syntax). GROUPING function since MySQL 8.0.

TABLESAMPLE

TABLESAMPLE

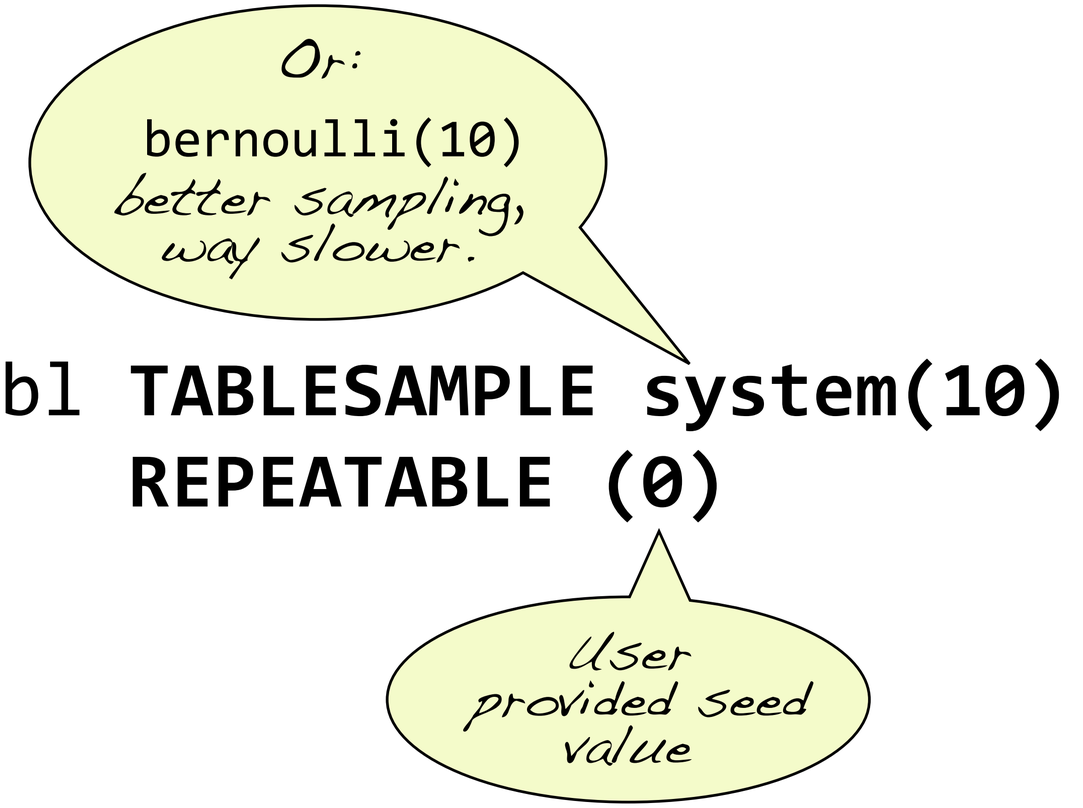
Since SQL:2003




```
SELECT *
  FROM tbl TABLESAMPLE system(10)
```

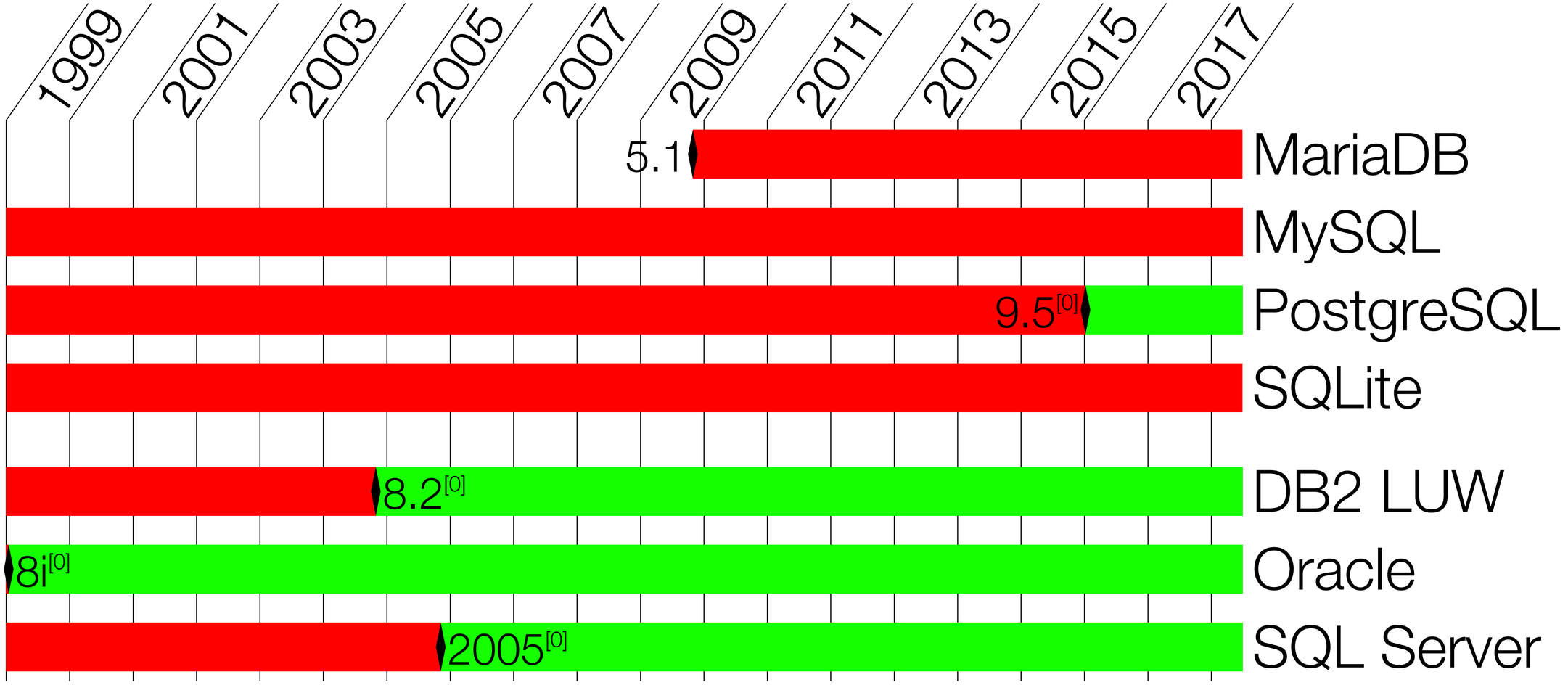
TABLESAMPLE

Since SQL:2003



TABLESAMPLE

Availability



[0]Not for derived tables

PostgreSQL 10

2017-10-05

XMLTABLE

XMLTABLE

Since SQL:2006

```
FROM tbl
   , XMLTABLE(
        '/d/e'
        ) r
```

XPath* expression to identify rows

Stored in tbl.x:

```
<d>
  <e id="42">
    <c1>…</c1>
  </e>
</d>
```

*Standard SQL allows XQuery

XMLTABLE

Since SQL:2006

```
FROM tbl
   , XMLTABLE(
        '/d/e'
        PASSING x




   ) r
```

Stored in tbl.x:

```
<d>
  <e id="42">
    <c1>…</c1>
  </e>
</d>
```

*Standard SQL allows XQuery

XMLTABLE

Since SQL:2006

```
FROM tbl
   , XMLTABLE(
        '/d/e'
        PASSING x
        COLUMNS id INT PATH '@id'
              , c1 VARCHAR(255) PATH 'c1'

     ) r
```

XPath* expressions to extract data

Stored in tbl.x:

```
<d>
  <e id="42">
    <c1>…</c1>
  </e>
</d>
```

*Standard SQL allows XQuery

XMLTABLE

Since SQL:2006

```
FROM tbl
   , XMLTABLE(
        '/d/e'
        PASSING x
        COLUMNS id INT PATH '@id'
              , c1 VARCHAR(255) PATH 'c1'
              , n                    FOR ORDINALITY
     ) r
```

Stored in tbl.x:

```
<d>
  <e id="42">
    <c1>…</c1>
  </e>
</d>
```

Row number (like for unnest)

*Standard SQL allows XQuery

XMLTABLE

Since SQL:2006

```
SELECT id
     , c1
     , n
  FROM tbl
     , XMLTABLE(
         '/d/e'
         PASSING x
         COLUMNS id INT PATH '@id'
               , c1 VARCHAR(255) PATH 'c1'
               , n                FOR ORDINALITY
       ) r
```

Result

```
 id | c1 | n
----+----+---
 42 | …  | 1
```

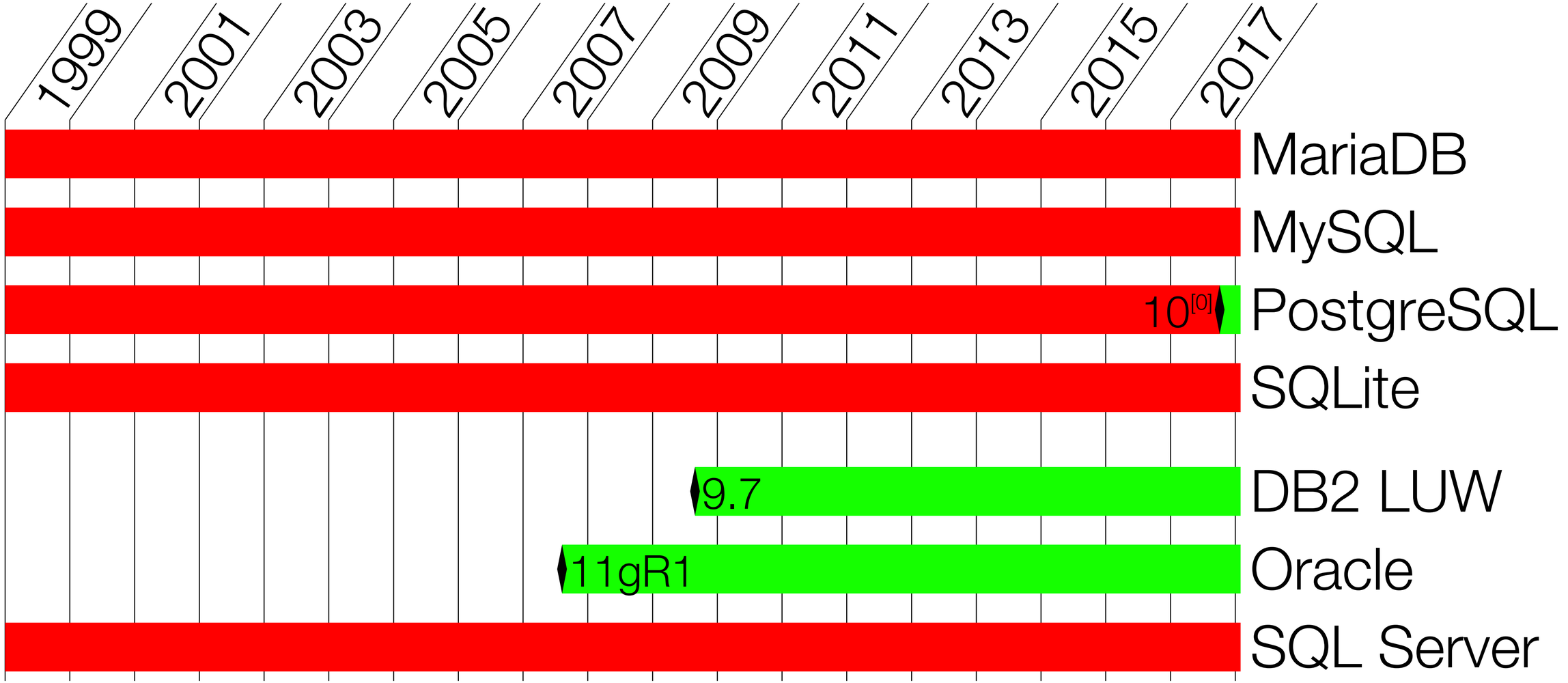
Stored in tbl.x:

```
<d>
  <e id="42">
    <c1>…</c1>
  </e>
</d>
```

*Standard SQL allows XQuery

XMLTABLE

Availability



[0]No XQuery (only XPath). No default namespace declaration.

IDENTITY COLUMNS

`IDENTITY` column

Since SQL:2003

Similar to **`serial`** columns, but standard and more powerful.

```
CREATE TABLE tbl (
    id INTEGER GENERATED BY DEFAULT AS IDENTITY


 , PRIMARY KEY (id)
)
```

`IDENTITY` column

Since SQL:2003

Similar to **`serial`** columns, but standard and more powerful.

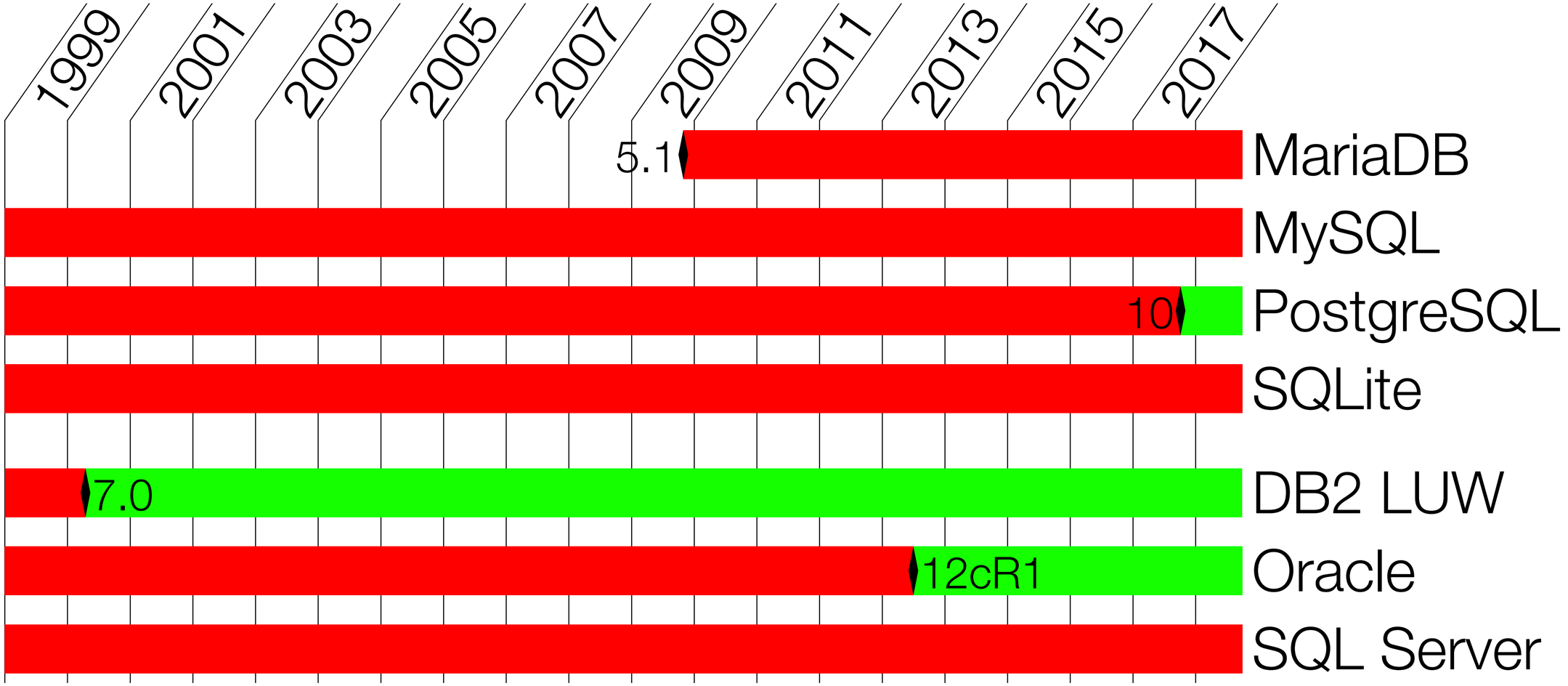
```
CREATE TABLE tbl (
    id INTEGER GENERATED BY DEFAULT AS IDENTITY
               (START WITH 10 INCREMENT BY 10)

  , PRIMARY KEY (id)
)
```

More (esp. vs. serial):
https://blog.2ndquadrant.com/postgresql-10-identity-columns/

`IDENTITY` column

Availability



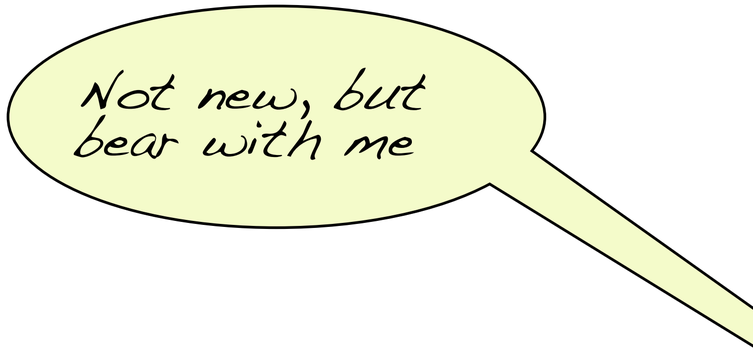
PostgreSQL 11

201?-??-??

OVER

and

PARTITION BY



OVER (PARTITION BY)

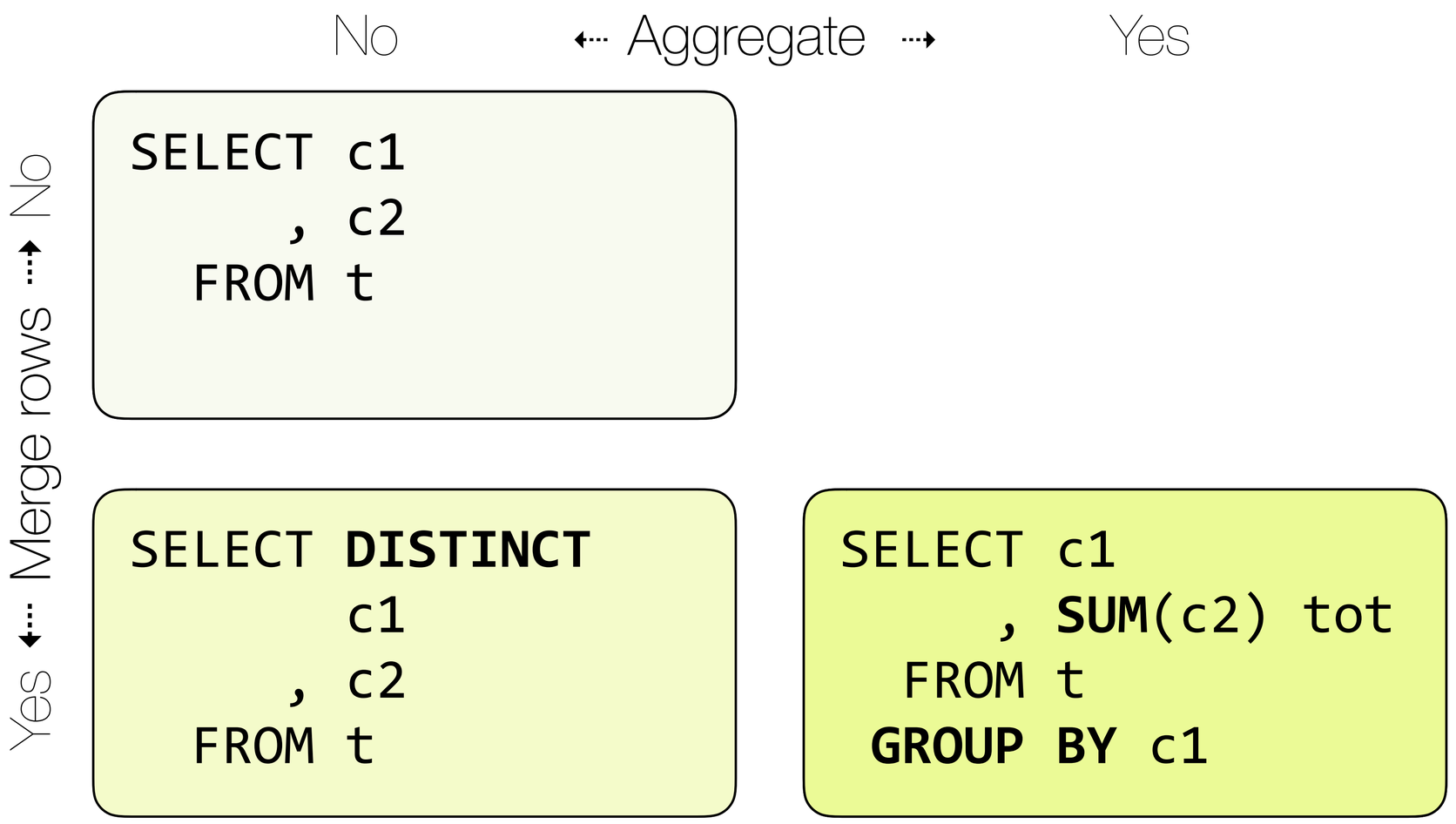
The Problem

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

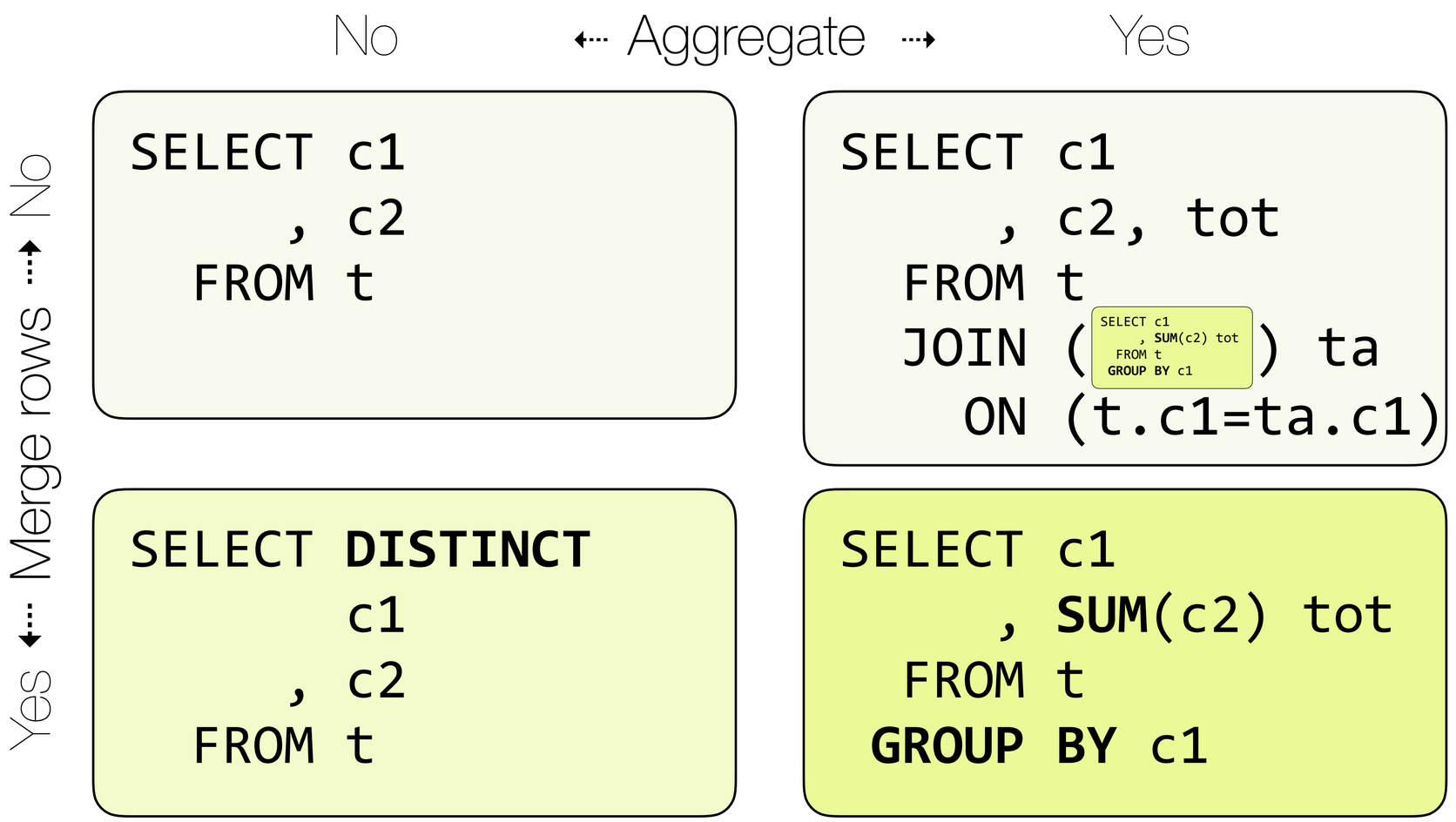
OVER (PARTITION BY)

The Problem



OVER (PARTITION BY)

The Problem



OVER (PARTITION BY)

The Problem

No ⇠ Aggregate ⇢ Yes

Yes ⇣ Merge rows ⇡ No

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

```
SELECT
  F
  J                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

No ⇠ Aggregate ⇢ Yes

Yes ⇣ Merge rows ⇡ No

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

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

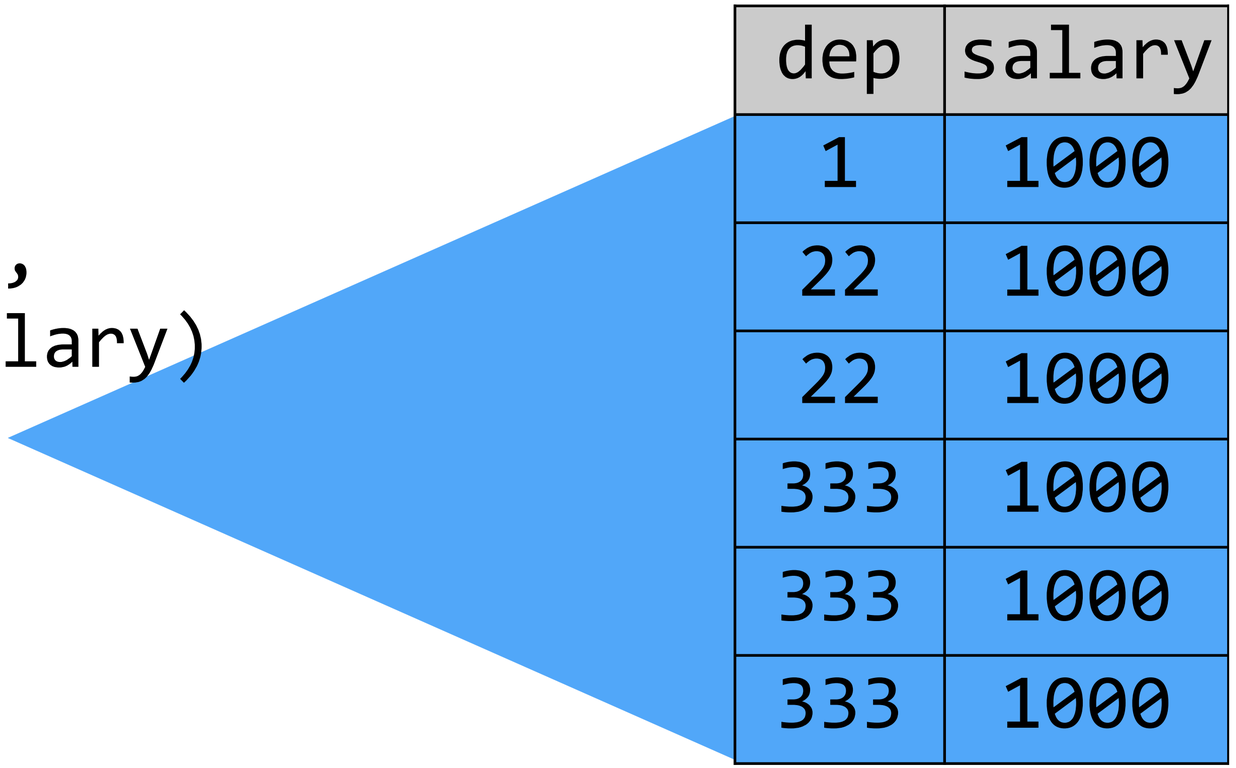
```
SELECT DISTINCT
       c1
     , c2
  FROM t
```

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

OVER (PARTITION BY)

How it works

```
SELECT dep,
       salary,
       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
```

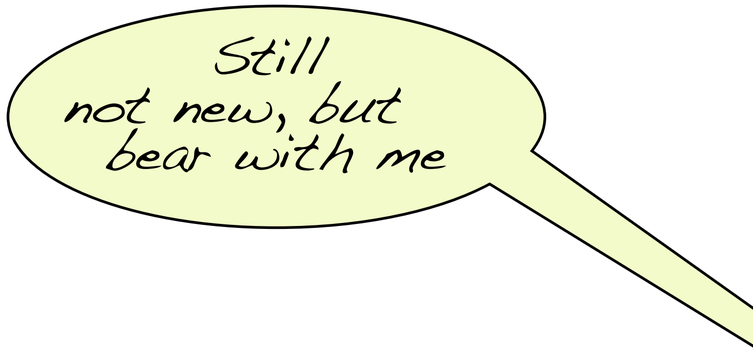
dep	salary	ts
1	1000	1000
22	1000	2000
22	1000	2000
333	1000	3000
333	1000	3000
333	1000	3000

OVER

and

ORDER BY

(Framing & Ranking)



OVER (ORDER BY)

The Problem

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

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

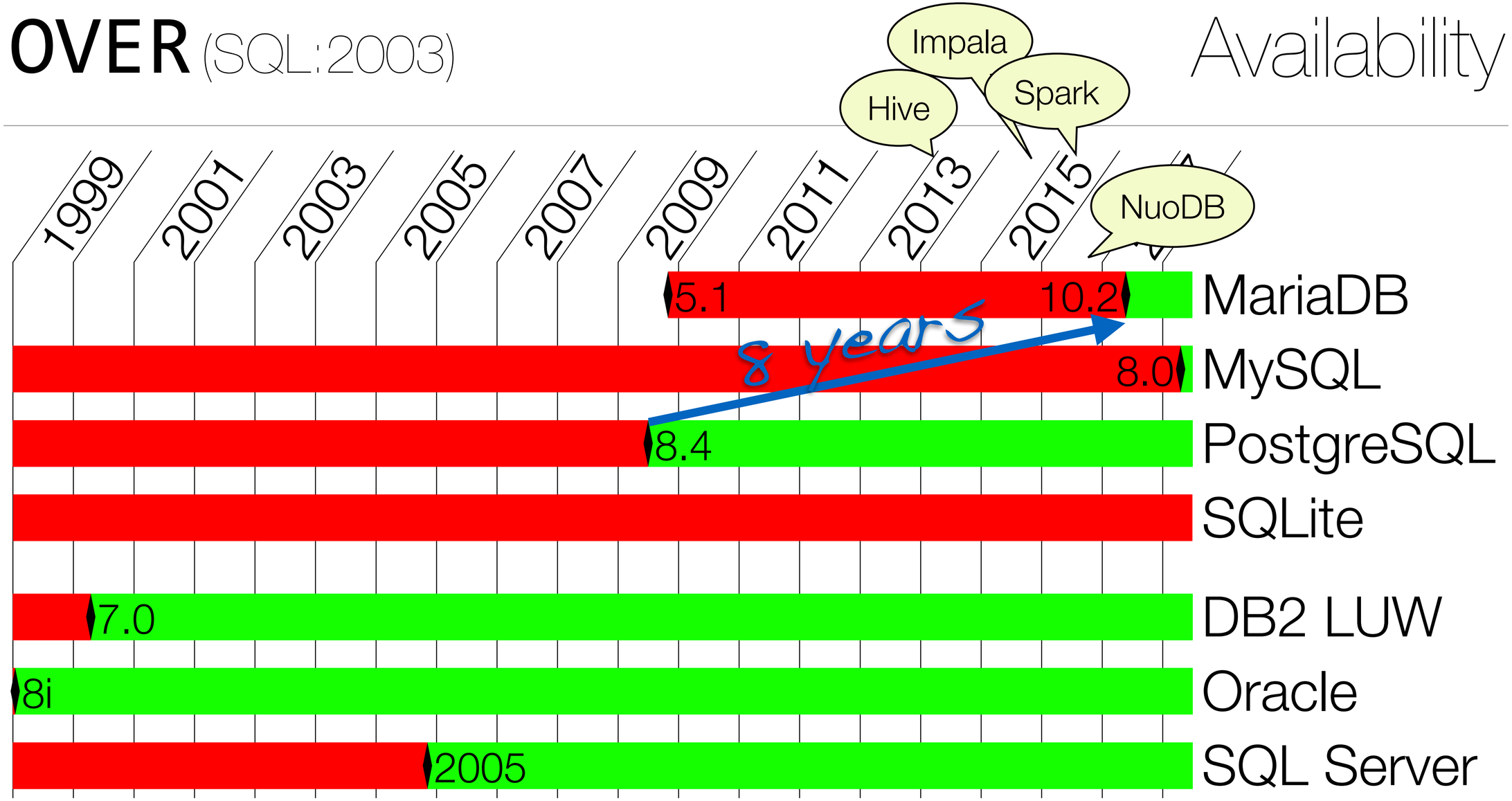
OVER() makes all rows visible at every row

OVER(PARTITION BY ...) segregates like **GROUP BY**

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

OVER (SQL:2003)

Availability



OVER

SQL:2011 functions

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)
```

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 Since SQL:2011

SQL:2011 can access other rows directly:

```
SELECT *, balance - LAG(balance, 1, 0)
                    OVER(ORDER BY x)
  FROM data
```

OVER Since SQL:2011

SQL:2011 can access other rows directly:

```
SELECT *, balance - LAG(balance, 1, 0)
                    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:2011

SQL:2011 can access other rows directly:

```
SELECT *, balance - LAG(balance, 1, 0)
                    OVER(ORDER BY x)
  FROM data
```

Available functions:

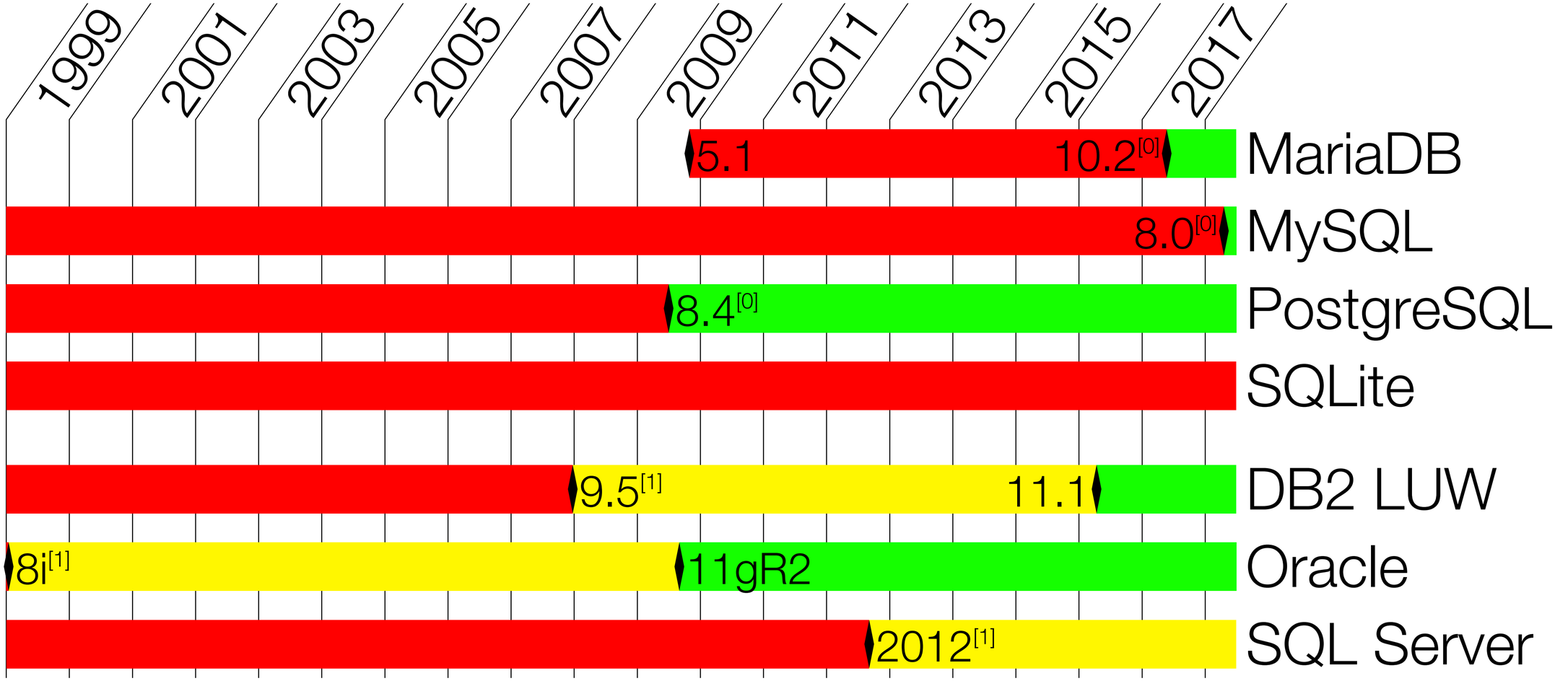
```
LEAD / LAG
FIRST_VALUE / LAST_VALUE
NTH_VALUE(col, n)
```

Not supported by PostgreSQL (as of 11)

```
FROM FIRST/LAST
RESPECT/IGNORE NULLS
```

OVER (LEAD, LAG, ...)

Since SQL:2011



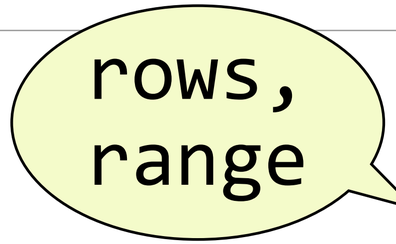
[0]No `IGNORE NULLS` and `FROM LAST`
[1]No `NTH_VALUE`

OVER

SQL:2011 **groups** option

OVER (groups option)

Since SQL:2011



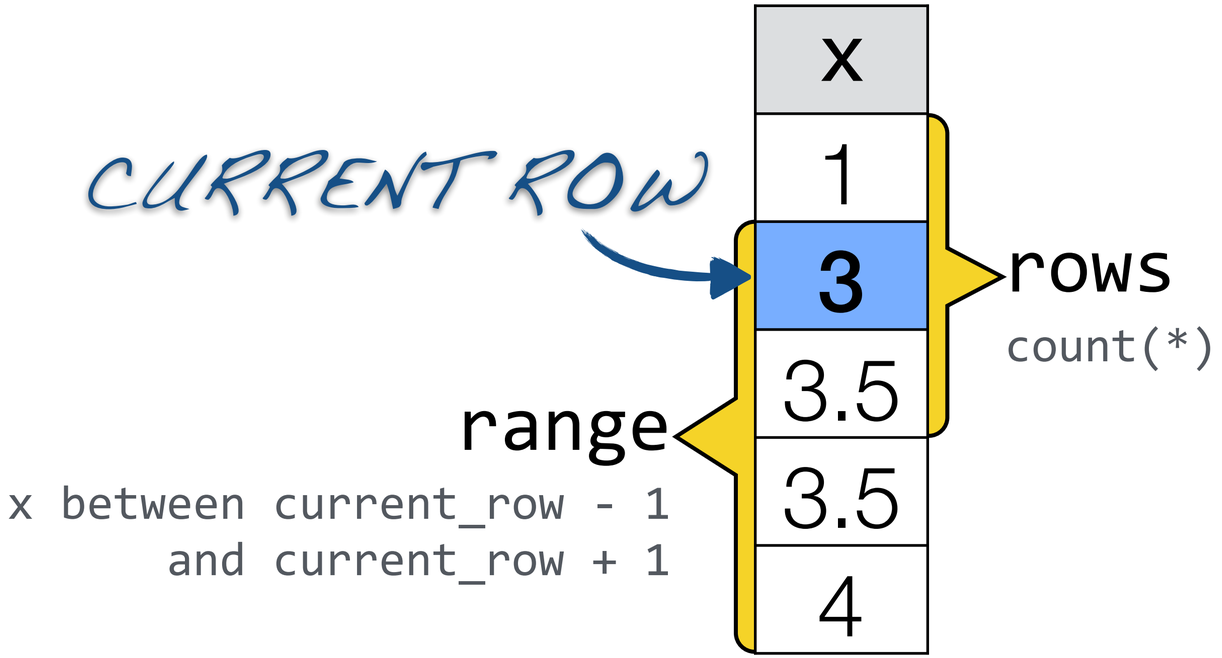

```
ORDER BY x
<frame unit> between 1 preceding
                 and 1 following
```

OVER (groups option)

Since SQL:2011

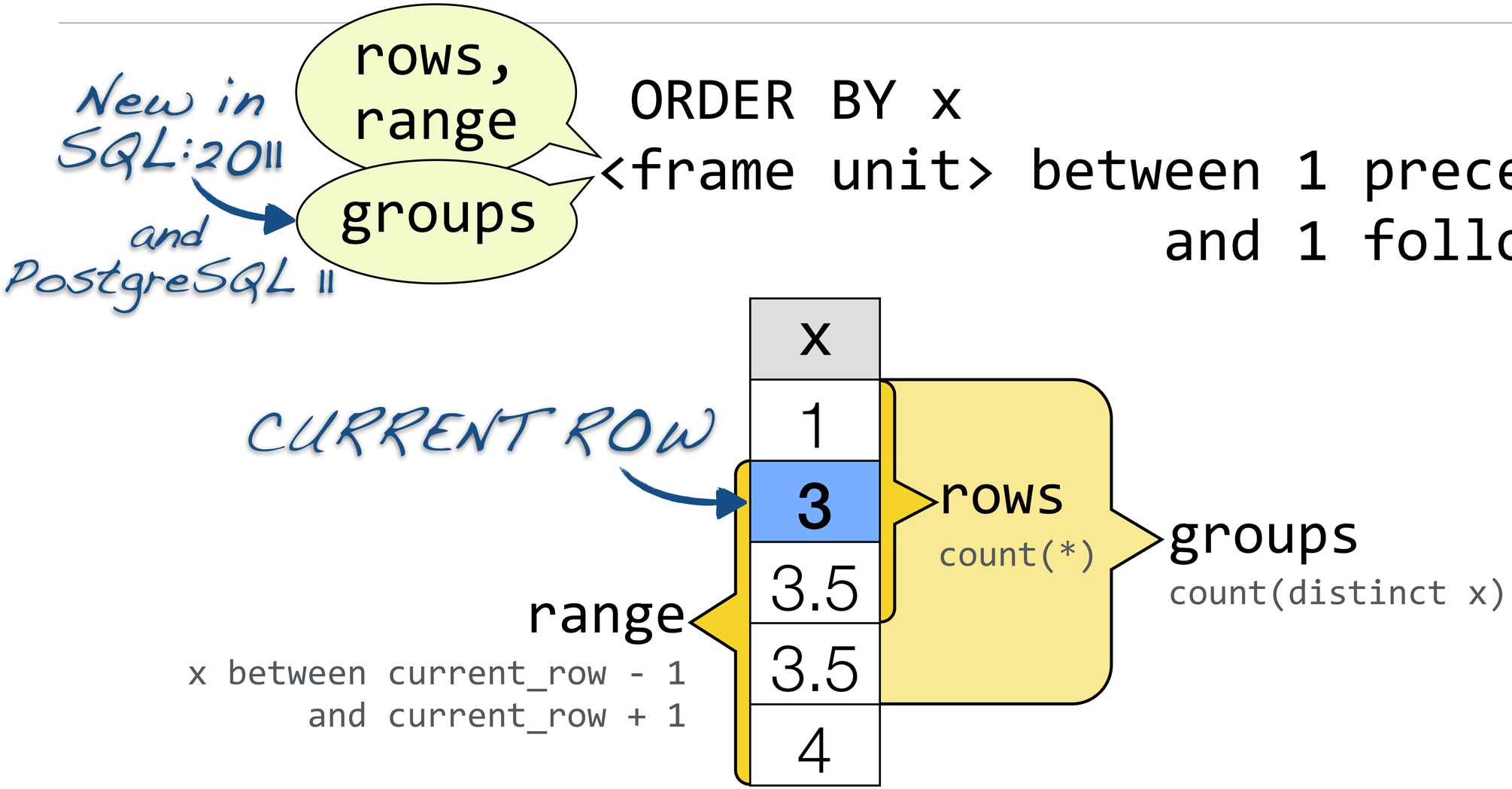
rows,
range

```
ORDER BY x
<frame unit> between 1 preceding
                 and 1 following
```



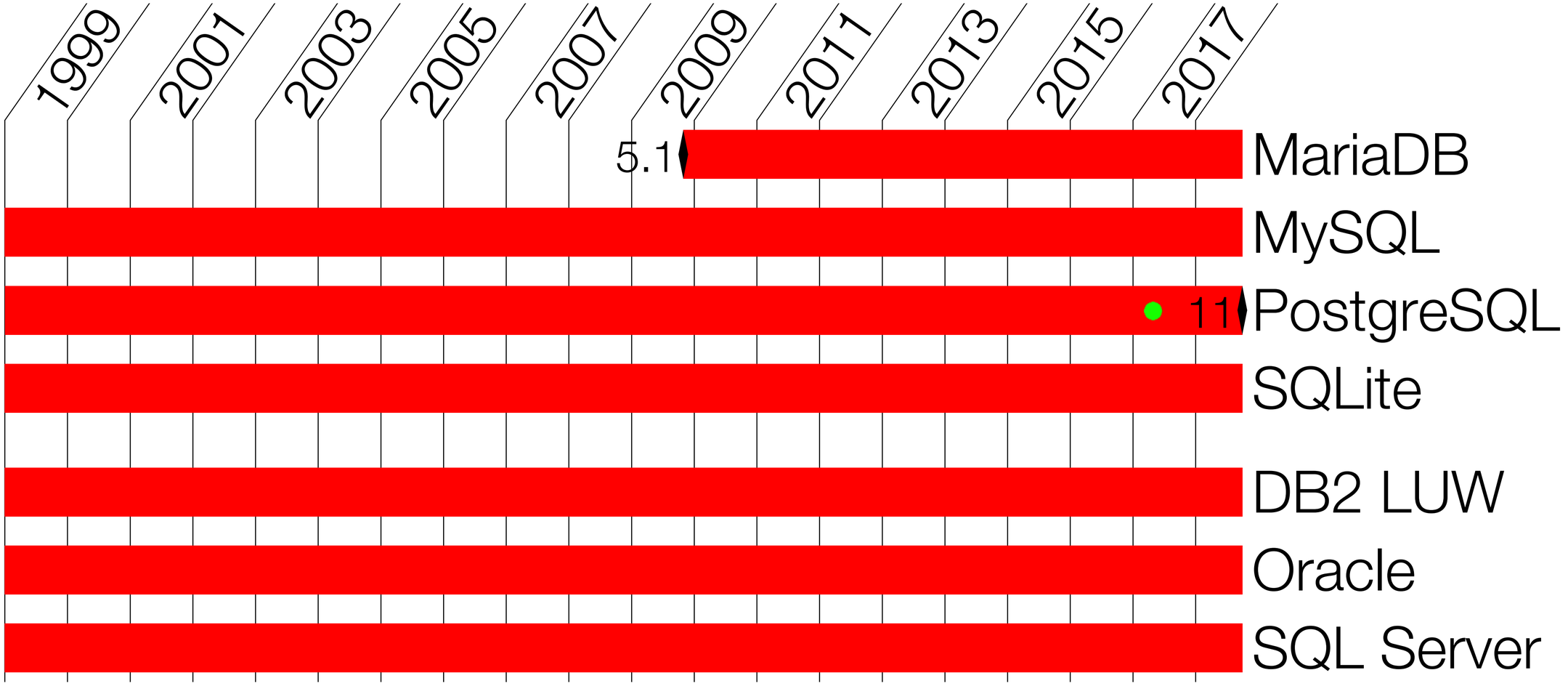
OVER (groups option)

Since SQL:2011



OVER (groups option)

Since SQL:2011



OVER

SQL:2003 frame exclusion

OVER (frame exclusion)

Since SQL:2003

```
OVER (ORDER BY
        … BETWEEN …
        exclude [ no others | current row
                | group | ties ] )
```

default

New in PostgreSQL 11

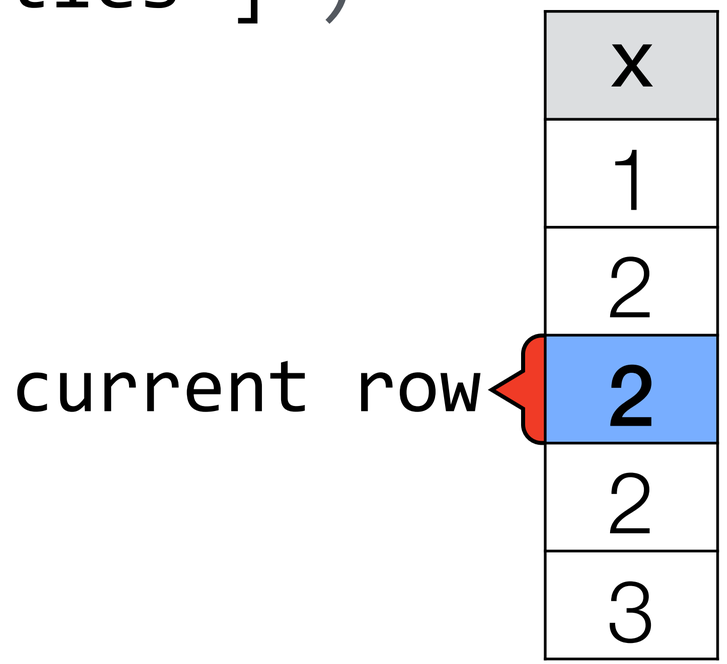
OVER (frame exclusion)

Since SQL:2003

```
OVER (ORDER BY
        … BETWEEN …
        exclude [ no others | current row
                | group | ties ] )
```

default

New in PostgreSQL 11



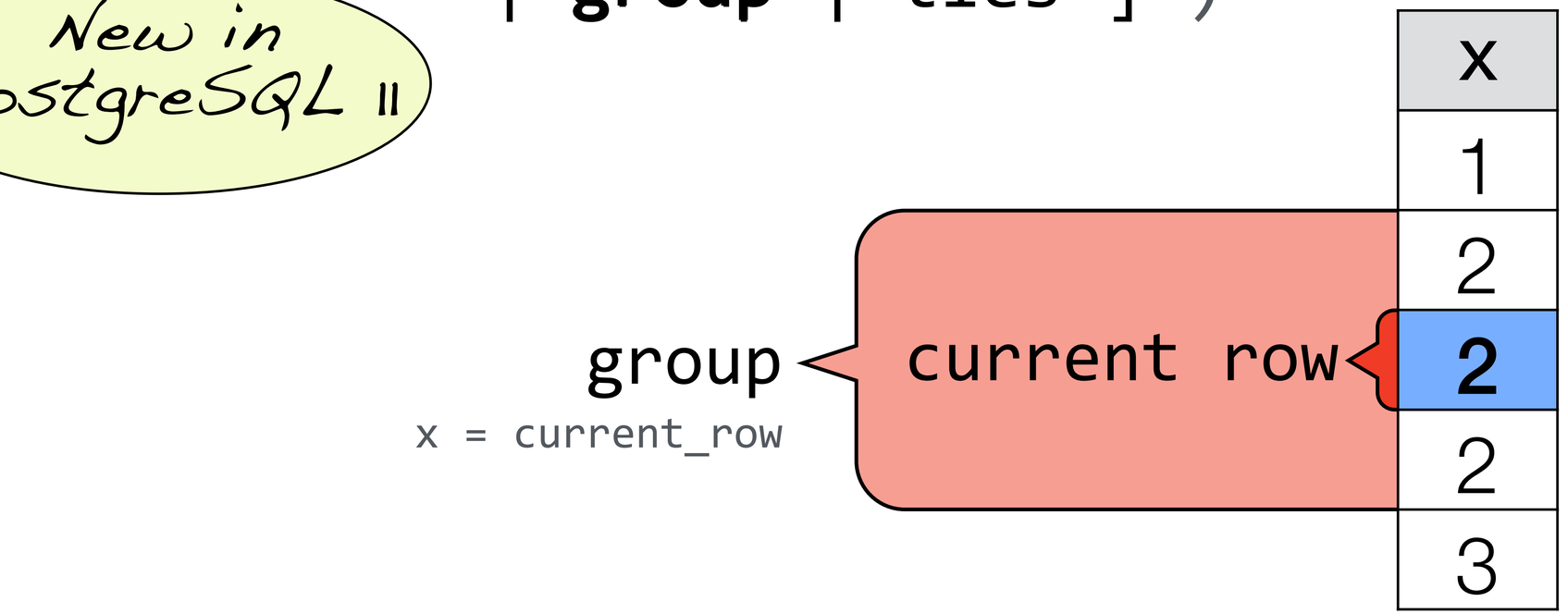
OVER (frame exclusion)

Since SQL:2003

```
OVER (ORDER BY
        … BETWEEN …
        exclude [ no others | current row
                | group | ties ] )
```

default

New in PostgreSQL 11



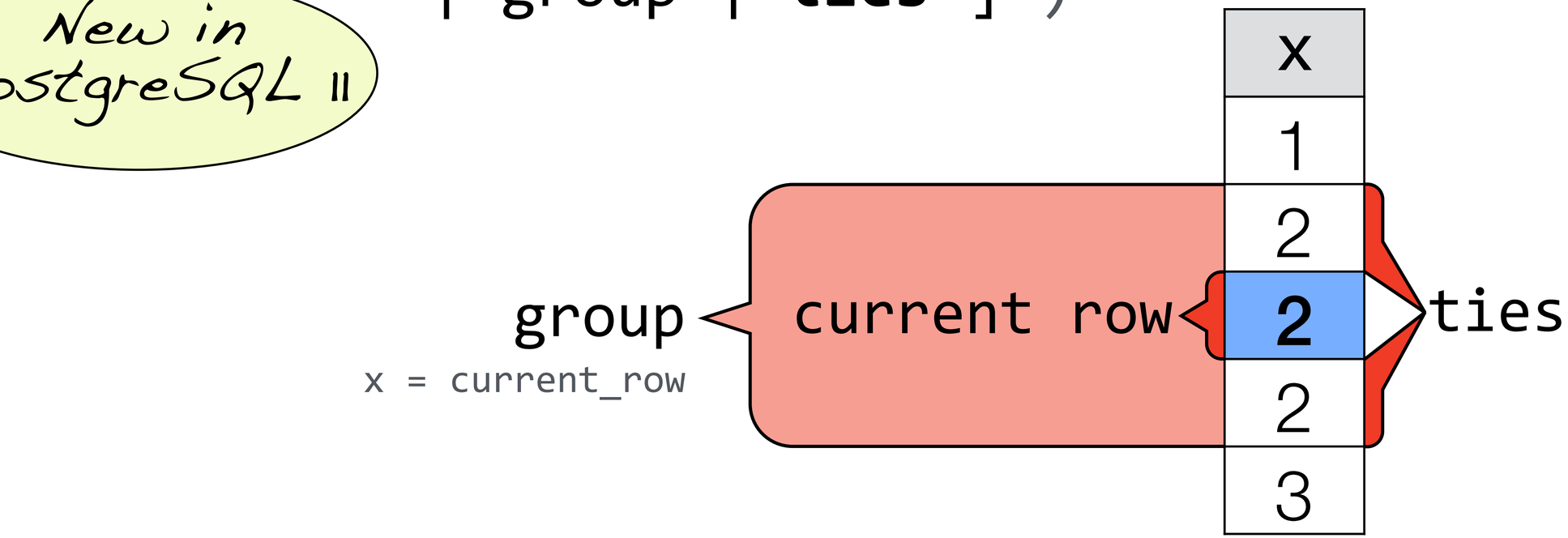
OVER (frame exclusion)

Since SQL:2003

```
OVER (ORDER BY
        … BETWEEN …
        exclude [ no others | current row
                | group | ties ] )
```

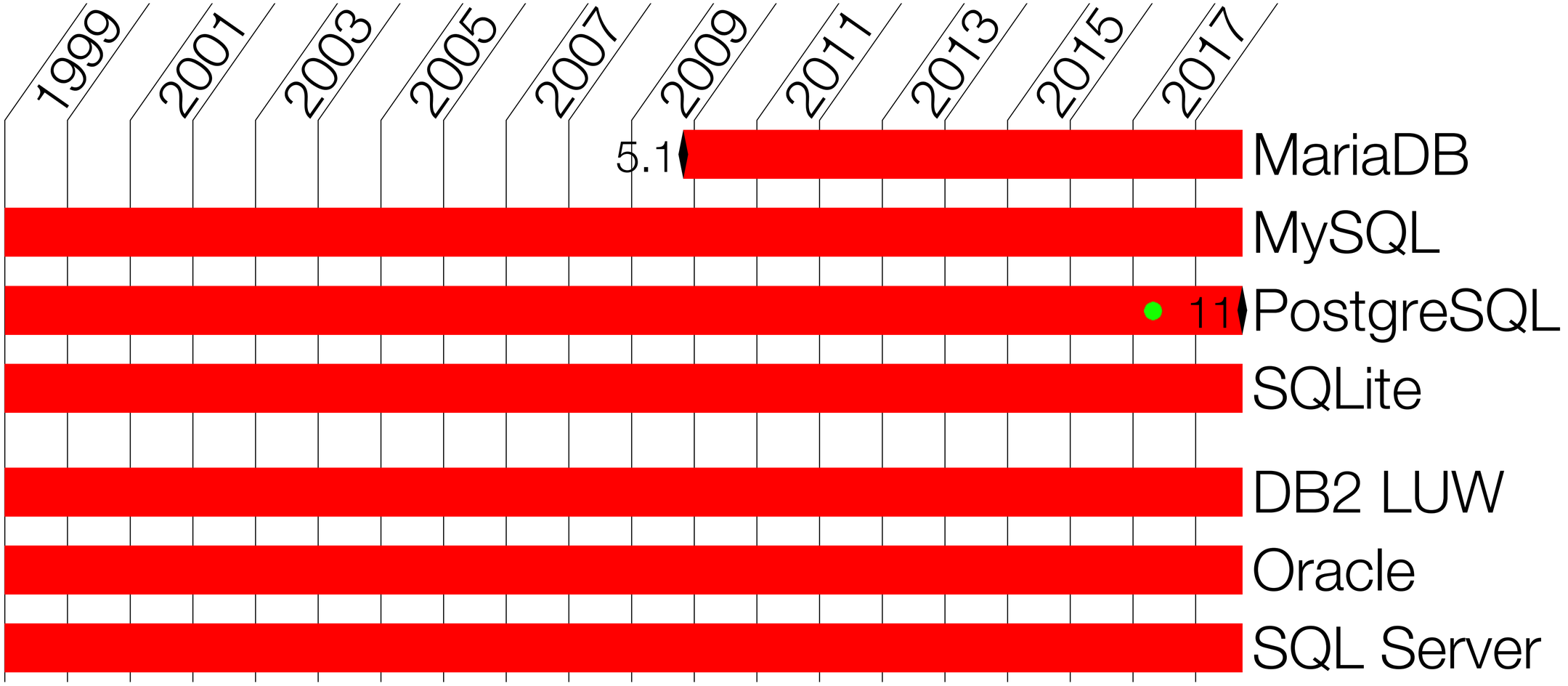
default

New in PostgreSQL 11



OVER (frame exclusion)

Since SQL:2011



PostgreSQL 12+

20??-??-??

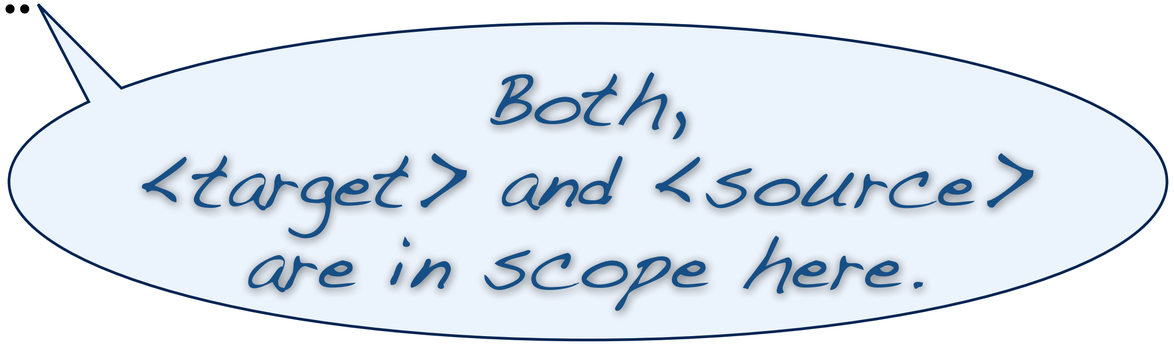
MERGE

MERGE

The Problem

Copying rows from another table is easy:

```
INSERT INTO <target>
SELECT …
  FROM <source>
 WHERE NOT EXISTS (SELECT *
                     FROM <target>
                    WHERE …
                   )
```



MERGE

The Problem

Deleting rows that exist in another table is also possible:

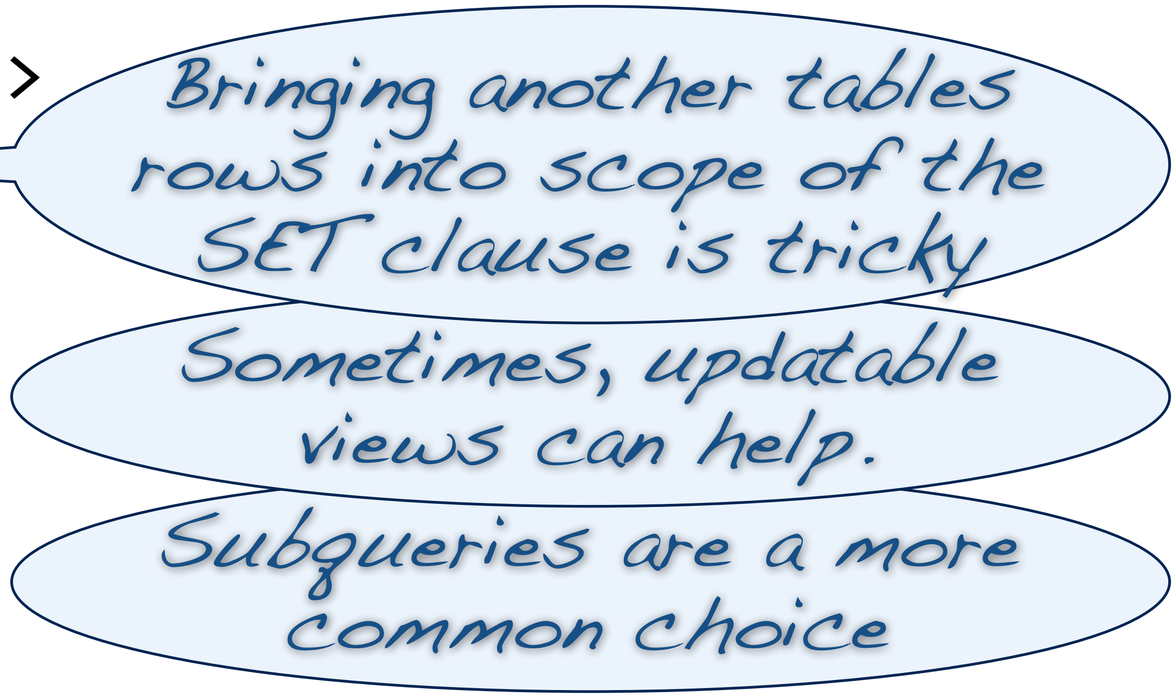
```
DELETE FROM <target>
 WHERE EXISTS (SELECT *
                 FROM <source>
                WHERE …
               )
```

MERGE

The Problem

Updating rows from another table is awkward:

```
UPDATE <target>
   SET …
 WHERE …
```

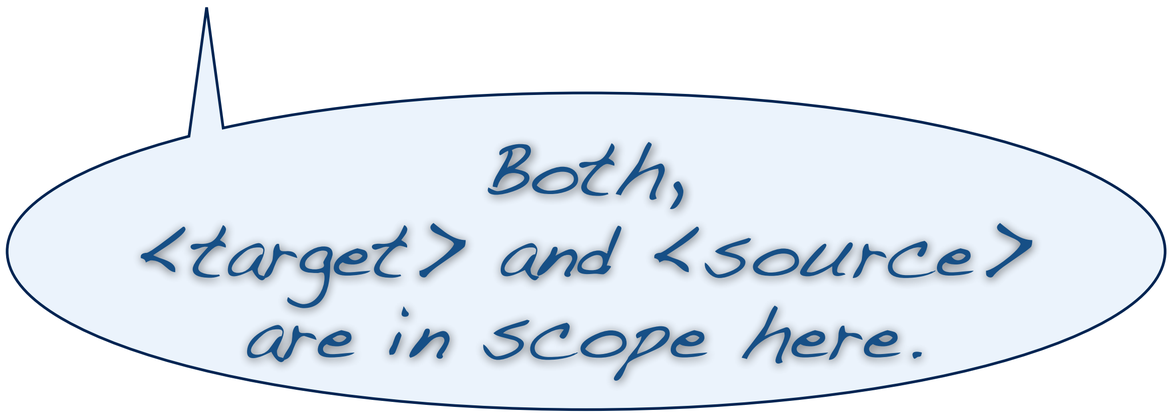


MERGE

The Problem

Updating rows from another table is awkward:

```
UPDATE <target>
   SET (col1, col2) = (SELECT col1, col2
                         FROM <source>
                        WHERE …
                       )
 WHERE …
```



MERGE

Since SQL:2008

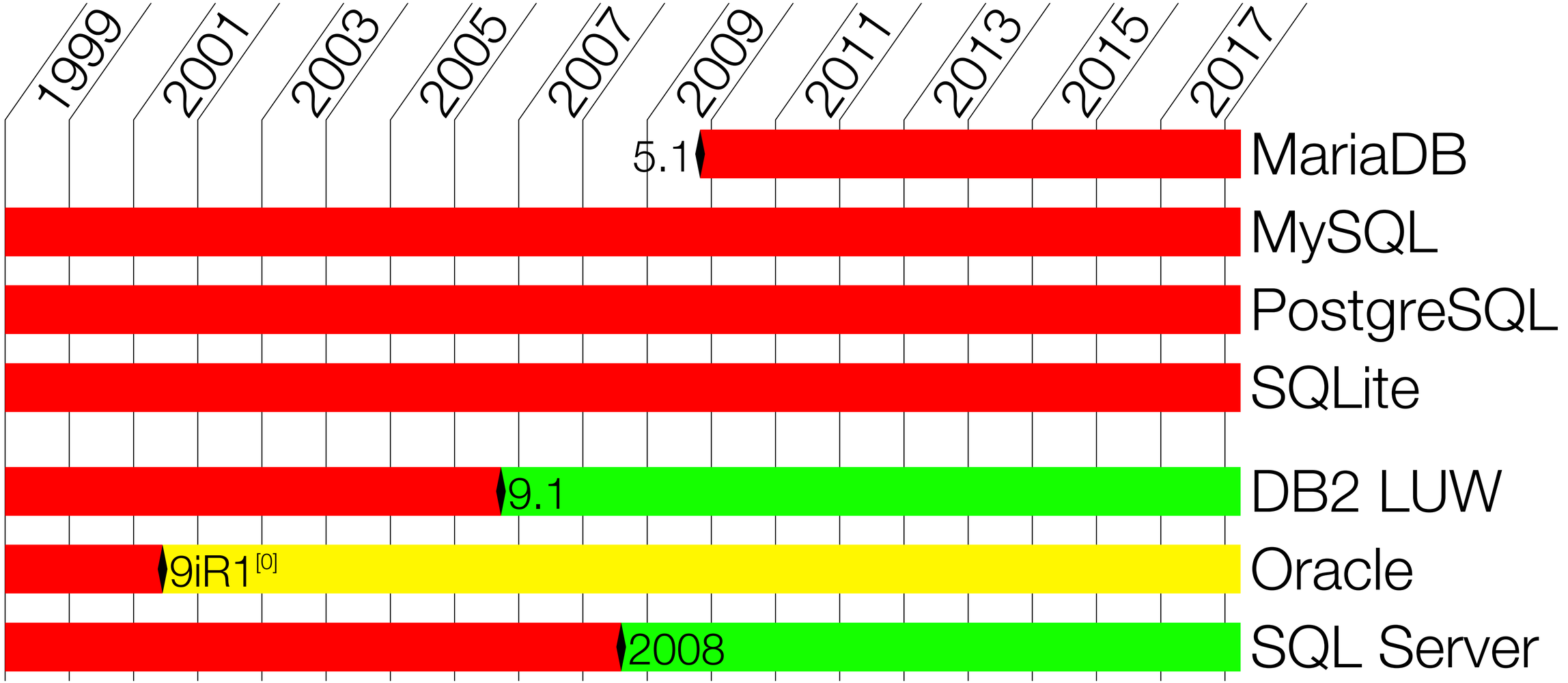
SQL:2008 introduced **`merge`** to improve this situation two-fold:

- It has always two tables in scope,
  the source can be a derived table (subquery).
- It can do **`insert`**, **`update`**, or **`delete`** in one go.


```
MERGE INTO <target>
USING <source>
   ON <join condition>
 WHEN MATCHED [AND <cond>] THEN [UPDATE…|DELETE…]
 WHEN NOT MATCHED [AND <cond>] THEN INSERT…
```

MERGE

Since SQL:2008



[0]No AND condition.

(standard) **JSON**

JSON

PostgreSQL got the ***JSON data type*** and
the first JSON related functions
with 9.2 (2012)

ISO added SQL JSON functions (but ***no type***)
with SQL:2016

Both approaches follow a different mantra.
The following slides only reflects the standards perspective.

JSON

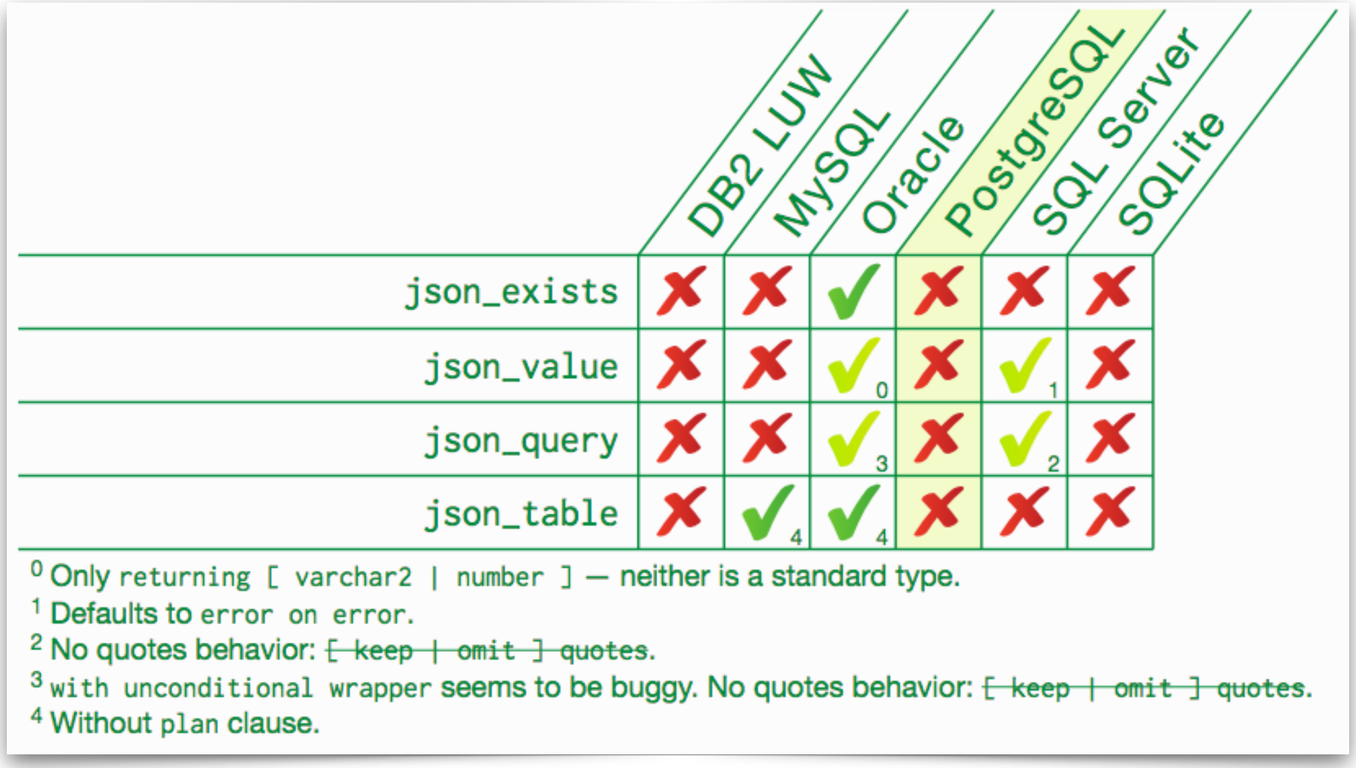
	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`is [not] json predicate`	✗	✗	✓[0]	✗	✗	✗
`on [ error | empty ] clauses`	✗	✗	✓[1]	✗	✗	✗

[0] No type constraints: `is json [ value | array | object | scalar ]`. Also `unique keys` (T822).
[1] No `unknown on error`. No expressions in `default ... on [ error | empty ]`

	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`json_object`	✗	✗	✓[0]	✗	✗	✗
`json_array`	✗	✓[1]	✓[2]	✗	✗	✓[1]
`json_objectagg`	✗	✗[3]	✓[4]	✗	✗	✗
`json_arrayagg(… order by …)`	✗	✗	✓[5]	✗	✗	✗

[0] No colon syntax (T814). No key uniqueness constraint (T830): `[with|without] unique [keys]`.
[1] Defaults to `absent on null`. No construction by query: `json_array(select …)`.
[2] No construction by query: `json_array(select …)`.
[3] Supports comma (,) instead of `values` or colon (:).
[4] No colon syntax (T814). No key uniqueness constraint (T830): `[with|without] unique [keys]`. Sup
[5] `Absent on null` is buggy.

JSON



	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
json_exists	✗	✗	✓	✗	✗	✗
json_value	✗	✗	✓[0]	✗	✓[1]	✗
json_query	✗	✗	✓[3]	✗	✓[2]	✗
json_table	✗	✓[4]	✓[4]	✗	✗	✗

[0] Only `returning [ varchar2 | number ]` — neither is a standard type.
[1] Defaults to `error on error`.
[2] No quotes behavior: ~~`[ keep | omit ] quotes`~~.
[3] `with unconditional wrapper` seems to be buggy. No quotes behavior: ~~`[ keep | omit ] quotes`~~.
[4] Without `plan` clause.

JSON

	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
JSON path: `lax` mode (default)	✗	✓[1]	✓	✗	✓[0]	✓[1]
JSON path: strict mode	✗	✗	✗[2]	✗	✓	✗
JSON path: item method	✗	✗	✓[3]	✗	✗	✗
JSON path: multiple subscripts	✗	✗	✓	✗	✗	✗
JSON path: `.*` member accessor	✗	✓	✓	✗	✗	✗
JSON path: filter expressions	✗	✗	✓[4]	✗	✗	✗
JSON path: `starts with`	✗	✗	✓[4]	✗	✗	✗
JSON path: `like_regex`	✗	✗	✗	✗	✗	✗

[0] `Lax` mode does not unwrap arrays.
[1] Keyword `lax` not accepted (only default mode). `Lax` mode does not unwrap arrays.
[2] Keyword `strict` is accepted but not honored.
[3] Only in filters. Not supporting `size()`, `datetime()`, `keyvalue()`. `type()` returns `null` for arrays.
[4] Not in `json_value`. Not in `json_query`. Not in `json_table`. Only as last step of expression.
[5] Not in `json_query`. Only as last step of expression.

JSON — Preliminary testing of patches

Used a06e56b24 as basis, applied those patches on top:

0001-strict-do_to_timestamp-v16.patch
0002-pass-cstring-to-do_to_timestamp-v16.patch
0003-add-to_datetime-v16.patch
0004-jsonpath-v16.patch
0005-jsonpath-gin-v16.patch
0006-jsonpath-json-v16.patch
0007-remove-PG_TRY-in-jsonpath-arithmetics-v16.patch
0008-jsonpath-extensions-v16.patch
0009-jsonpath-extensions-tests-for-json-v16.patch

} SQL/JSON: jsonpath

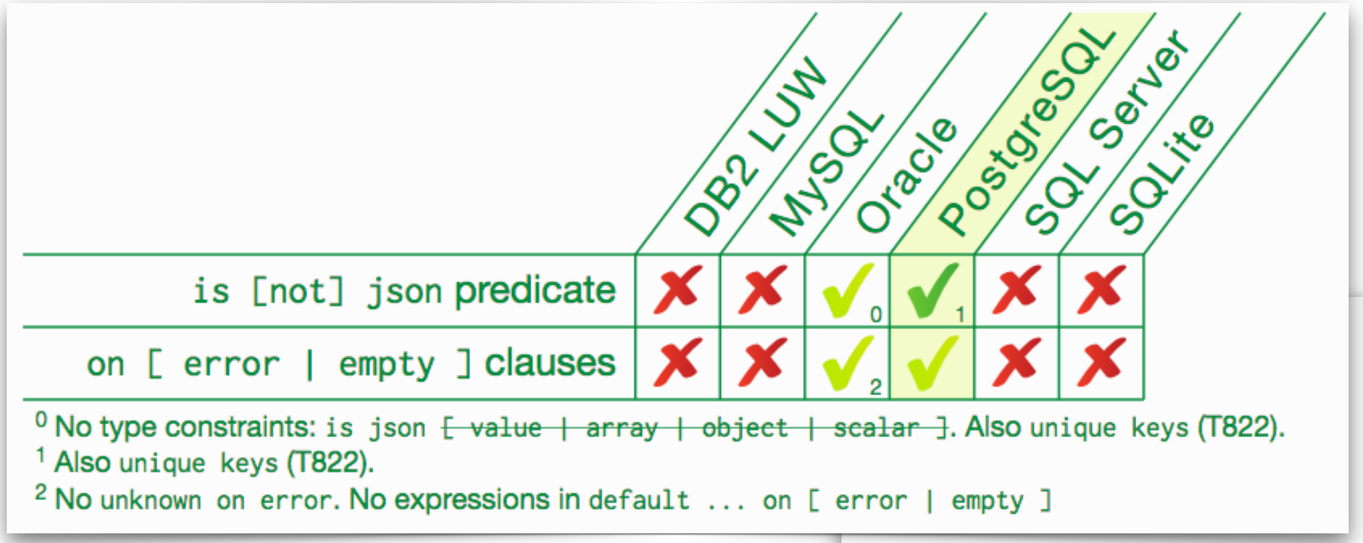
0010-add-invisible-coercion-form-v16.patch
0011-add-function-formats-v16.patch
0012-sqljson-v16.patch
0013-sqljson-json-v16.patch
0014-optimize-sqljson-subtransactions-v16.patch

} SQL/JSON: functions

0015-json_table-v16.patch
0016-json_table-json-v16.patch

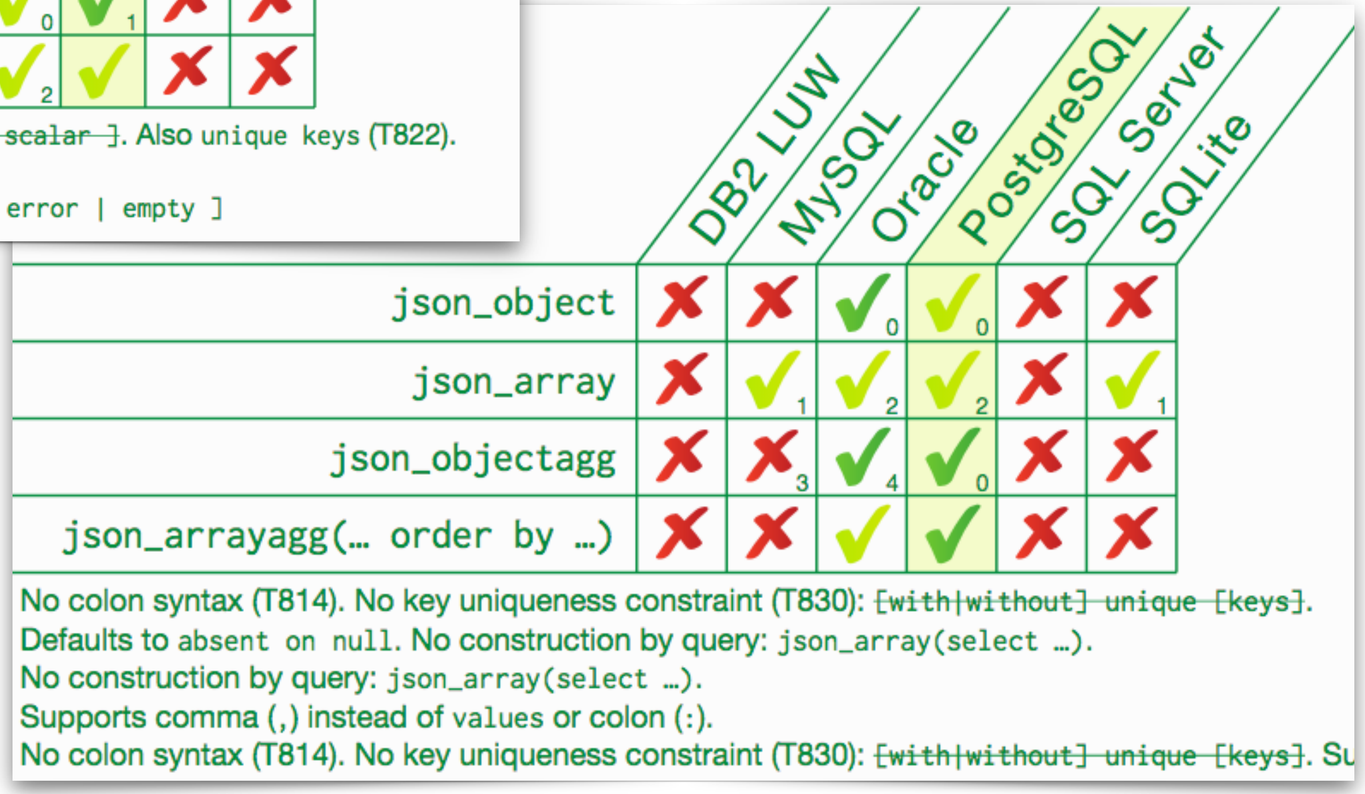
} SQL/JSON: JSON_TABLE

JSON — Preliminary testing of patches



	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`is [not] json` predicate	✗	✗	✓[0]	✓[1]	✗	✗
`on [ error | empty ]` clauses	✗	✗	✓[2]	✓	✗	✗

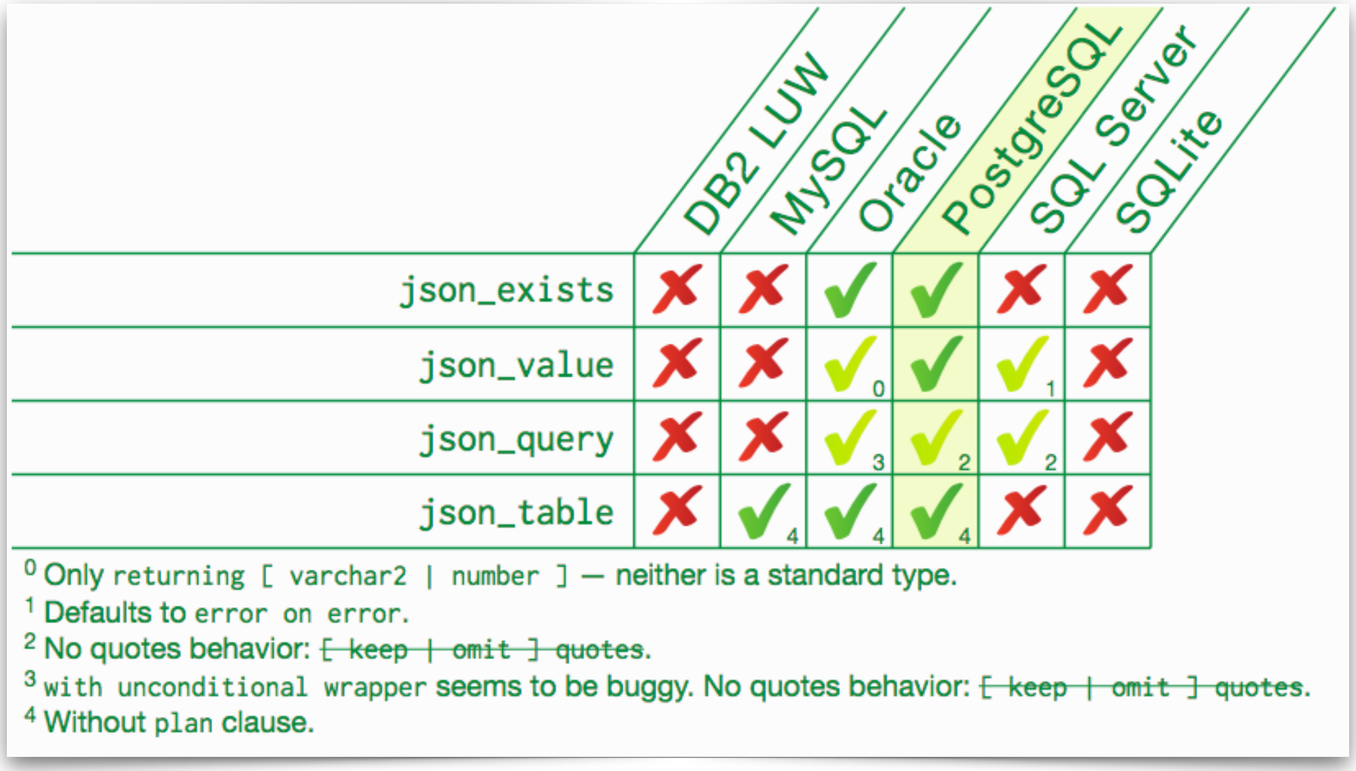
[0] No type constraints: `is json ~~[ value | array | object | scalar ]~~`. Also `unique keys` (T822).
[1] Also `unique keys` (T822).
[2] No `unknown on error`. No expressions in `default ... on [ error | empty ]`



	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`json_object`	✗	✗	✓[0]	✓[0]	✗	✗
`json_array`	✗	✓[1]	✓[2]	✓[2]	✗	✓[1]
`json_objectagg`	✗	✗[3]	✓[4]	✓[0]	✗	✗
`json_arrayagg(… order by …)`	✗	✗	✓	✓	✗	✗

No colon syntax (T814). No key uniqueness constraint (T830): `~~[with|without] unique [keys]~~`.
Defaults to `absent on null`. No construction by query: `json_array(select …)`.
No construction by query: `json_array(select …)`.
Supports comma (,) instead of `values` or colon (:).
No colon syntax (T814). No key uniqueness constraint (T830): `~~[with|without] unique [keys]~~`. Su

JSON — Preliminary testing of patches



	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
json_exists	✗	✗	✓	✓	✗	✗
json_value	✗	✗	✓[0]	✓	✓[1]	✗
json_query	✗	✗	✓[3]	✓[2]	✓[2]	✗
json_table	✗	✓[4]	✓[4]	✓[4]	✗	✗

[0] Only `returning [ varchar2 | number ]` — neither is a standard type.
[1] Defaults to `error on error`.
[2] No quotes behavior: ~~`[ keep | omit ] quotes`~~.
[3] `with unconditional wrapper` seems to be buggy. No quotes behavior: ~~`[ keep | omit ] quotes`~~.
[4] Without `plan` clause.

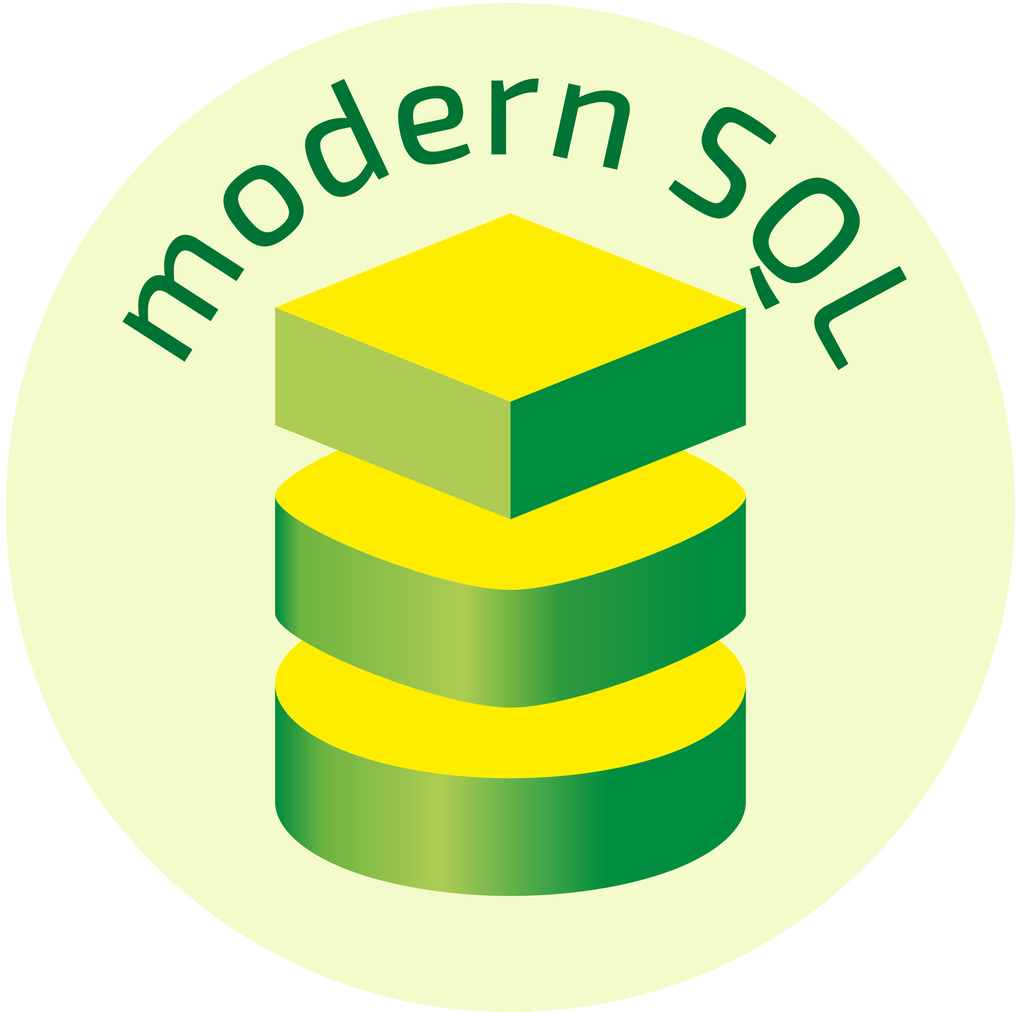
JSON — Preliminary testing of patches

	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
JSON path: `lax` mode (default)	✗	✓[1]	✓	✓	✓[0]	✓[1]
JSON path: strict mode	✗	✗	✗[2]	✗[2]	✓	✗
JSON path: item method	✗	✗	✓[3]	✓	✗	✗
JSON path: multiple subscripts	✗	✗	✓	✓	✗	✗
JSON path: `.*` member accessor	✗	✓	✓	✓	✗	✗
JSON path: filter expressions	✗	✗	✓[4]	✓[5]	✗	✗
JSON path: `starts with`	✗	✗	✓[4]	✓[7]	✗	✗
JSON path: `like_regex`	✗	✗	✗	✓	✗	✗

[0] `Lax` mode does not unwrap arrays.
[1] Keyword `lax` not accepted (only default mode). `Lax` mode does not unwrap arrays.
[2] Keyword `strict` is accepted but not honored.
[3] Only in filters. Not supporting `size()`, `datetime()`, `keyvalue()`. `type()` returns `null` for arrays.
[4] Not in `json_value`. Not in `json_query`. Not in `json_table`. Only as last step of expression.
[5] Not in `json_value`. Not in `json_query`.
[6] Not in `json_query`. Only as last step of expression.
[7] Not in `json_value`.

@ModernSQL

modern-sql.com



My other website:
https://use-the-index-luke.com

Training & co: https://winand.at/

Visit Vienna!

5-day intensive training — Sept 17-21

SQL Performance Kick-Start

- Indexing, Indexing, Indexing…
- Joining
- Ordering, grouping,
- …

Analysis and Aggregation

- OVER, GROUPING SETS
- Avoiding self-joins
- Grouping consecutive events
- …

SQL Reloaded

- Type safety, NULL an 3VL
- Modern interpretation of the relational model
- Keeping historic data
- Design guidelines

Recursive Queries

- WITH
- WITH RECURSIVE
- Use cases and examples

https://winand.at/sql-training/5-day-training/aug-sept-2018