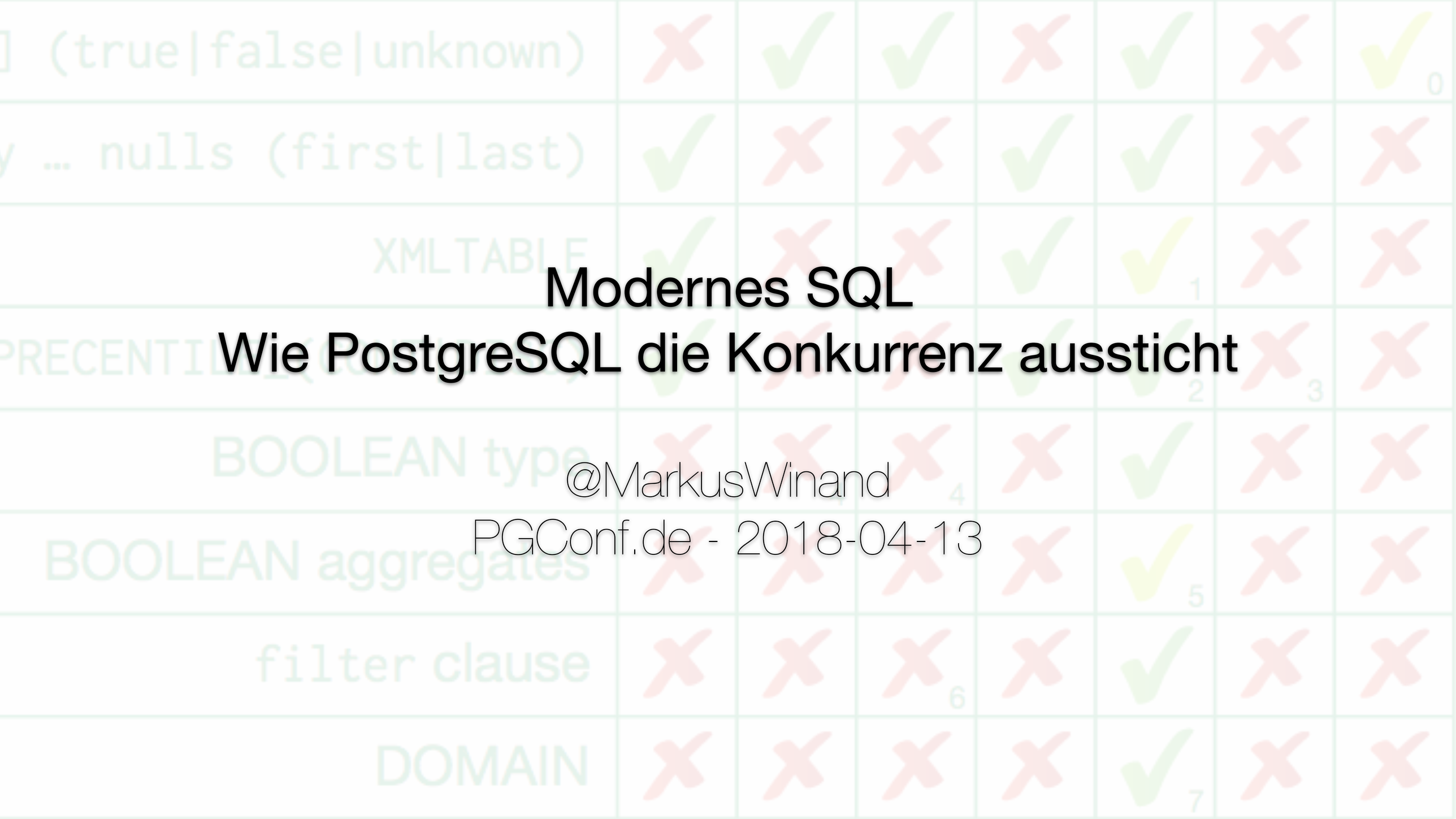


Modernes SQL
Wie PostgreSQL die Konkurrenz aussticht

@MarkusWinand

PGConf.de - 2018-04-13

FILTER

Before we start

In SQL, most aggregate functions*
drop `null` arguments
prior to the aggregation.

*Exceptions: Aggregate functions that return structured data:
`array_agg`, `json_objectagg`, `json_arrayagg`, `xmlagg`
See: http://modern-sql.com/concept/null#aggregates

FILTER

The Problem

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

```
SELECT YEAR,




  FROM sales
GROUP BY YEAR
```

FILTER

The Problem

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

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

FILTER

Since SQL:2003

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

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

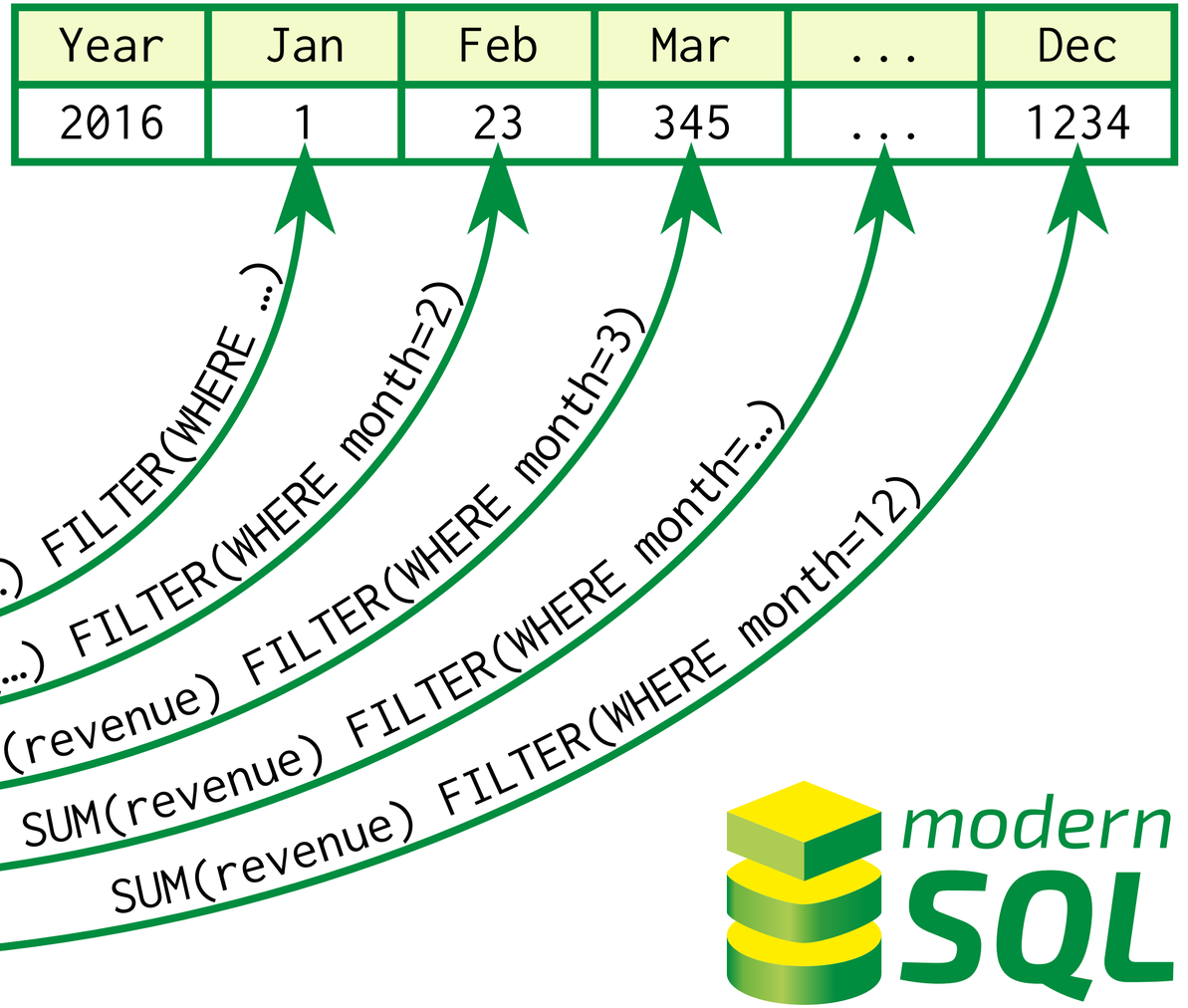
FILTER

Since SQL:2003

Pivot in SQL

1. Use `GROUP BY` to combine rows
2. Use `FILTER` to pick rows per column

Year	Month	Revenue
2016	1	1
2016	2	23
2016	3	345
2016	...	...
2016	12	1234



modern SQL

See: http://modern-sql.com/use-case/pivot

FILTER

Since SQL:2003

Use case: Flatten the EAV-Model
(entity-attribute-value)

```
SELECT ent

  FROM eav
GROUP BY ent
```

FILTER

Since SQL:2003

Use case: Flatten the EAV-Model
(entity-attribute-value)

```
SELECT ent
     , MAX(val) FILTER(WHERE att='name')    name
     , MAX(val) FILTER(WHERE att='email')   email
     , MAX(val) FILTER(WHERE att='website') website
  FROM eav
```

MAX works on strings too

ARRAY_AGG, XMLAGG, ... are useful too

Pick each attribute

FILTER

Since SQL:2003

Use case: Flatten the EAV-Model
(entity-attribute-value)

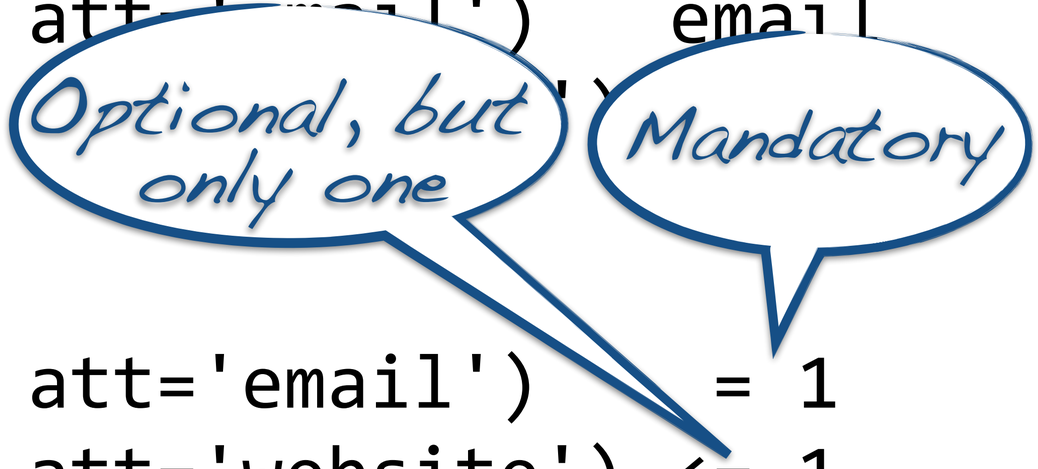
```
SELECT ent
     , MAX(val) FILTER(WHERE att='name')    name
     , MAX(val) FILTER(WHERE att='email')   email
     , MAX(val) FILTER(WHERE att='website')
  FROM eav
 GROUP BY ent
HAVING COUNT(*) FILTER(WHERE att='email')    = 1
```

FILTER

Since SQL:2003

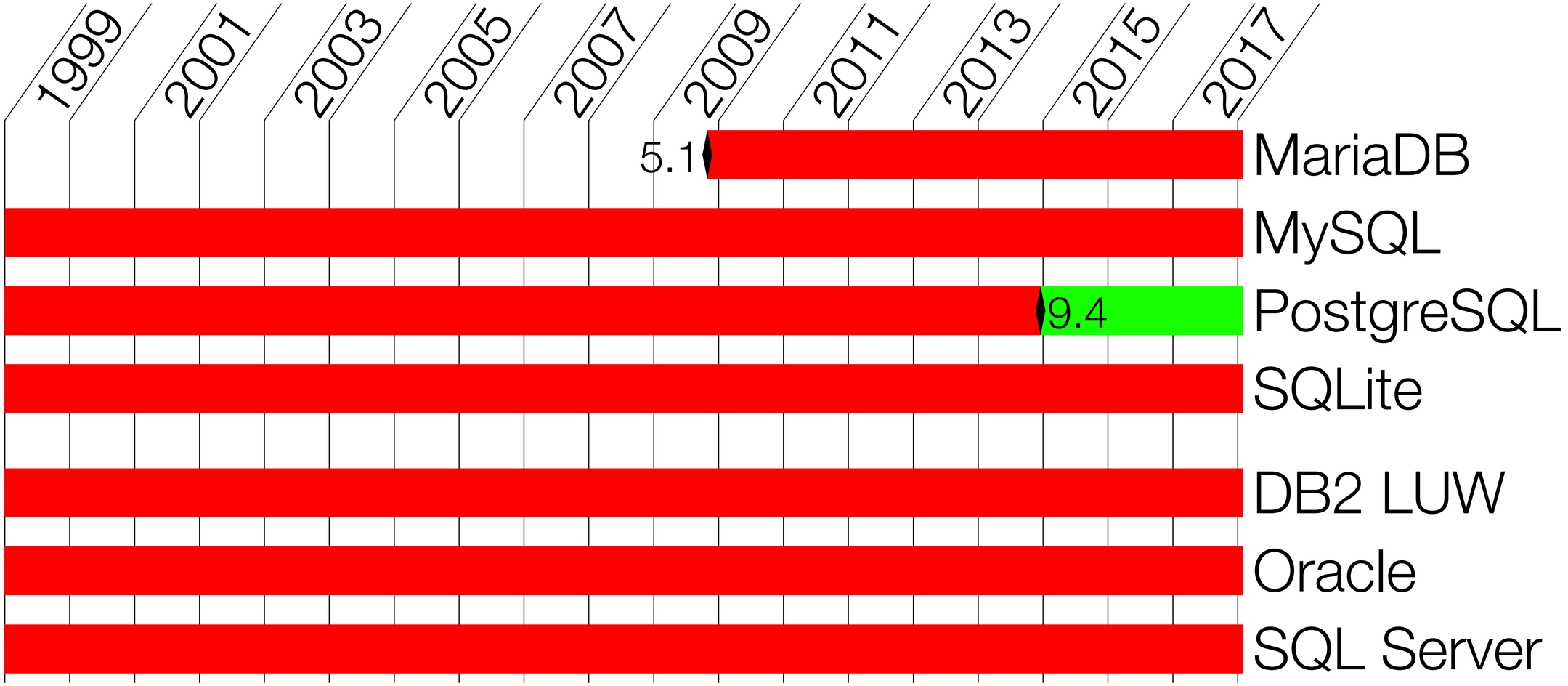
Use case: Flatten the EAV-Model
(entity-attribute-value)

```
SELECT ent
     , MAX(val) FILTER(WHERE att='name')    name
     , MAX(val) FILTER(WHERE att='email')   email
     , MAX(val) FILTER(WHERE att='website') website
  FROM eav
 GROUP BY ent
HAVING COUNT(*) FILTER(WHERE att='email')   = 1
   AND COUNT(*) FILTER(WHERE att='website') <= 1
```



FILTER

Availability



BOOLEAN Aggregates

`BOOLEAN` Aggregates

Before we start

SQL uses a three-valued logic.
Boolean values are either
`true`, `false` or `unknown`(=`null`).

See: http://modern-sql.com/concept/three-valued-logic

BOOLEAN Aggregates Since SQL:2003

Use case: Validate group properties
(previous example continued)

```
SELECT ent
     , MAX(val) FILTER(WHERE att='name')    name
     , MAX(val) FILTER(WHERE att='email')   email
     , MAX(val) FILTER(WHERE att='website') website
  FROM eav
 GROUP BY ent
HAVING COUNT(*) FILTER(WHERE att='email')    = 1
   AND COUNT(*) FILTER(WHERE att='website') <= 1
```

BOOLEAN Aggregates Since SQL:2003

Use case: Validate group properties
(previous example continued)

```
SELECT ent
     , MAX(val) FILTER(WHERE att='name')     name
     , MAX(val) FILTER(WHERE att='email')    email
     , MAX(val) FILTER(WHERE att='website')  website
  FROM eav
 GROUP BY ent
HAVING SOME(att='email')
   AND COUNT(*) FILTER(WHERE att='website') <= 1
```

Equivalent to

```
COUNT(*) FILTER(WHERE att='email') > 0
```

Assumption: constraint ensures only one email

BOOLEAN Aggregates

Since SQL:2003

ISO SQL

```
EVERY(<cond>)
```

BOOLEAN Aggregates

Since SQL:2003

ISO SQL

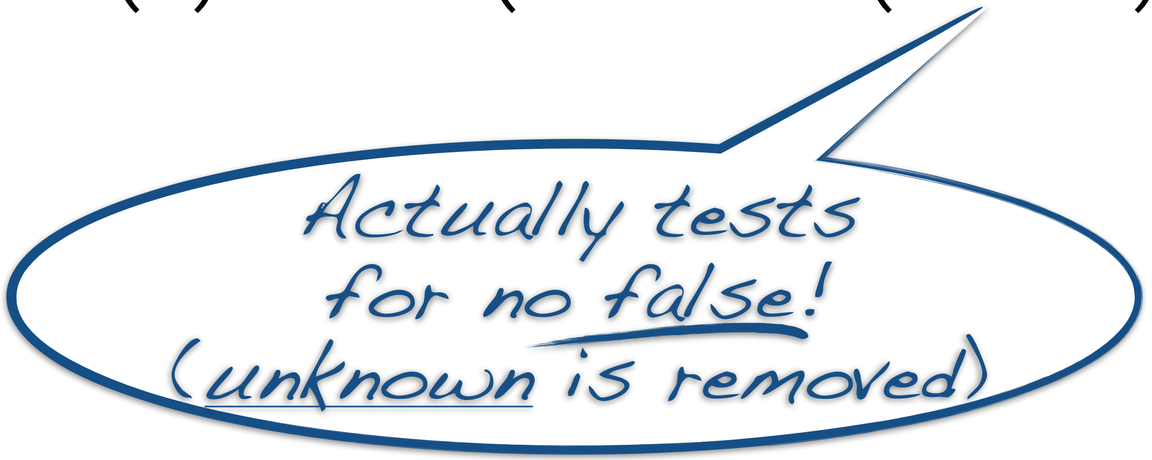
```
EVERY(<cond>) ⇔ COUNT(*) FILTER(WHERE NOT(<cond>)) = 0
```

BOOLEAN Aggregates

Since SQL:2003

ISO SQL

```
EVERY(<cond>) ⇔ COUNT(*) FILTER(WHERE NOT(<cond>)) = 0
```



BOOLEAN Aggregates

Since SQL:2003

ISO SQL

```
EVERY(<cond>) ⇔ COUNT(*) FILTER(WHERE NOT(<cond>)) = 0
 SOME(<cond>) ⇔ COUNT(*) FILTER(WHERE <cond>) > 0
  ANY(<cond>) ⇔ COUNT(*) FILTER(WHERE <cond>) > 0
```

Same!

BOOLEAN Aggregates Since SQL:2003

ISO SQL

```
EVERY(<cond>) ⇔ COUNT(*) FILTER(WHERE NOT(<cond>)) = 0
 SOME(<cond>) ⇔ COUNT(*) FILTER(WHERE <cond>) > 0
  ANY(<cond>) ⇔ COUNT(*) FILTER(WHERE <cond>) > 0
```

Same!

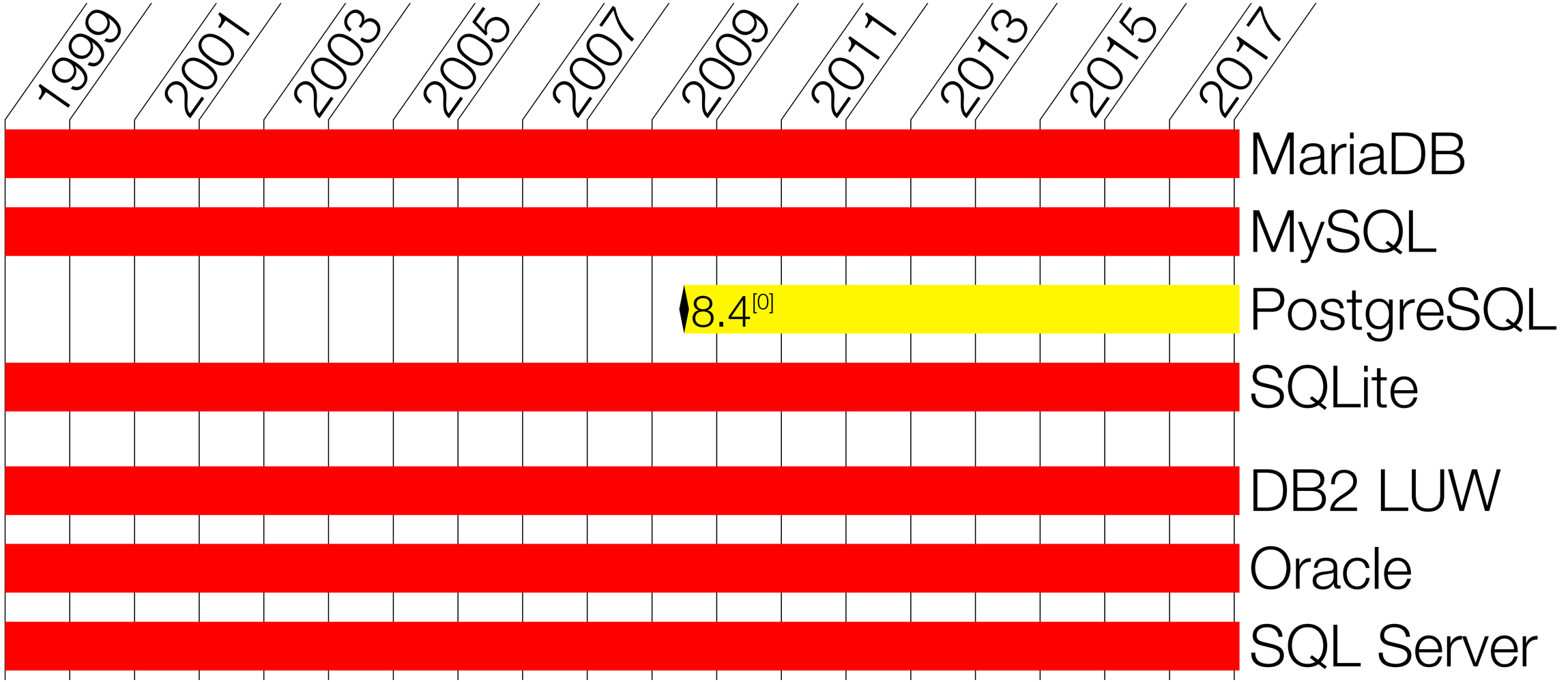
PostgreSQL seems to have a small incompatibility:
if all values are unknown, it returns unknown instead of true.

PostgreSQL

```
EVERY(…)     ⎫
BOOL_AND(…)  ⎭ ⇔ { SUM(CASE … WHEN TRUE THEN 0
                           WHEN FALSE THEN 1 END) = 0

BOOL_OR(…)   ⇔  SUM(CASE … WHEN TRUE  THEN 1
                           WHEN FALSE THEN 0 END) > 0
```

BOOLEAN Aggregates Since SQL:2003



[0]Only EVERY(), which returns UNKNOWN if everything is NULL. Also: bool_or (similar to SOME).

BOOLEAN Tests

`BOOLEAN` Tests

Since SQL:2003

Similar to `is null`, there are tests for each Boolean value
(of which there are three: `true`, `false`, `unknown`/`null`)

```
IS [NOT] [TRUE|FALSE|UNKNOWN]
```

Example:

```
<cond> IS NOT TRUE
```

`BOOLEAN` Tests

Since SQL:2003

Similar to `is null`, there are tests for each Boolean value
(of which there are three: `true`, `false`, `unknown`/`null`)

```
IS [NOT] [TRUE|FALSE|UNKNOWN]
```

Example:

```
COUNT(*) FILTER(WHERE <cond> IS NOT TRUE)
```

`BOOLEAN` Tests

Since SQL:2003

Similar to `is null`, there are tests for each Boolean value
(of which there are three: `true`, `false`, `unknown`/`null`)

```
IS [NOT] [TRUE|FALSE|UNKNOWN]
```

Example:
Truly checking for “every” (no `false`, no `unknown`):

```
COUNT(*) FILTER(WHERE <cond> IS NOT TRUE) = 0
```

(empty group returns `true` — like ISO SQL’s `every`)

BOOLEAN Tests

Since SQL:2003

Similar to **`is null`**, there are tests for each Boolean value
(of which there are three: **`true`**, **`false`**, **`unknown`**/**`null`**)

```
IS [NOT] [TRUE|FALSE|UNKNOWN]
```

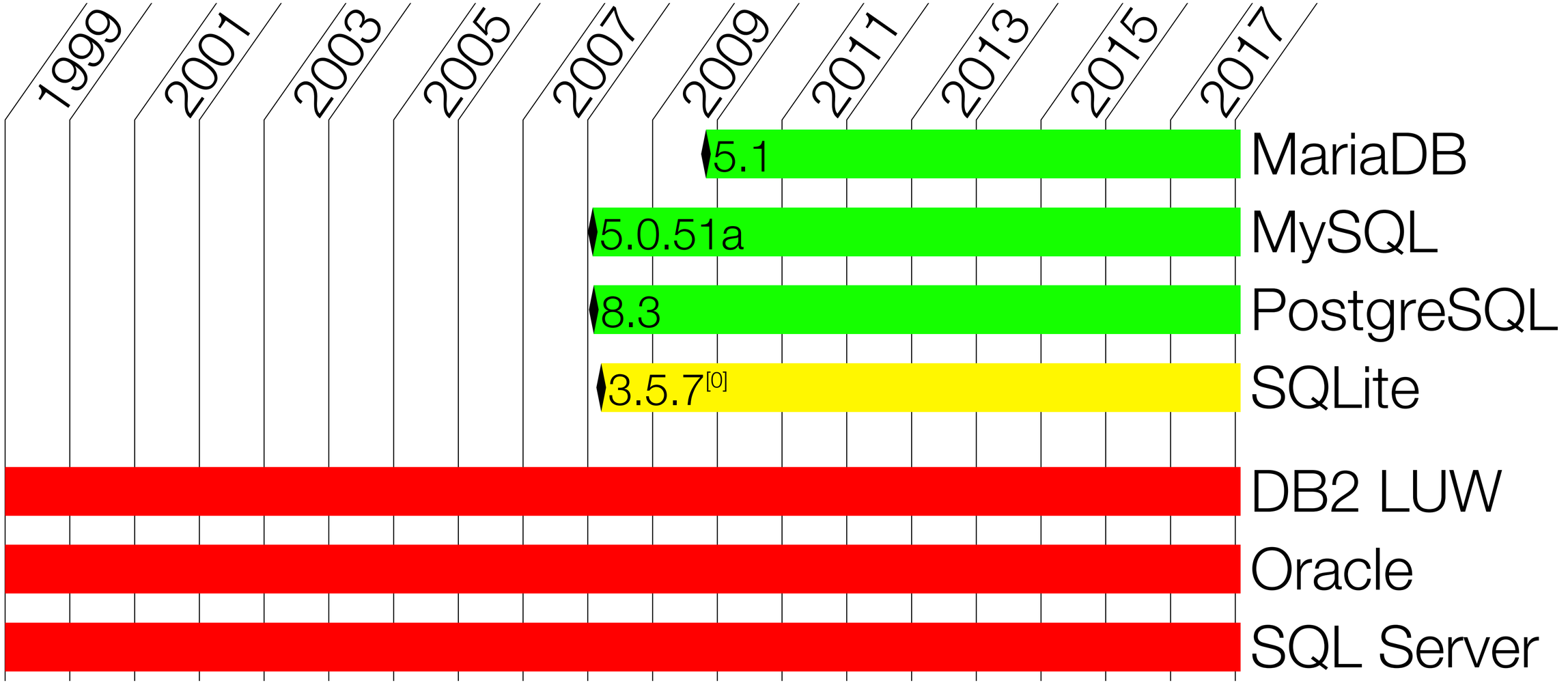
Example:
Truly checking for “every” (no **`false`**, no **`unknown`**):

```
COUNT(*) FILTER(WHERE <cond> IS NOT TRUE) = 0
  COUNT(*) FILTER(WHERE <cond>) = COUNT(*)
```

(empty group returns **`true`** — like ISO SQL’s **`every`**)

BOOLEAN Tests

Since SQL:2003



[0]No IS [NOT] UNKNOWN. Use IS [NOT] NULL instead

BOOLEAN Type

BOOLEAN Type

Since SQL:2003

```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL,
        …
)


SELECT …
  FROM …
 WHERE NOT(deleted)
```

BOOLEAN Type

Since 2003


```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL,
        …
)
```

Alias for tinyint

BOOLEAN Type

Since 2003

```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL,
        …
)
```

Alias for tinyint

```
INSERT … (…, deleted, …) VALUES (…, true, …)
```

BOOLEAN Type

Since 2003

```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL,
        …
)
```

Alias for tinyint

```
INSERT … (…, deleted, …) VALUES (…, true, …)
INSERT … (…, deleted, …) VALUES (…, false, …)
```

BOOLEAN Type

Since 2003

```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL,
        …
)
```

Alias for tinyint

```
INSERT … (…, deleted, …) VALUES (…, true, …)
INSERT … (…, deleted, …) VALUES (…, false, …)
INSERT … (…, deleted, …) VALUES (…, 42, …)
```

BOOLEAN Type

Since 2003

```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL UNIQUE,
        …
)
```

Alias for tinyint

```
INSERT … (…, deleted, …) VALUES (…, true, …)
INSERT … (…, deleted, …) VALUES (…, false, …)
INSERT … (…, deleted, …) VALUES (…, 42, …)
```

BOOLEAN Type

Since 2003

```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL UNIQUE,
        …
)
```

Alias for tinyint

```
INSERT … (…, deleted, …) VALUES (…, true, …)
INSERT … (…, deleted, …) VALUES (…, false, …)
INSERT … (…, deleted, …) VALUES (…, 42, …)
```

```
+----+
| de |
+----+
|  1 |
|  0 |
| 42 |
+----+
```

`BOOLEAN` Type

Since SQL:2003

Note that `boolean` in base tables is often questionable:

- `deleted` flags are often better represented as `deleted_at`
  (or more advanced temporal models)
- States often need more than two (or three) values
  consider using an enum instead

See: 3 Reasons I Hate Booleans In Databases by Jeff Potter
https://medium.com/@jpotts18/646d99696580

`BOOLEAN` Type

Since SQL:2003

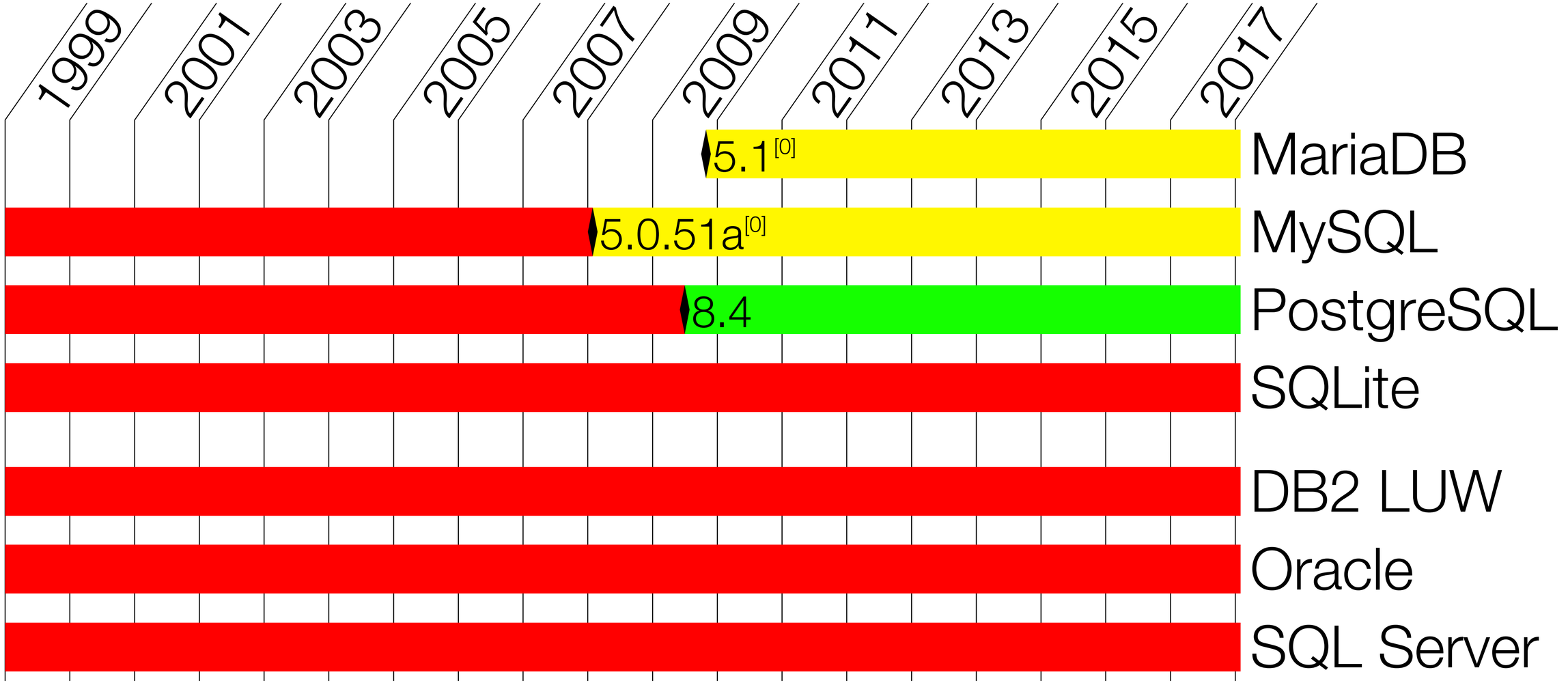
However, `boolean` is also useful in

derived tables (subqueries)
to improve readability


```
SELECT order_id
     , SOME(gift_wrap IS NOT NULL) contains_gifts
  FROM order_lines
 GROUP BY order_id
```

BOOLEAN Type

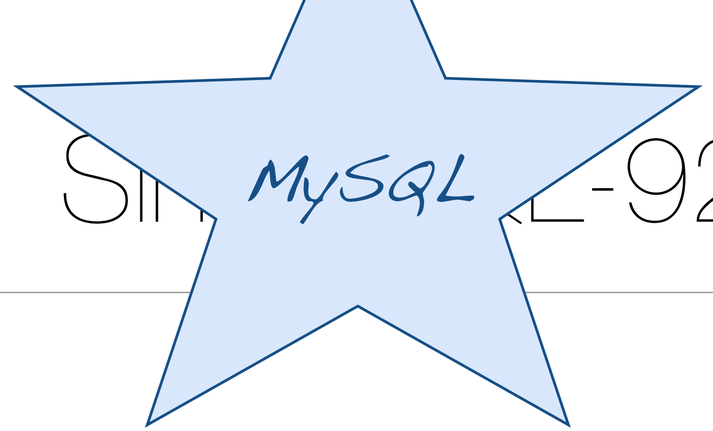
Since SQL:2003



[0]BOOLEAN, TRUE, FALSE are aliases for TINYINT(1), 1, 0 respectivley.

CHECK Constraints

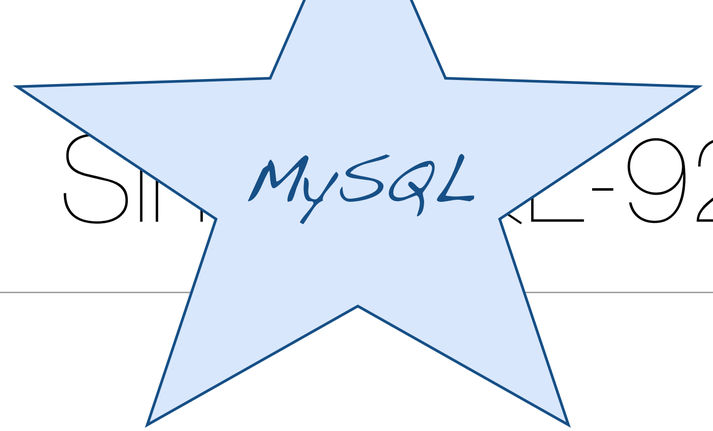
CHECK Constraints



SQL-92

```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL
                CHECK (deleted IN (true, false)),
        …
)
```

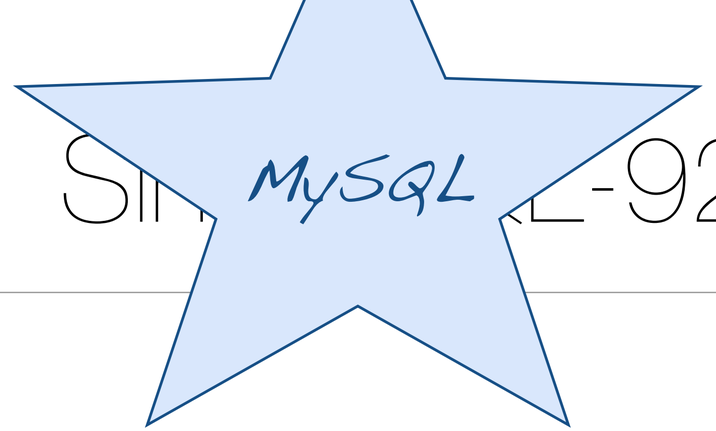
CHECK Constraints




```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL
                CHECK (deleted IN (true, false)),
        …
)

INSERT … (…, deleted, …) VALUES (…, true, …)
```

CHECK Constraints

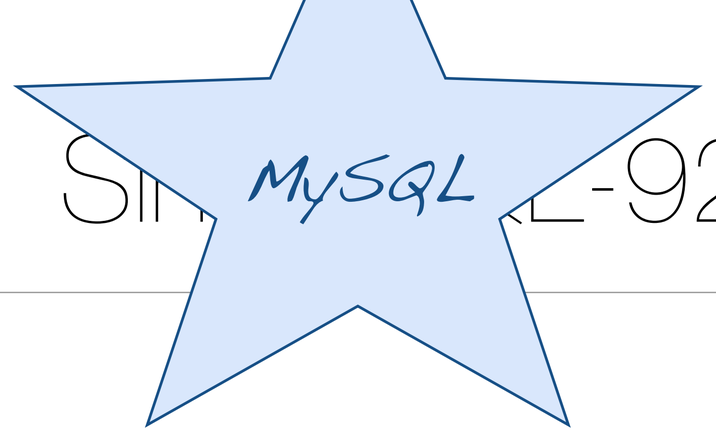


SQL-92

```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL
                CHECK (deleted IN (true, false)),
        …
)

INSERT … (…, deleted, …) VALUES (…, true, …)
INSERT … (…, deleted, …) VALUES (…, false, …)
```

CHECK Constraints



SQL-92

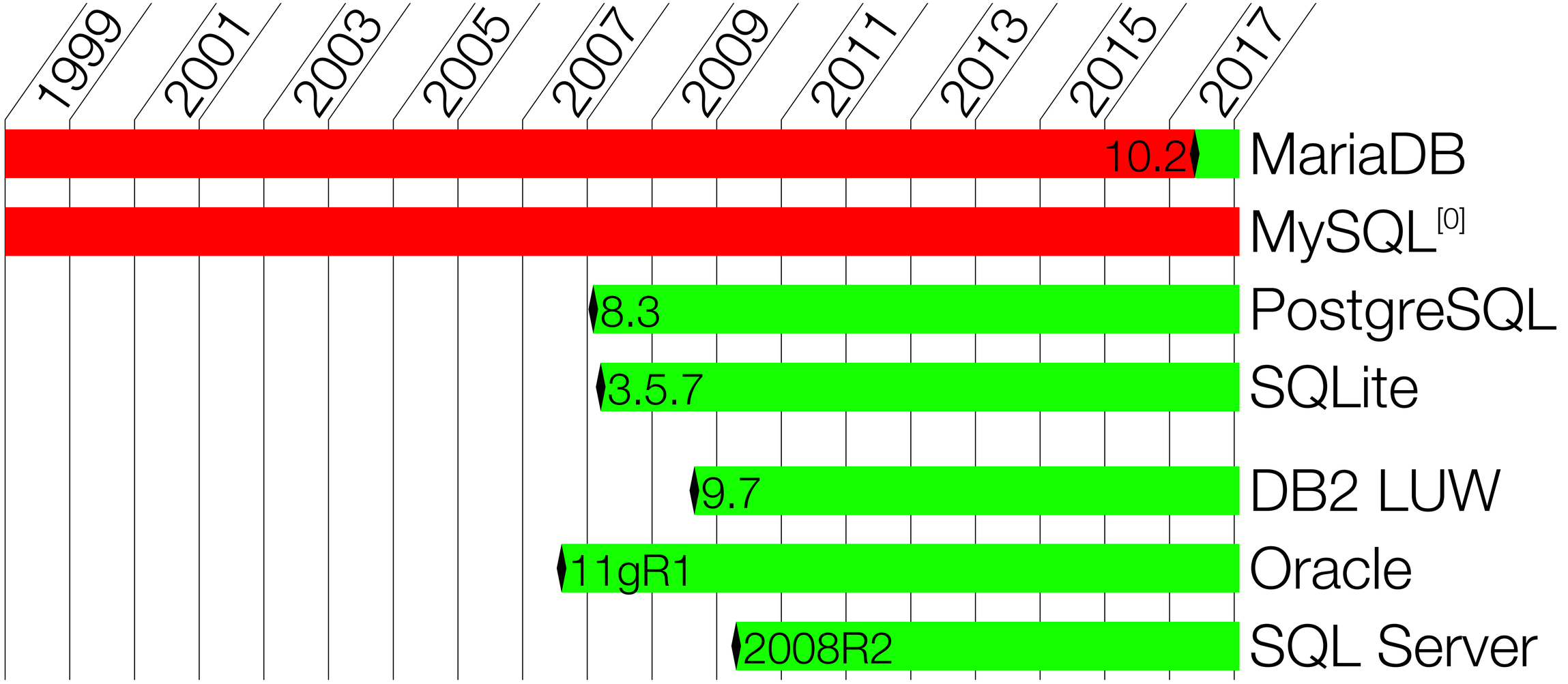
```
CREATE TABLE … (
        …
        deleted BOOLEAN NOT NULL
                CHECK (deleted IN (true, false)),
```

Syntax accepted,
Constraint ignored

```
INSERT … (…, deleted, …) VALUES (…, true, …)
INSERT … (…, deleted, …) VALUES (…, false, …)
INSERT … (…, deleted, …) VALUES (…, 42, …)
```

CHECK Constraints

Since SQL-92



[0]Syntax accepted, but ignored without notice.

DOMAIN

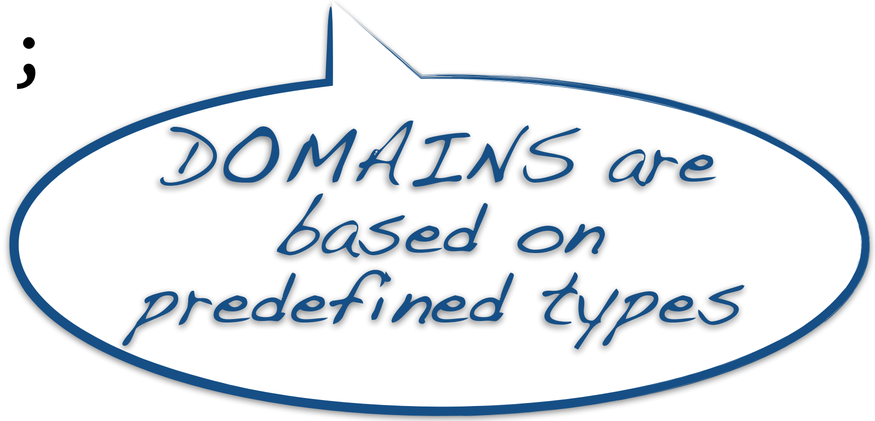
DOMAIN

Since SQL:2003

A SQL domain is a set of permissible values.[SQL:2016-2: §4.12]

Or: A way to manage **CHECK** constraints and **DEFAULT**s.

```
CREATE DOMAIN positive_int AS INTEGER
        CHECK (VALUE > 0);
```



DOMAIN

Since SQL:2003

A SQL domain is a set of permissible values.[SQL:2016-2: §4.12]

Or: A way to manage **CHECK** constraints and **DEFAULT**s.

```
CREATE DOMAIN positive_int AS INTEGER
        CHECK (VALUE > 0);

CREATE TABLE order_lines (
    …,
    quantity positive_int NOT NULL,
    …
);
```

DOMAIN

Since SQL:2003

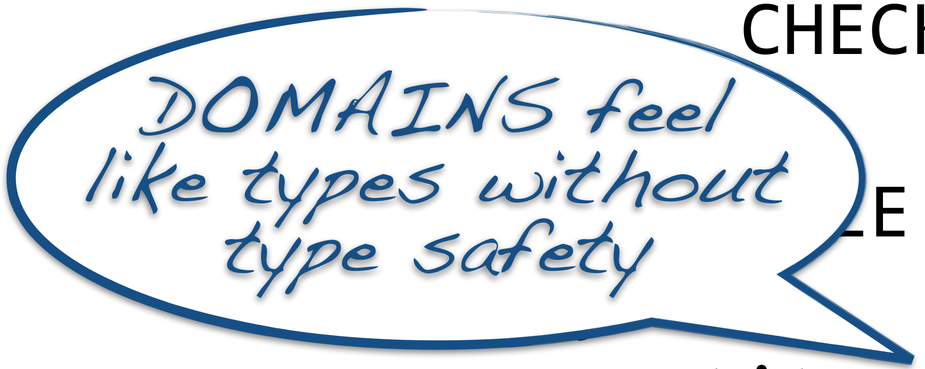
A SQL domain is a set of permissible values.[SQL:2016-2: §4.12]

Or: A way to manage **CHECK** constraints and **DEFAULT**s.

```
CREATE DOMAIN positive_int AS INTEGER
             CHECK (VALUE > 0);

          LE order_lines (

    quantity positive_int NOT NULL,
    …
);
```



DOMAIN

Since SQL:2003

A SQL domain is a set of permissible values.[SQL:2016-2: §4.12]

Or: A way to manage **CHECK** constraints and **DEFAULT**s.

```
CREATE DOMAIN positive_int AS INTEGER
        CHECK (VALUE > 0);

CREATE TABLE order_
    quantity positive_int NOT NULL,
    …
);
```

DOMAINS feel like types without type safety

CAST(… AS <domain>) casts to the base type and checks the constraint

DOMAIN

Since SQL:2003

Domains can have multiple, named check constraints.

```
CREATE DOMAIN positive_int AS INTEGER
       CONSTRAINT gt_zero CHECK (VALUE > 0);
```

DOMAIN

Since SQL:2003

Domains can have multiple, named check constraints.

```
CREATE DOMAIN positive_int AS INTEGER
       CONSTRAINT gt_zero CHECK (VALUE > 0);

 ALTER DOMAIN positive_int
   ADD CONSTRAINT ge_zero CHECK (VALUE >= 0);
```

DOMAIN

Since SQL:2003

Domains can have multiple, named check constraints.

```
CREATE DOMAIN positive_int AS INTEGER
       CONSTRAINT gt_zero CHECK (VALUE > 0);

 ALTER DOMAIN positive_int
   ADD CONSTRAINT ge_zero CHECK (VALUE >= 0);

 ALTER DOMAIN positive_int
  DROP CONSTRAINT gt_zero;
```

DOMAIN

Since SQL:2003

Domains can have multiple, named check constraints.

```
CREATE DOMAIN positive_int AS INTEGER
       CONSTRAINT gt_zero CHECK (VALUE > 0);

 ALTER DOMAIN positive_int
   ADD CONSTRAINT ge_zero CHECK (VALUE >= 0);

 ALTER DOMAIN positive_int
  DROP CONSTRAINT gt_zero;

 ALTER DOMAIN positive_int RENAME TO unsigned_int;
```

DOMAIN

Since 2003

PostgreSQL has a great extension: **NOT VALID**

DOMAIN

Since PostgreSQL 2003

PostgreSQL has a great extension: **NOT VALID**

```
ALTER DOMAIN unsigned_int
  ADD CONSTRAINT gt_zero CHECK (VALUE > 0) NOT VALID;
```

Enforced on INSERT & UPDATE but not for existing values

DOMAIN

Since 2003

PostgreSQL has a great extension: **NOT VALID**

```
ALTER DOMAIN unsigned_int
  ADD CONSTRAINT gt_zero CHECK (VALUE > 0) NOT VALID;

UPDATE order_lines
   SET quantity = ?
 WHERE quantity = 0;
```

DOMAIN

Since PostgreSQL 7.3

PostgreSQL has a great extension: **NOT VALID**

```
ALTER DOMAIN unsigned_int
  ADD CONSTRAINT gt_zero CHECK (VALUE > 0) NOT VALID;

UPDATE order_lines
   SET quantity = ?
 WHERE quantity = 0;

ALTER DOMAIN unsigned_int
    VALIDATE CONSTRAINT ge_zero;
```

Enforced on INSERT & UPDATE but not for existing values

DOMAIN

Since 2003

PostgreSQL has a great extension: **NOT VALID**

```
ALTER DOMAIN unsigned_int
  ADD CONSTRAINT gt_zero CHECK (VALUE > 0) NOT VALID;

UPDATE order_lines
   SET quantity = ?
 WHERE quantity = 0;

ALTER DOMAIN unsigned_int
    VALIDATE CONSTRAINT ge_zero;

ALTER DOMAIN unsigned_int DROP CONSTRAINT gt_zero;
```

Enforced on INSERT & UPDATE but not for existing values

DOMAIN

Since PostgreSQL 2003

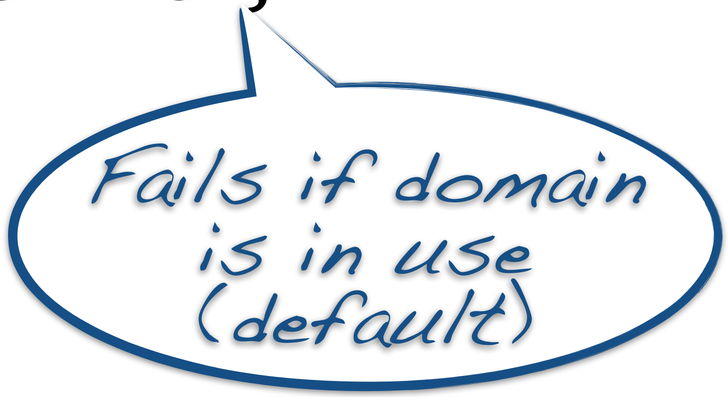
PostgreSQL handles **DROP DOMAIN … CASCADE** proprietarily:

DOMAIN

Since 2003

PostgreSQL handles **DROP DOMAIN … CASCADE** proprietarily:

```
DROP DOMAIN unsigned_int RESTRICT;
```



DOMAIN

Since 2003

PostgreSQL handles **DROP DOMAIN … CASCADE** proprietarily:

```
DROP DOMAIN unsigned_int RESTRICT;

DROP DOMAIN unsigned_int CASCADE;
```

ISO behaviour is to copy the check constraints to the columns

Drops COLUMNS that use the domain (and the domain)

DOMAIN

Since SQL:2003

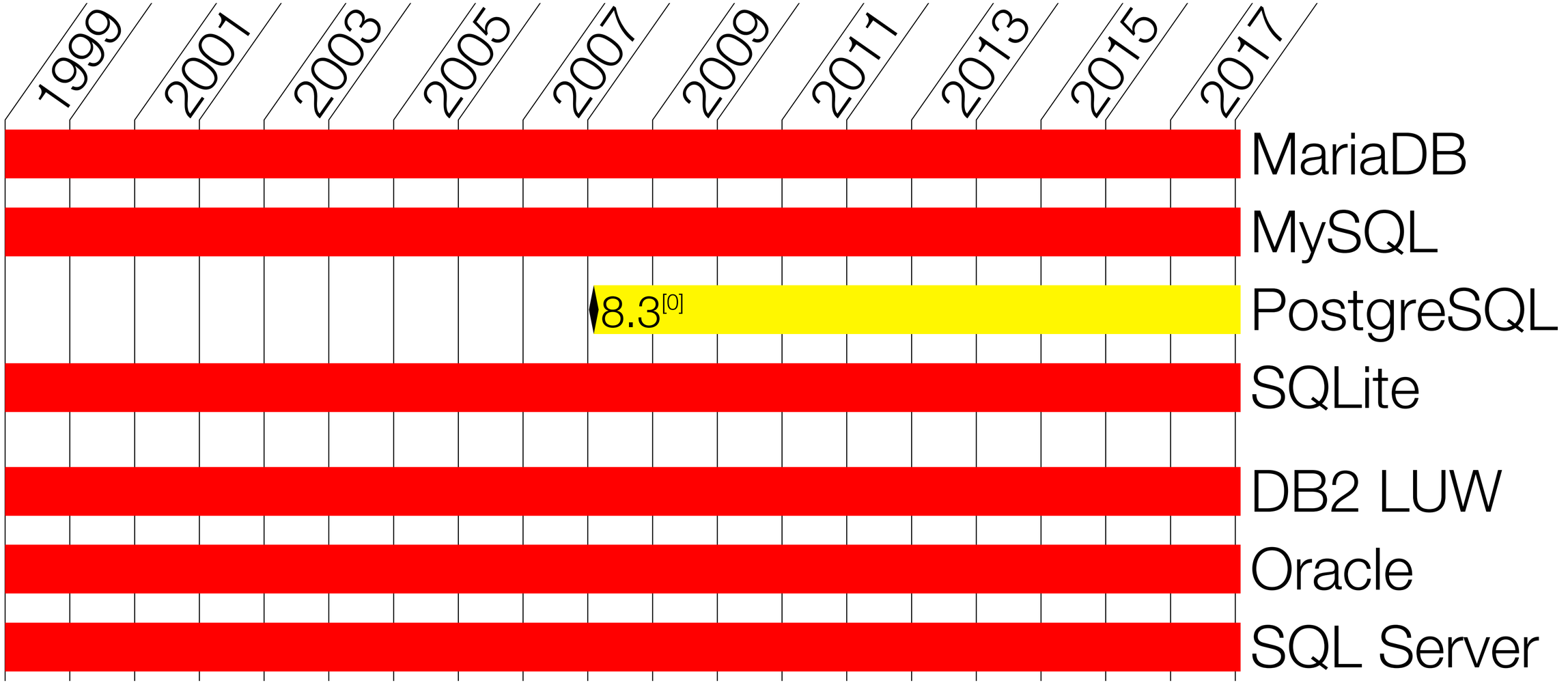
Domains can also specify a **DEFAULT** value.

```
ALTER DOMAIN unsigned_int SET DEFAULT 1;

ALTER DOMAIN unsigned_int DROP DEFAULT;
```

DOMAIN

Since SQL:2003



[0]DROP DOMAIN differs from the standard: defaults to RESTRICT, drops columns on CASCADE.

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, PostgreSQL supports only XPath

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,
PostgreSQL supports only XPath

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, PostgreSQL supports only XPath

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, PostgreSQL supports only XPath

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
```

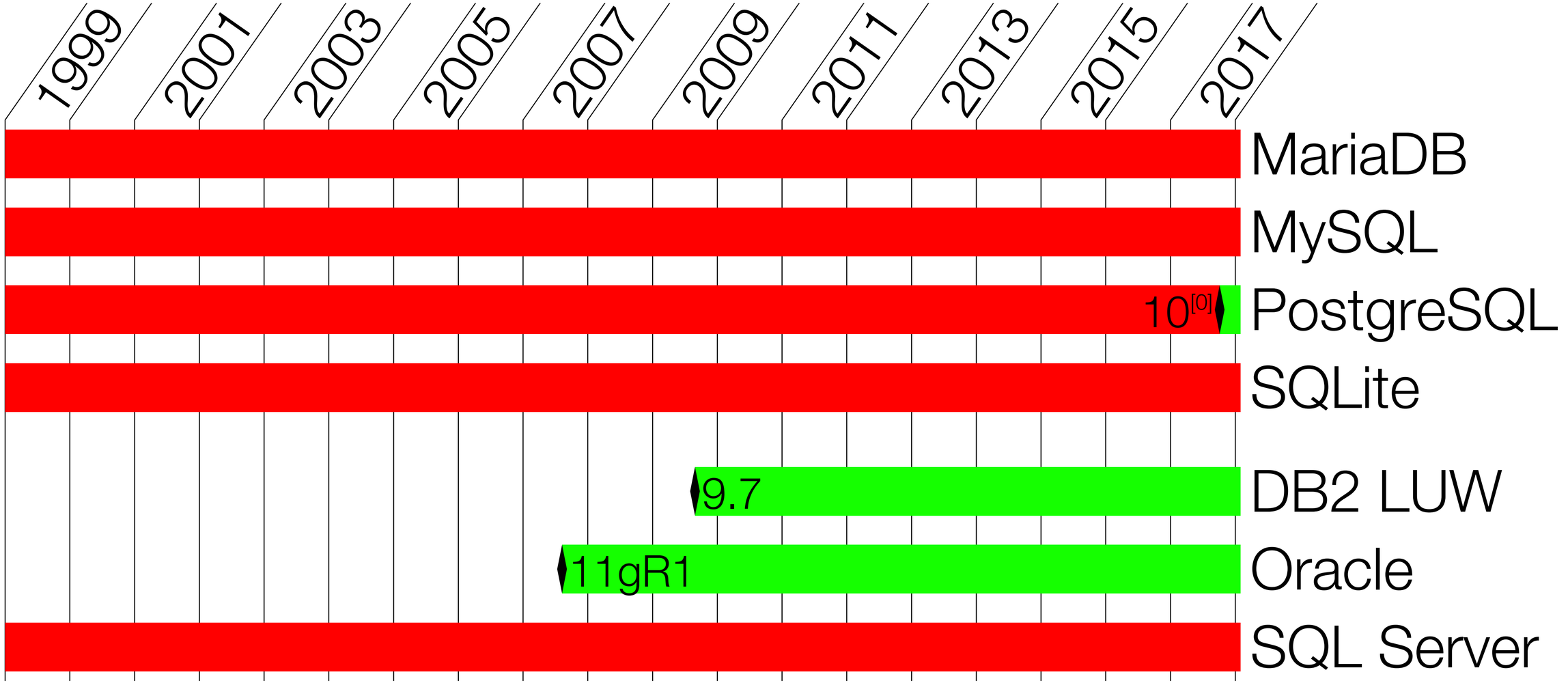
Stored in tbl.x:

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

*Standard SQL allows XQuery,
PostgreSQL supports only XPath

XMLTABLE

Availability



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

NULLS FIRST/LAST

NULLS FIRST/LAST Before SQL:2003

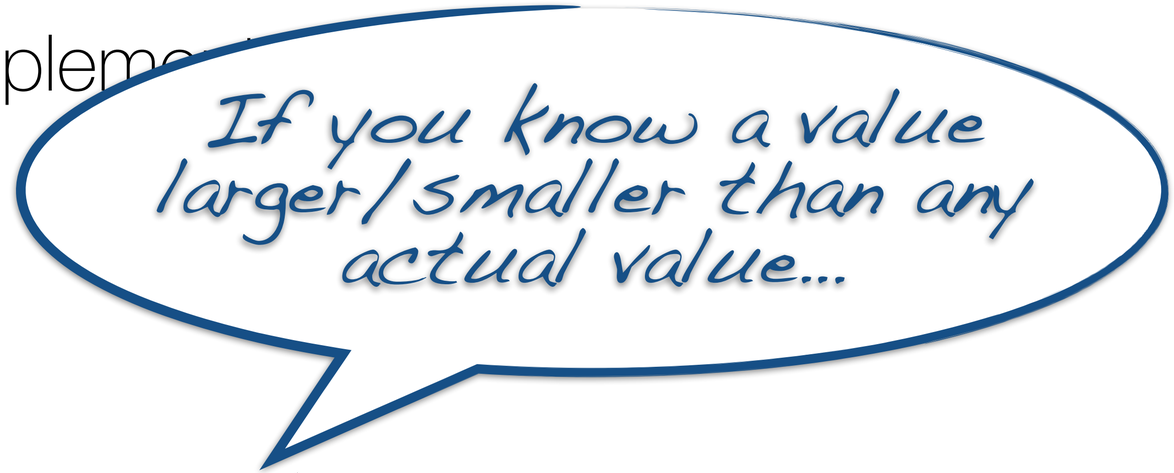
The sorting of **`NULL`** is implementation defined

(some DBs sort **`NULL`** as great, others as very small value)

NULLS FIRST/LAST Before SQL:2003

The sorting of **NULL** is implem

(some DBs sort **NULL** as great,




```
SELECT …
  FROM …
 ORDER BY COALESCE(nullable, ?);
```

NULLS FIRST/LAST

Before SQL:2003

The sorting of **NULL** is implementation defined

(some DBs sort **NULL** as great, others as

```
SELECT …
  FROM …
 ORDER BY COALESCE(nullable, ?);


 ORDER BY CASE WHEN nullable IS NULL THEN 0
                                    ELSE 1
           END
       , nullable;
```

This shows NULLs first (no matter if nullable is sorted ASC or DESC)

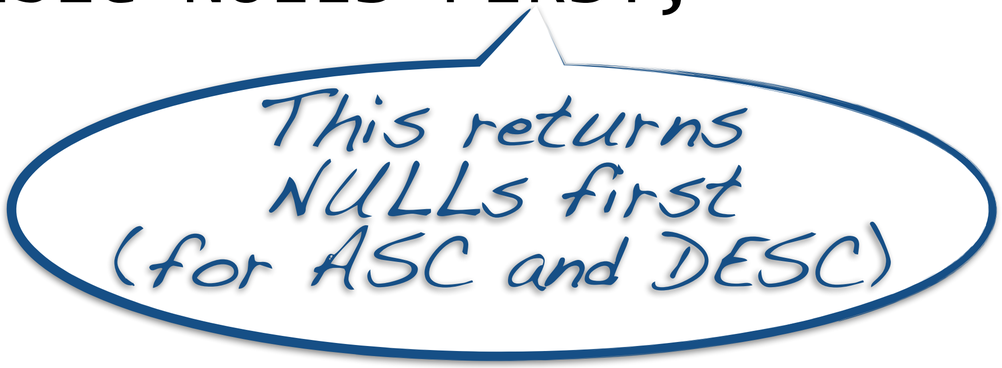
Using an extra sort key to put NULL and NOT NULL apart is more robust

NULLS FIRST/LAST

Since SQL:2003

SQL:2003 introduced `ORDER BY … NULLS FIRST/LAST`

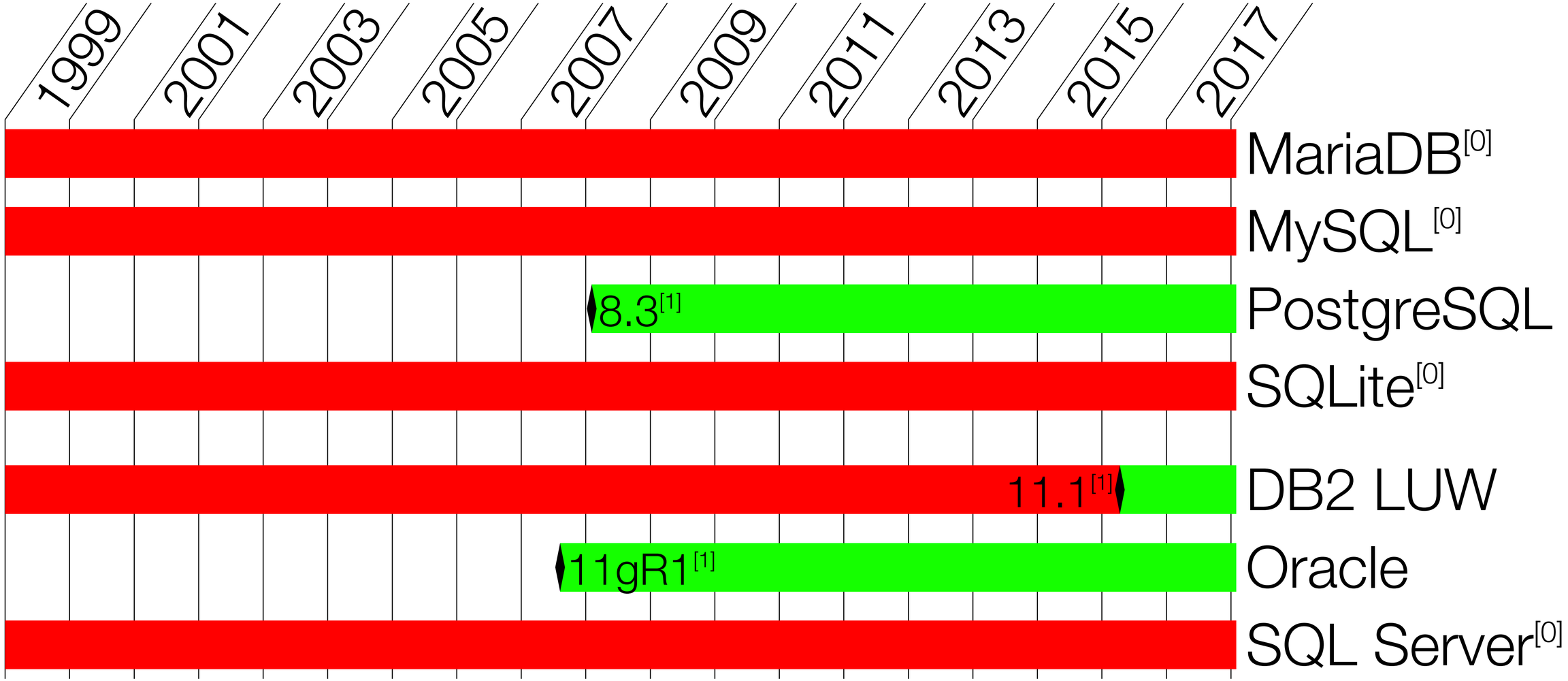
```
SELECT …
  FROM …
 ORDER BY nullable NULLS FIRST;
```



Note: PostgreSQL accepts `NULLS FIRST/LAST` in index definitions.

NULLS FIRST/LAST

Since SQL:2003



[0]By default sorted as smallest
[1]By default sorted as greatest

Inverse Distribution Functions
(percentiles)

Inverse Distribution Functions

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

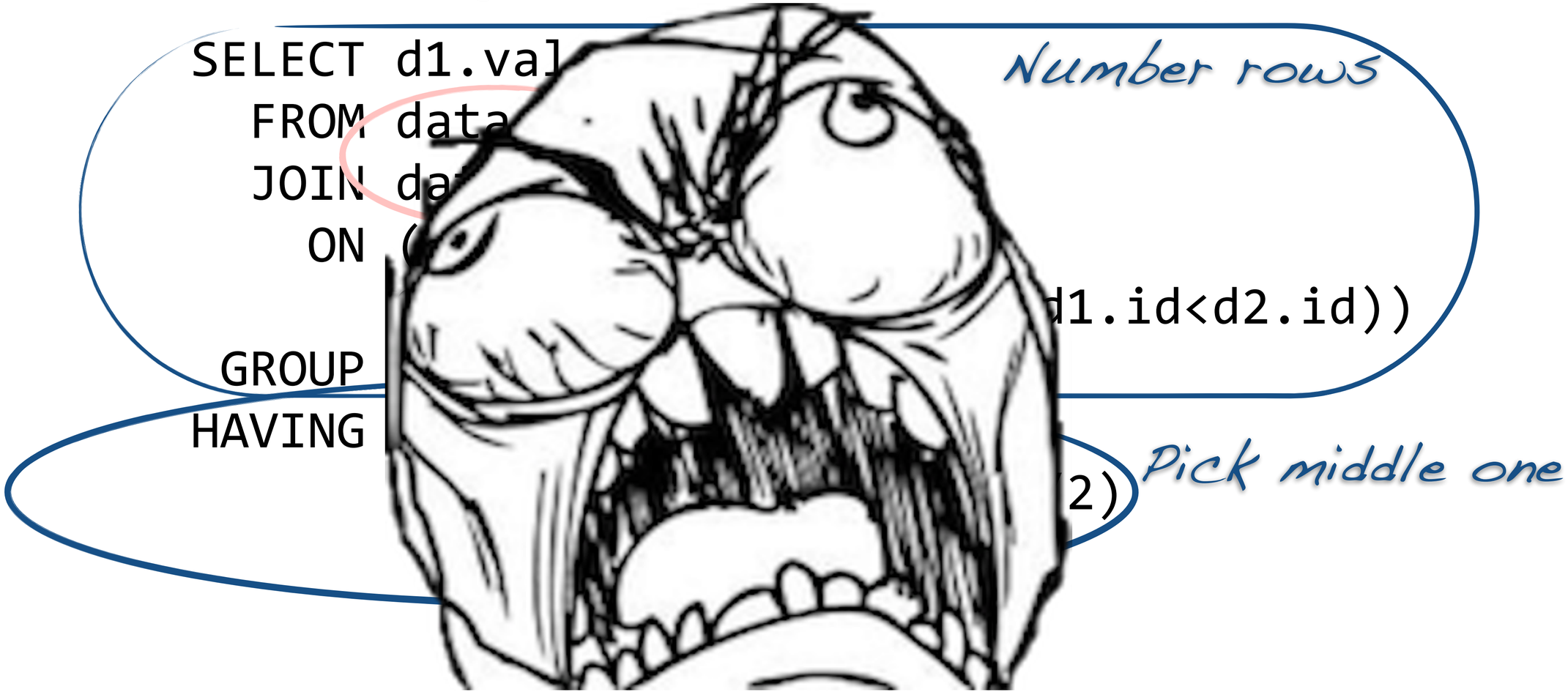
Pick middle one

Inverse Distribution Functions

The Problem

Grouped rows cannot be ordered prior aggregation.

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



Inverse Distribution Functions Since SQL:2003

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

Median

Which value?

Inverse Distribution Functions Since SQL:2003

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

Two variants:

- for discrete values (categories)
- for continuous values (linear interpolation)

Inverse Distribution Functions Since SQL:2003

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

Two variants:

- for discrete values (categories)
- for continuous values (linear interpolation)

1

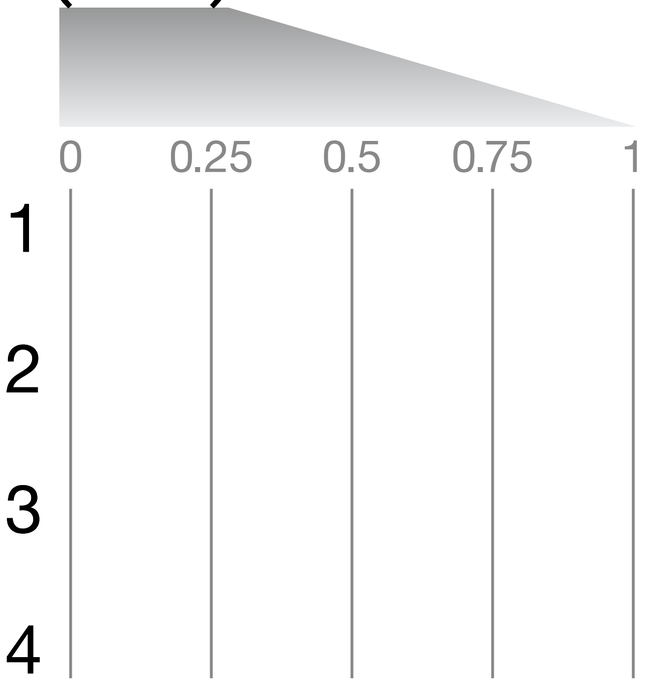
2

3

4

Inverse Distribution Functions Since SQL:2003

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



Two variants:

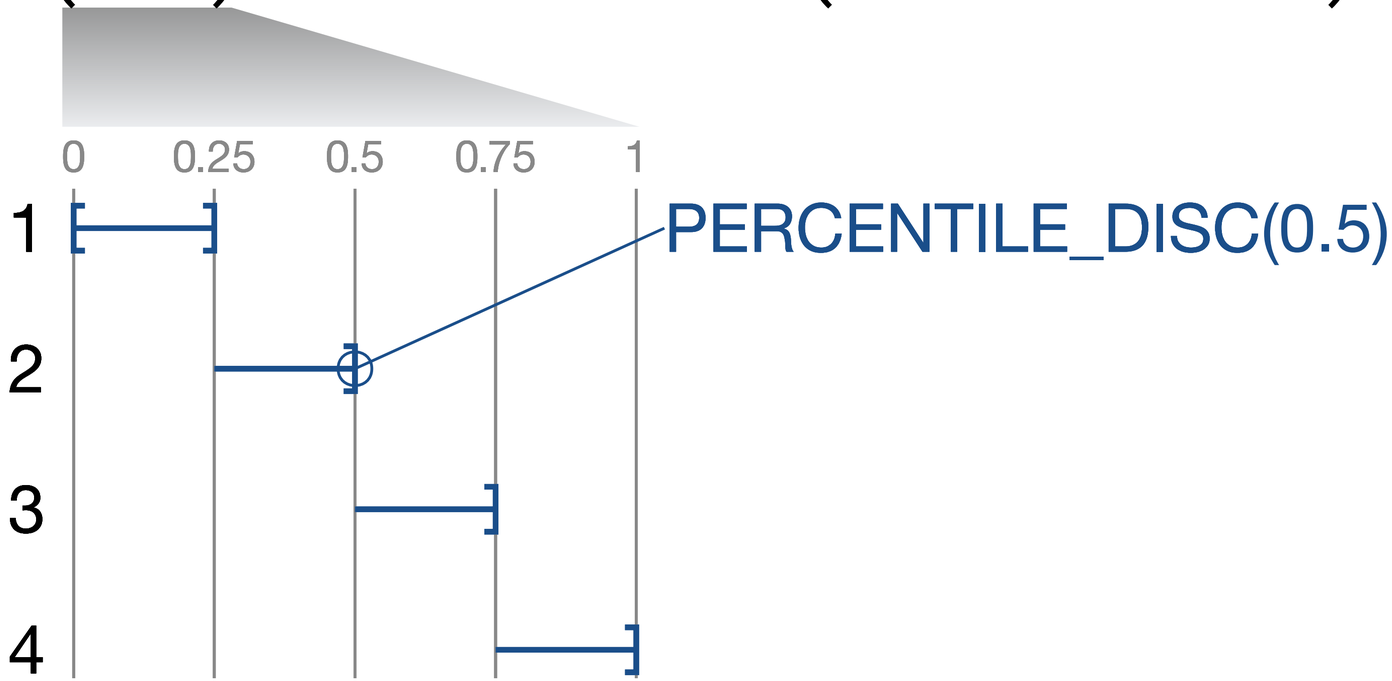
- for <u>discrete</u> values (categories)
- for <u>continuous</u> values (linear interpolation)

Inverse Distribution Functions Since SQL:2003

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

Two variants:

- for discrete values (categories)
- for continuous values (linear interpolation)

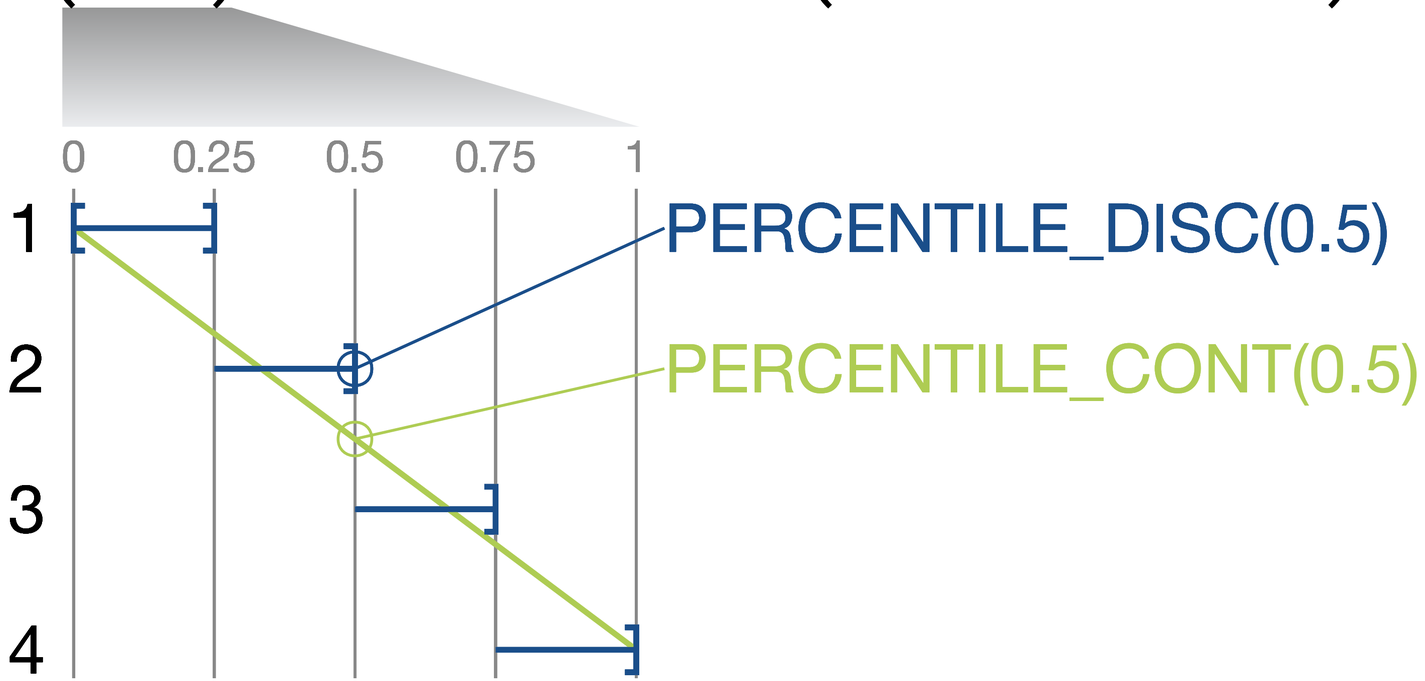


Inverse Distribution Functions Since SQL:2003

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

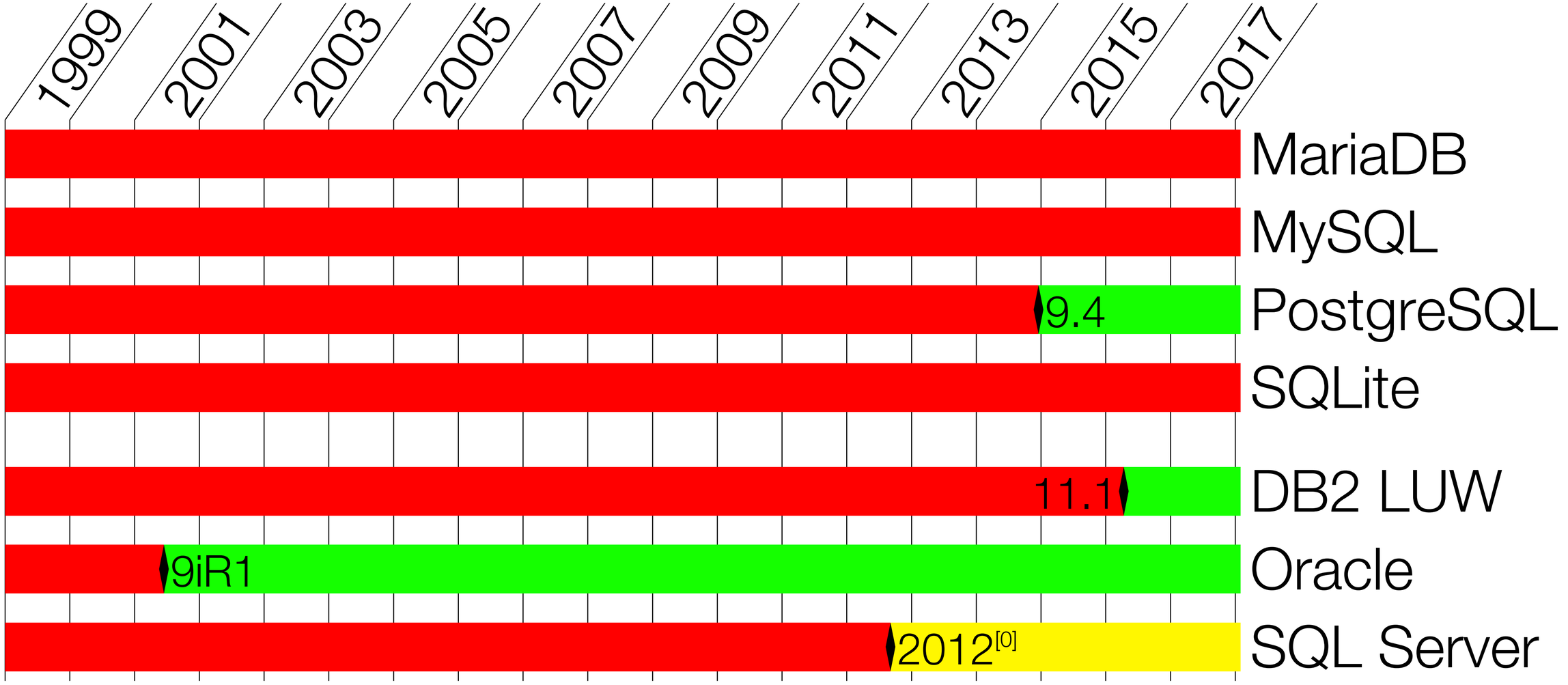
Two variants:

- for discrete values (categories)
- for continuous values (linear interpolation)



Inverse Distribution Functions

Availability



[0]Only as window function (requires OVER clause)

MERGE

(won't come before PostgreSQL 12)

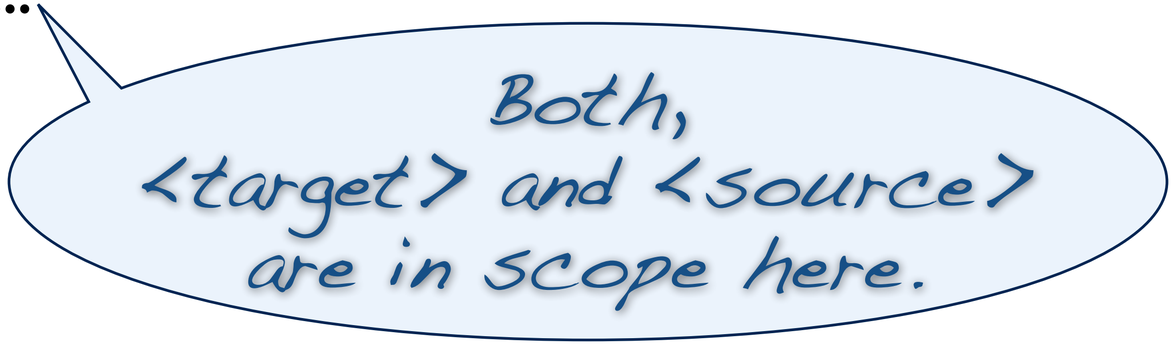
(was committed, is now reverted)

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:

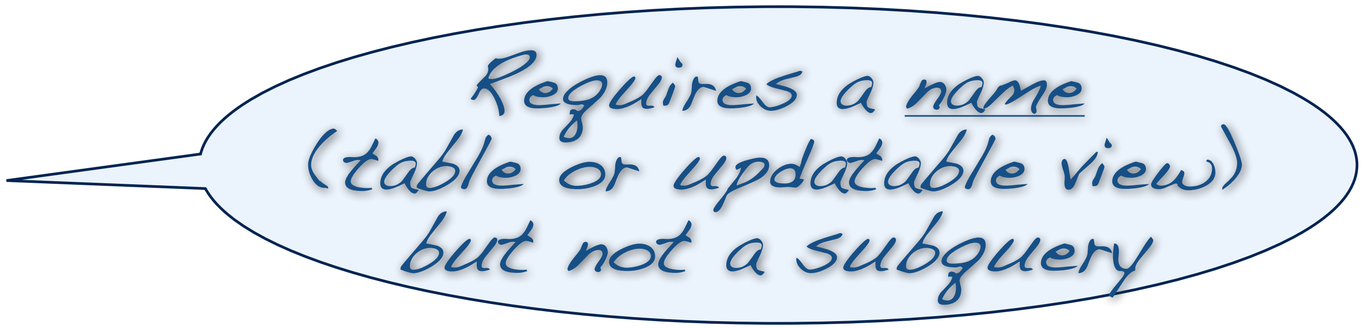
```
DELETE FROM <target>
 WHERE EXISTS (SELECT *
                 FROM <source>
                WHERE …
               )
```

MERGE

The Problem

Updating rows from another table is awkward:

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

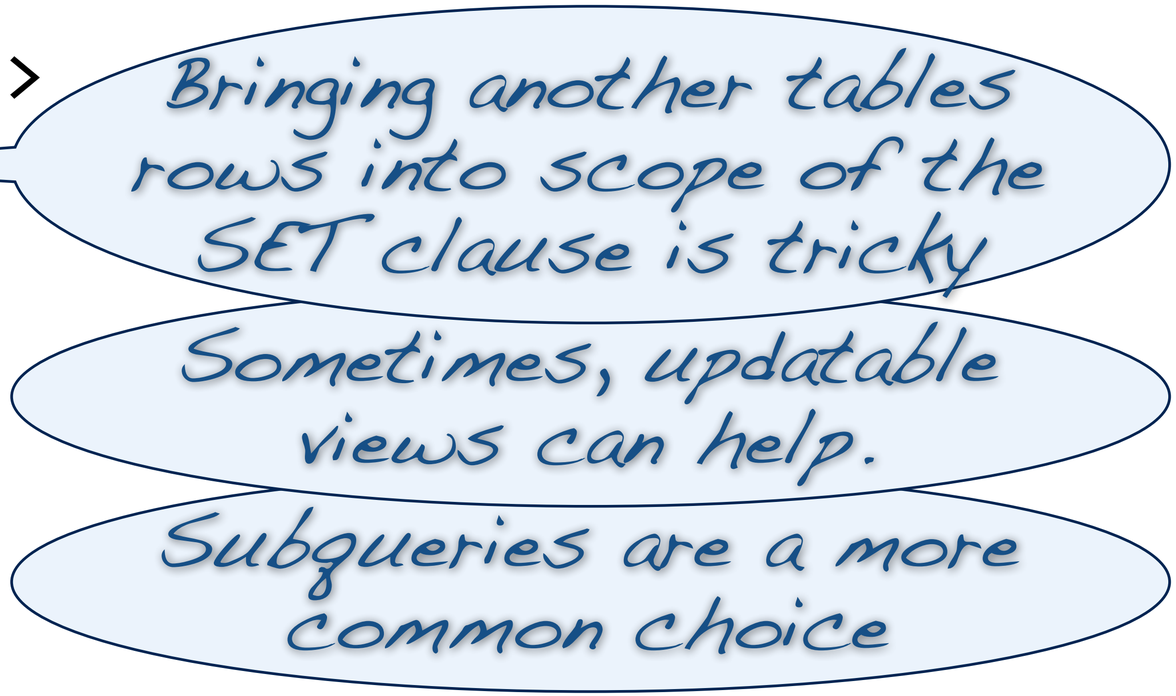


MERGE

The Problem

Updating rows from another table is awkward:

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

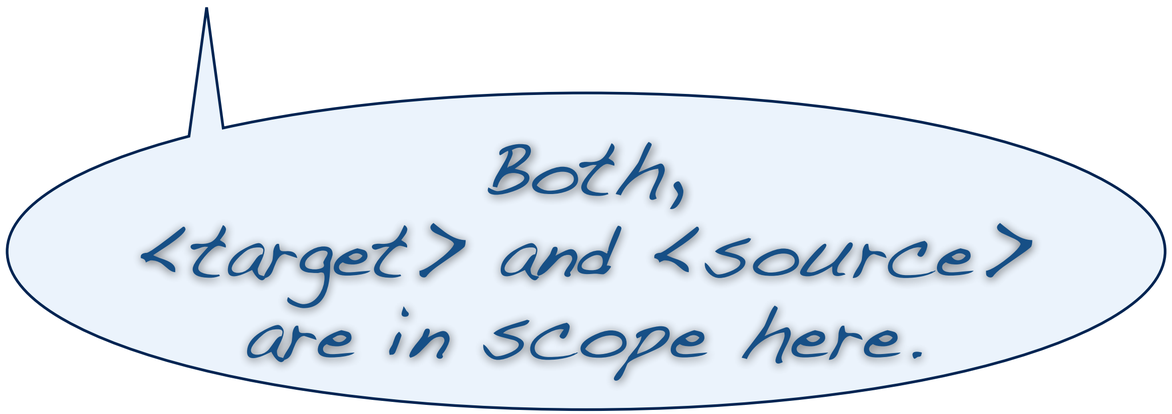


MERGE

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

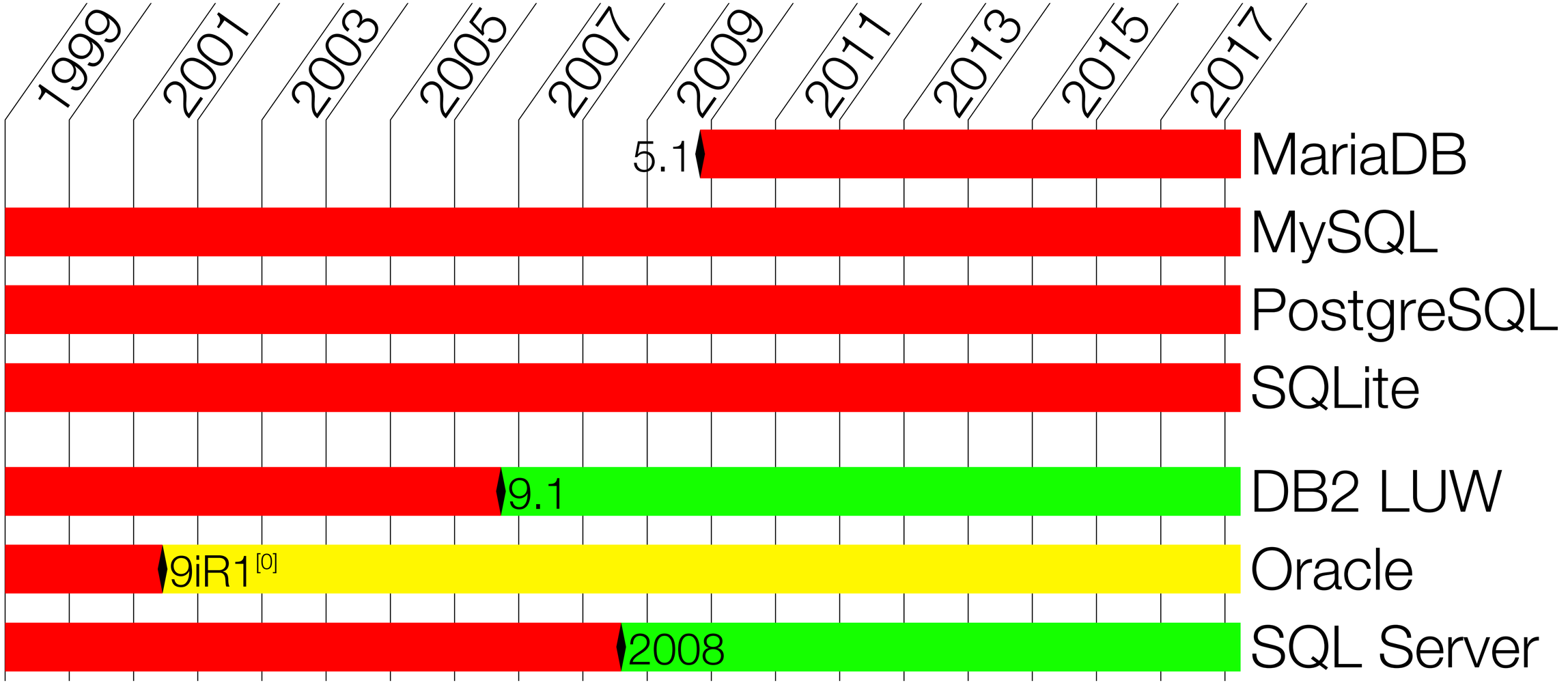
SQL:2008 introduced merge to improve this situation two-fold:

- 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.

	DB2 LUW	MariaDB	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`is [not] (true|false|unknown)`	✗	✓	✓	✗	✓	✗	✓[0]
`order by … nulls (first|last)`	✓	✗	✗	✓	✓	✗	✗
`XMLTABLE`	✓	✗	✗	✓	✓[1]	✗	✗
`PRECENTILE_(CONT|DISC)`	✓	✗	✗	✓	✓[2]	✗[3]	✗
BOOLEAN type	✗	✗[4]	✗[4]	✗	✓	✗	✗
BOOLEAN aggregates	✗	✗	✗	✗	✓[5]	✗	✗
`filter` clause	✗	✗	✗[6]	✗	✓	✗	✗
DOMAIN	✗	✗	✗	✗	✓[7]	✗	✗

[0] No `is unknown`. `Is null` can be used for this purpose.
[1] No XQuery.
[2] Not as window function.
[3] Only as window function.
[4] Boolean type known, but implemented as number.
[5] Returns `unknown` on effectively empty input. Not `some`. Not `any`. Proprietary `bool_or` (similar to `some`/`any`).
[6] The `filter_plugin` extension (3rd party) rewrites `filter` to `case` using regular expressions.
[7] `DROP CONSTRAINT` has different semantic: `CASCADE` drops columns, rather than copying the constraint.

About @MarkusWinand

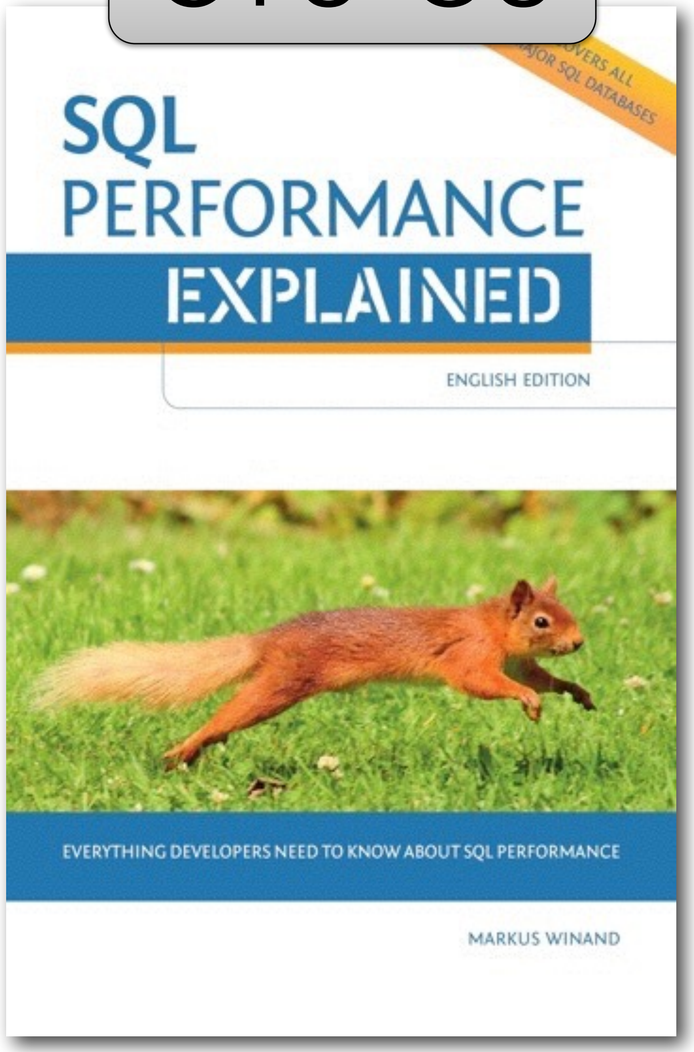
- Training for Developers
  - SQL Performance (Indexing)
  - Modern SQL
  - On-Site or Online
- SQL Tuning
  - Index-Redesign
  - Query Improvements
  - On-Site or Online

https://winand.at/

About @MarkusWinand

€0,-

€10-30



sql-performance-explained.com

About @MarkusWinand

@ModernSQL

http://modern-sql.com