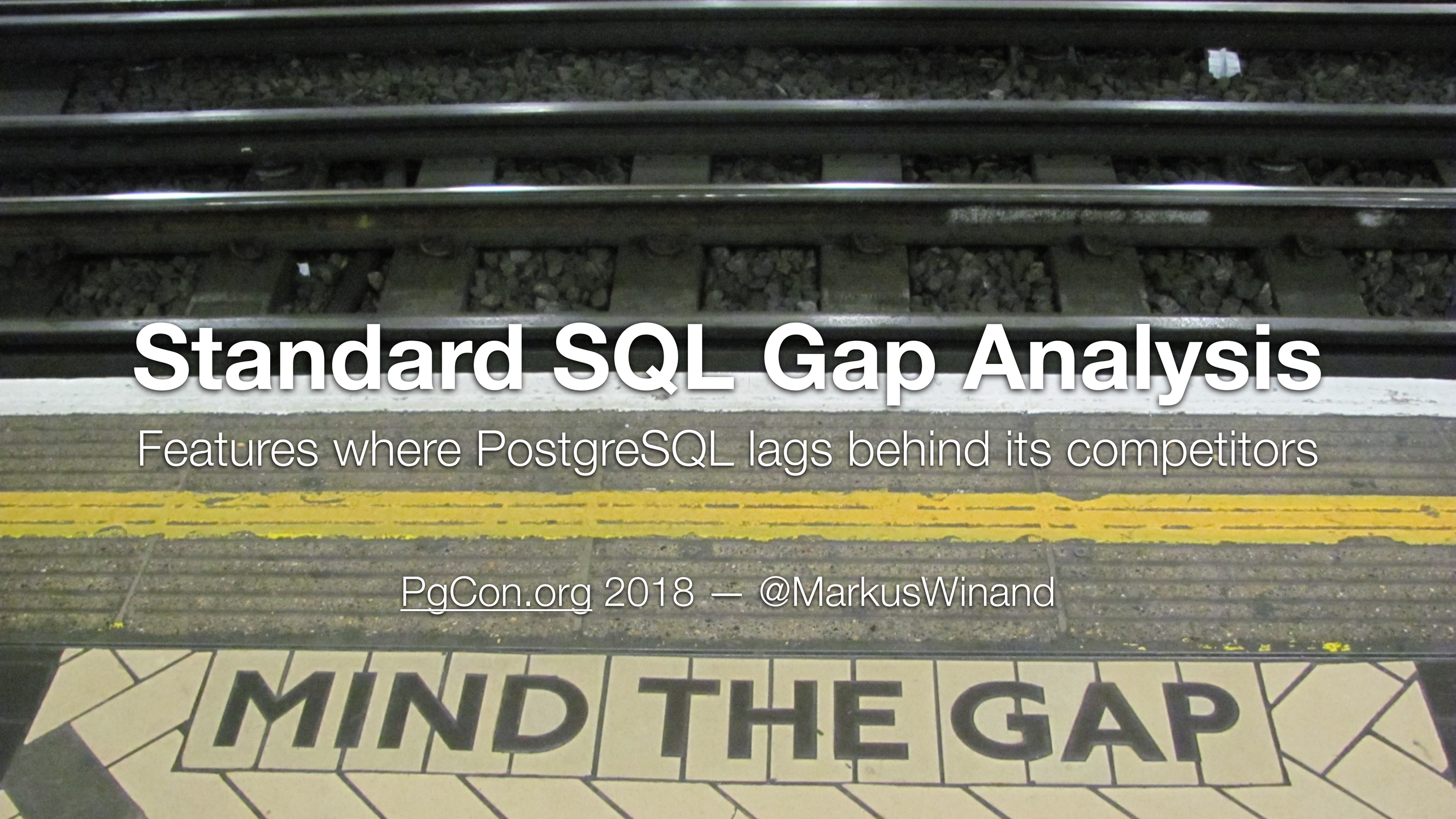


Standard SQL Gap Analysis

Features where PostgreSQL lags behind its competitors

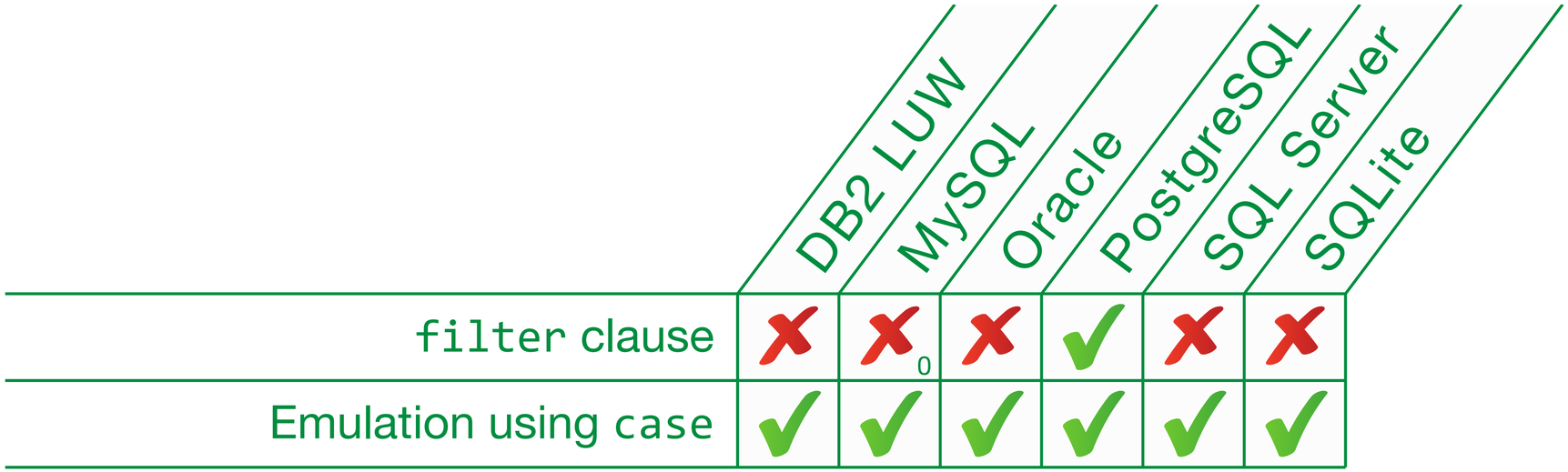
PgCon.org 2018 — @MarkusWinand

Background: Where the data comes from

I run modern-sql.com:

- Teaching “new” SQL features to developers
- Showing availability of those features in popular databases

Background: Where the data comes from



[0] The `filter_plugin` extension (3rd party) rewrites `filter` to `case` using regular expressions.

The charts are based on test cases.[0]

The test cases are created while reading ISO/IEC 9075:2016.

The level of detail for different features varies widely at the moment.

[0] Some “legacy charts” are still based on reading the docs.

One last Word

For brevity, I'm using the word

"wrong"

to mean

"not conforming to the standard".

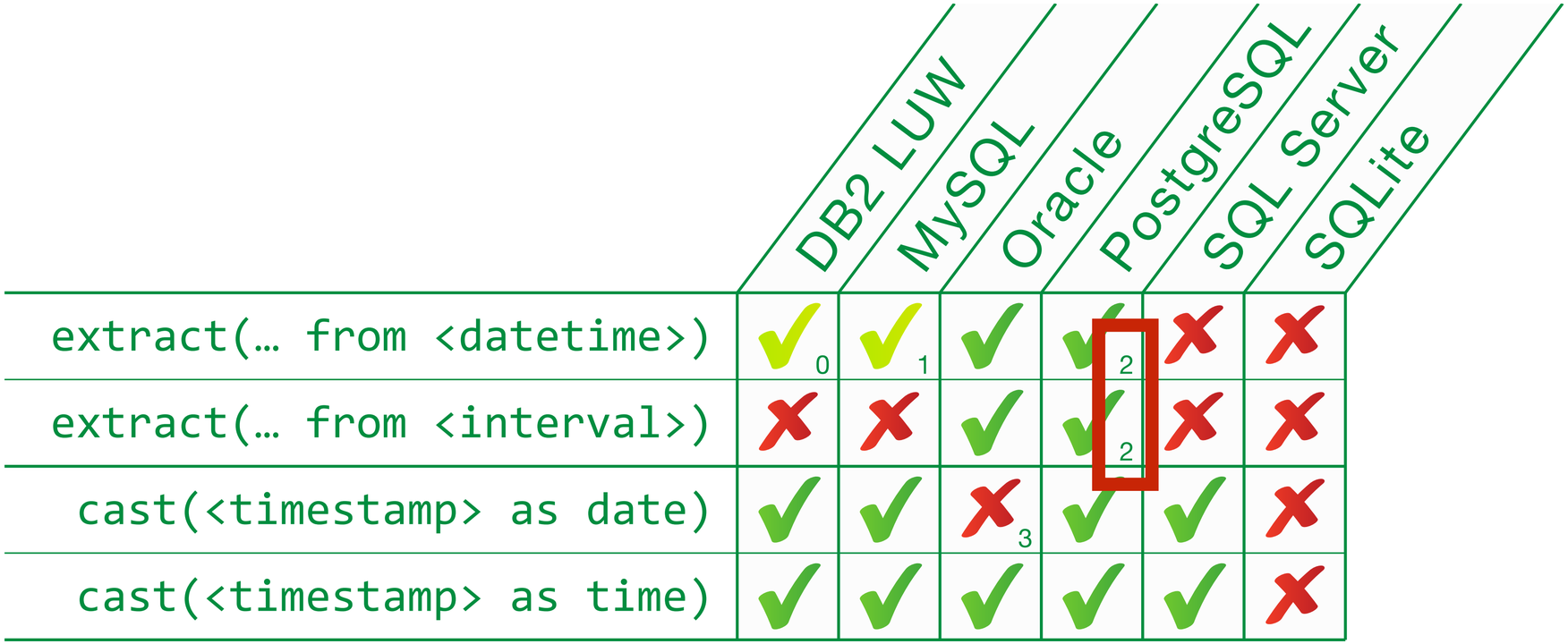
This neither implies that it is "bad" nor that it is a bug,
nor that it is worth changing.
I just means that it is not the way I understand the standard.

Less Complete or Conforming Features

EXTRACT

Get a Field from a Date or Time Value

EXTRACT: “Wrong” declared type



	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`extract(… from <datetime>)`	✓[0]	✓[1]	✓	✓[2]	✗	✗
`extract(… from <interval>)`	✗	✗	✓	✓[2]	✗	✗
`cast(<timestamp> as date)`	✓	✓	✗[3]	✓	✓	✗
`cast(<timestamp> as time)`	✓	✓	✓	✓	✓	✗

[0] No time zone fields.

[1] No time zone fields. `SECOND` does not include fractions. Use `SECOND_MICROSECOND`.

[2] Returns approximate numeric type.

[3] See “Caution: Oracle Database” above.

EXTRACT: “Wrong” declared type

`extract(`*field* `from timestamp)`	double precision
`extract(`*field* `from interval)`	double precision

extract(… from <datetime>)	✓ [0]	✓ [1]	✓	✓ [2]	✗	✗
extract(… from <interval>)	✗	✗	✓	✓ [2]	✗	✗
cast(<timestamp> as date)	✓	✓	✗ [3]	✓	✓	✗
cast(<timestamp> as time)	✓	✓	✓	✓	✓	✗

[0] No time zone fields.

[1] No time zone fields. SECOND does not include fractions. Use SECOND_MICROSECOND.

[2] Returns approximate numeric type.

[3] See “Caution: Oracle Database” above.

EXTRACT: “Wrong” declared type

`extract(`*field* `from timestamp)`	double precision
`extract(`*field* `from interval)`	double precision

7) If <extract expression> is specified, then

Case:

a) If <extract field> is a <primary datetime field> that does not specify SECOND or <extract field> is not a <primary datetime field>, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

b) Otherwise, the declared type of the result is an implementation-defined exact numeric type with scale not less than the specified or implied <time fractional seconds precision> or <interval fractional seconds precision>, as appropriate, of the SECOND <primary datetime field> of the <extract source>.

No time zone fields.

[1] No time zone fields. `SECOND` does not include fractions. Use `SECOND_MICROSECOND`.

[2] Returns approximate numeric type.

[3] See "Caution: Oracle Database" above.

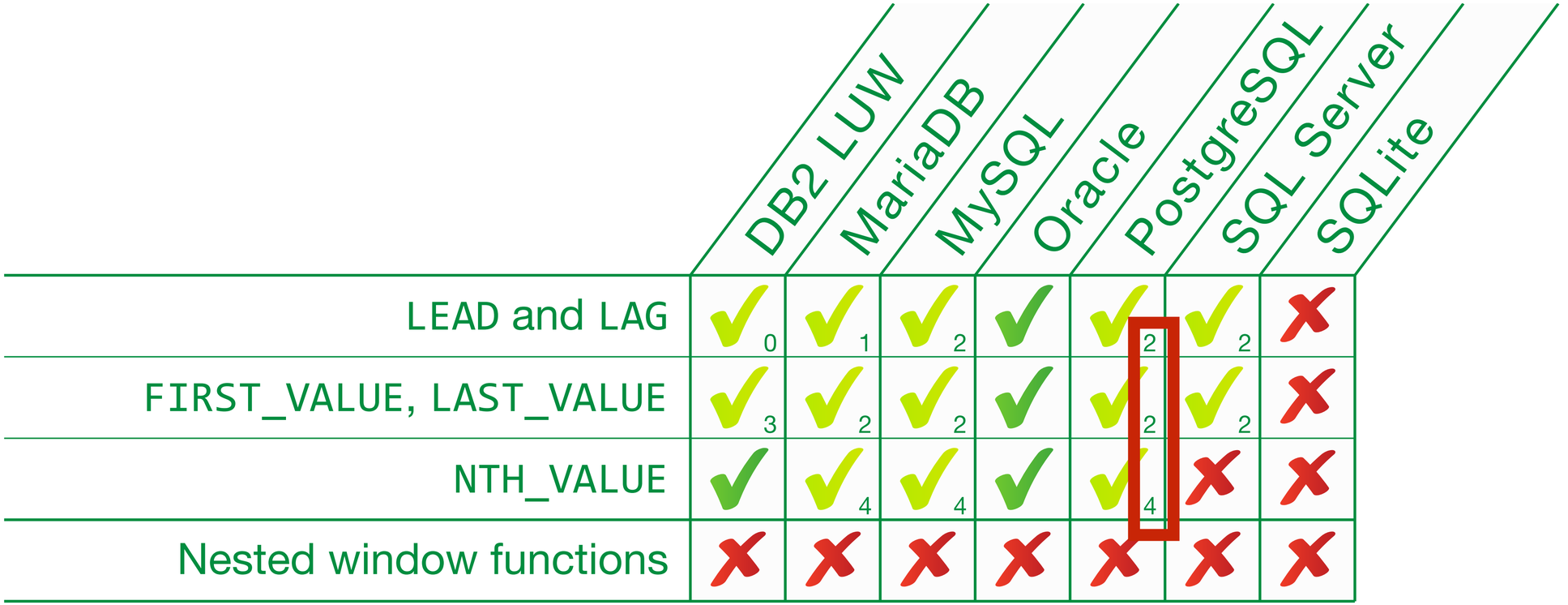
`[RESPECT|IGNORE] NULLS`

Skip over `null` values in window functions

`lead`, `lag`, `fist_value`, `last_value`, `nth_value`

(T616, T618)

Window Functions: null handling, from last

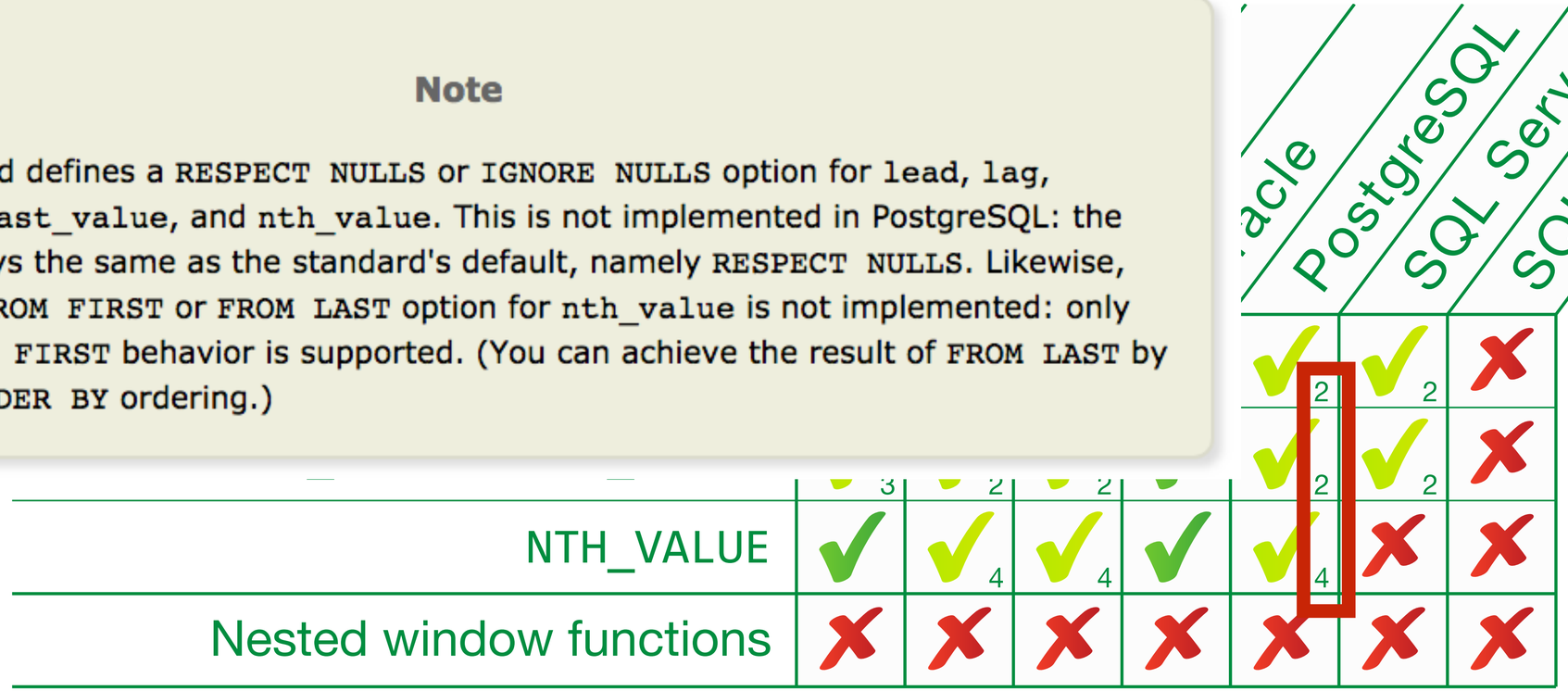


[0] No `IGNORE NULLS` Different syntax: `first_value(<expr>, 'IGNORE NULLS')` (it's a string argument)
[1] No `IGNORE NULLS` No default possible (3rd argument).
[2] No `IGNORE NULLS`
[3] No `IGNORE NULLS` Different syntax: `lead(<expr>, 1, null, 'IGNORE NULLS')` (it's a string argument)
[4] No `IGNORE NULLS`. No `FROM LAST`

Window Functions: null handling, from last

Note

The SQL standard defines a `RESPECT NULLS` or `IGNORE NULLS` option for `lead`, `lag`, `first_value`, `last_value`, and `nth_value`. This is not implemented in PostgreSQL: the behavior is always the same as the standard's default, namely `RESPECT NULLS`. Likewise, the standard's `FROM FIRST` or `FROM LAST` option for `nth_value` is not implemented: only the default `FROM FIRST` behavior is supported. (You can achieve the result of `FROM LAST` by reversing the `ORDER BY` ordering.)



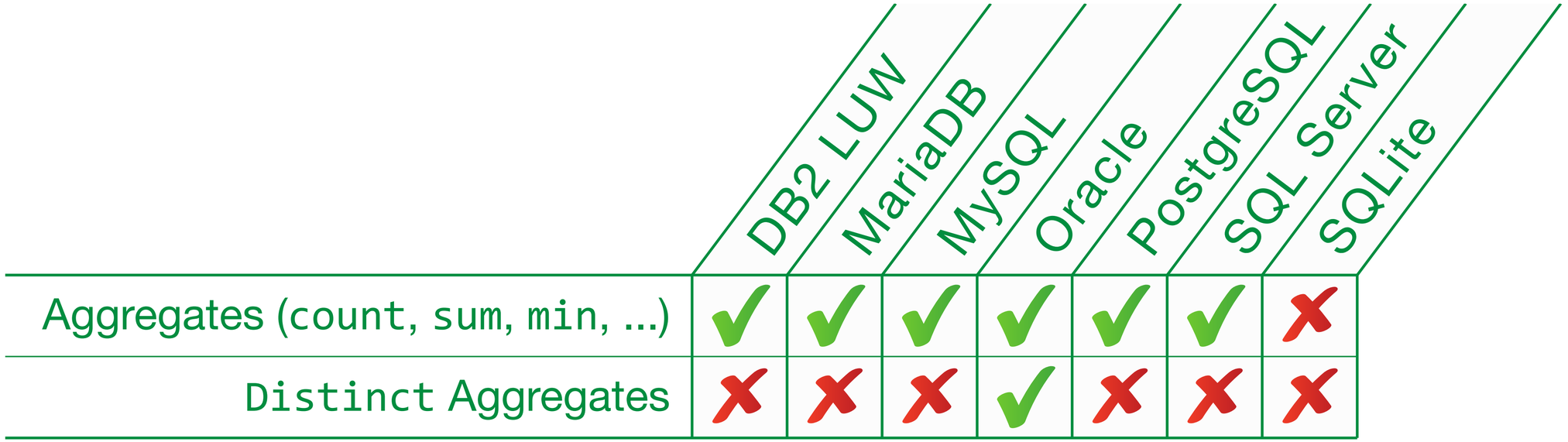
[0] No `IGNORE NULLS` Different syntax: `first_value(<expr>, 'IGNORE NULLS')` (it's a string argument)
[1] No `IGNORE NULLS` No default possible (3rd argument).
[2] No `IGNORE NULLS`
[3] No `IGNORE NULLS` Different syntax: `lead(<expr>, 1, null, 'IGNORE NULLS')` (it's a string argument)
[4] No `IGNORE NULLS`. No `FROM LAST`

`COUNT(DISTINCT …) OVER(…)`

Distinct aggregates as window function

(T611)

Window Functions: no distinct aggregates



`FETCH [FIRST|NEXT] …`

The standard's answer to **`LIMIT`**, but more options

(T866, T867)

`FETCH FIRST`: no percent, no with ties

	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
Top-level `fetch first`	✓	✗[0]	✓	✓	✗[1]	✗[0]
Subqueries with `fetch first`	✓	✗[0]	✓	✓	✗[1]	✗[0]
Top-level `fetch first` in views	✗[2]	✗[0]	✓	✓	✗[1]	✗[0]
Dynamic quantity	✓	✗	✓	✓[3]	✗	✗
`fetch first … percent`	✗	✗	✓	✗	✗[4]	✗
`fetch first … with ties`	✗	✗	✓	✗	✗[5]	✗
SQL State `2201W` if quantity < 1	✓[6]	✗	✗	✓[6]	✗	✗

[0] Use proprietary `limit`
[1] Use proprietary `top`
[2] Use nested query: `CREATE VIEW … AS SELECT … FROM (SELECT … FROM … FETCH FIRST …) t`
[3] Requires parenthesis: `(?)`
[4] Use proprietary `select top … percent`
[5] Use proprietary `select top … with ties`
[6] Not for `0` (zero)

`FETCH FIRST`: no percent, no with ties

Docs: unsupported features:

F866		FETCH FIRST clause: PERCENT option	
F867		FETCH FIRST clause: WITH TIES option	

	LUW	SQL	cle	tgreSQL	Server	Lite
Top-level `fetch first` in views	✗[2]	✗[0]	✓	✓	✗[1]	✗[0]
Dynamic quantity	✓	✗	✓	✓[3]	✗	✗
`fetch first … percent`	✗	✗	✓	✗	✗[4]	✗
`fetch first … with ties`	✗	✗	✓	✗	✗[5]	✗
SQL State `2201W` if quantity < 1	✓[6]	✗	✗	✓[6]	✗	✗

[0] Use proprietary `limit`
[1] Use proprietary `top`
[2] Use nested query: `CREATE VIEW … AS SELECT … FROM (SELECT … FROM … FETCH FIRST …) t`
[3] Requires parenthesis: `(?)`
[4] Use proprietary `select top … percent`
[5] Use proprietary `select top … with ties`
[6] Not for `0` (zero)

Functional Dependencies

(T301)

Functional dependencies: only simplest cases

	DB2 LUW	MariaDB	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
Base table `PRIMARY KEY`	✗	✗	✓	✗	✓	✗	✗
Base table `UNIQUE`	✗	✗	✓	✗	✗	✗	✗
Joined tables	✗	✗	✓	✗	✓[0]	✗	✗
`WHERE` clause	✗	✗	✓	✗	✗	✗	✗
`GROUP BY` clause	✗	✗	✓	✗	✗	✗	✗

[0] Not following joins to `PRIMARY KEY`s or `UNIQUE` constraints

Docs: unsupported features:

T301		Functional dependencies	partially supported

Functional dependencies: only simplest cases

```
SELECT COUNT(*) cnt, t2.b
  FROM t1
 INNER JOIN t2 ON (t1.pk = t2.pk)
 GROUP BY t1.pk
```

	DB2 LUW	MariaDB	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
Base table PRIMARY KEY	✗	✗	✓	✗	✓	✗	✗
Base table UNIQUE	✗	✗	✓	✗	✗	✗	✗
Joined tables	✗	✗	✓	✗	✓[0]	✗	✗
WHERE clause	✗	✗	✓	✗	✗	✗	✗
GROUP BY clause	✗	✗	✓	✗	✗	✗	✗

[0] Not following joins to PRIMARY KEYs or UNIQUE constraints

Docs: unsupported features:

T301		Functional dependencies	partially supported

Functional dependencies: only simplest cases

Functional dependencies: only simplest cases

Still room for vendor extensions.
e.g. related to `ROW_NUMBER` and `ORDINALITY`.

Unsupported features that other DBs have

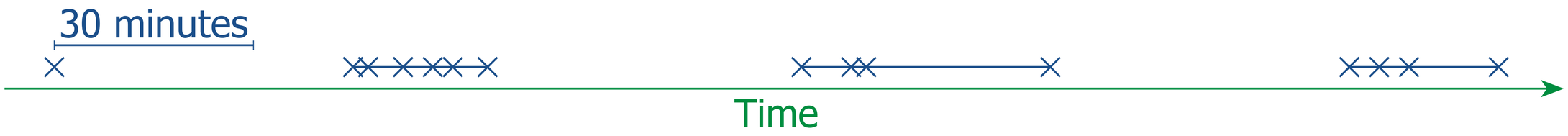
Row Pattern Recognition

(`match_recognize`)

(R010, R020, R030)

Row Pattern Matching

Since SQL:2016

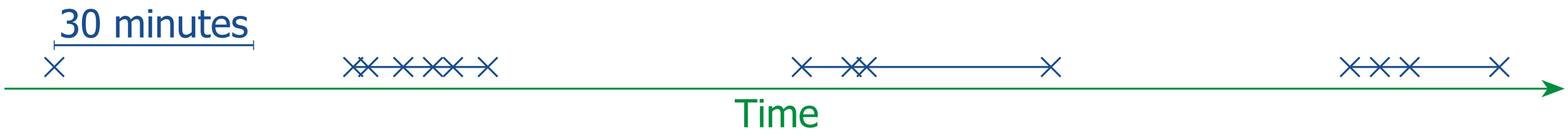



```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( any cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL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

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

Row Pattern Matching

Since SQL:2016



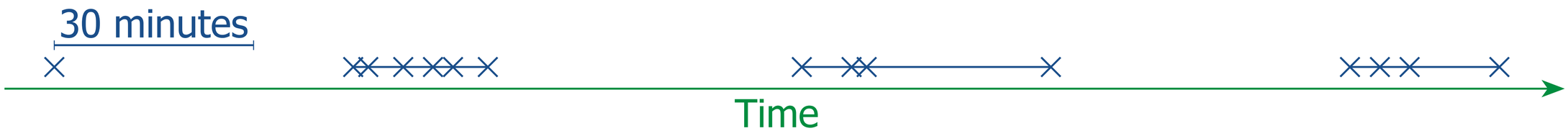

```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( any cont* )
        DEFINE cont AS ts < PREV(ts)
                       + INTERVAL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

define continued

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

Row Pattern Matching

Since SQL:2016



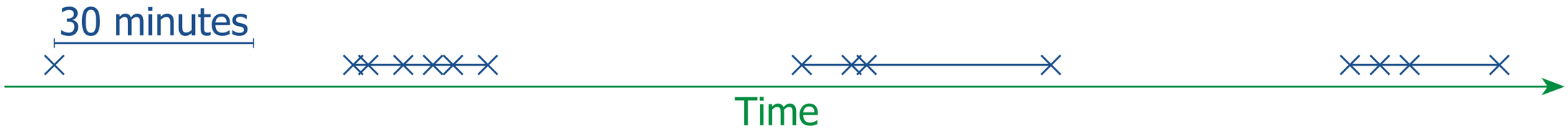

```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( any cont* )
        DEFINE cont AS ts < PREV(ts)
                       + INTERVAL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



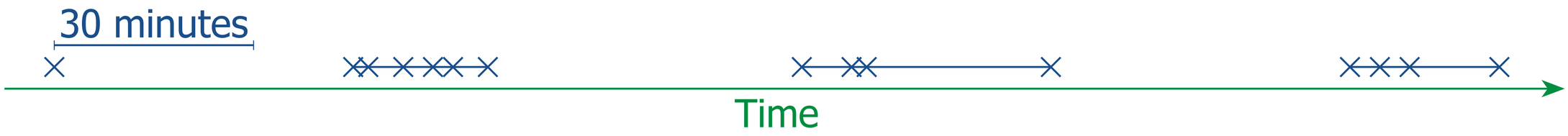

```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( any cont* )
        DEFINE cont AS ts < PREV(ts)
                       + INTERVAL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



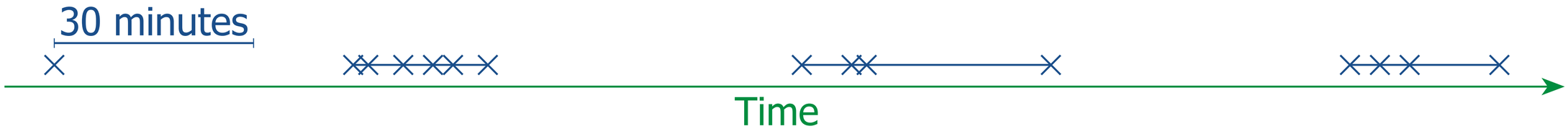

```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( any cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL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



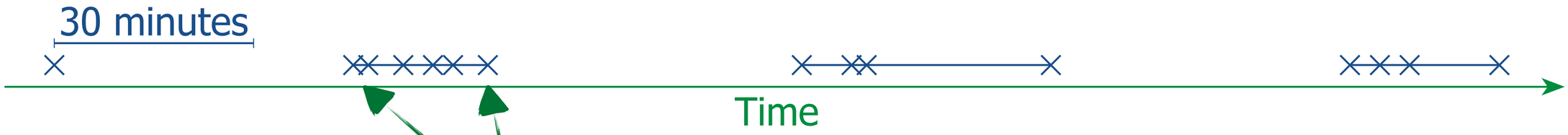

```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( any cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL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



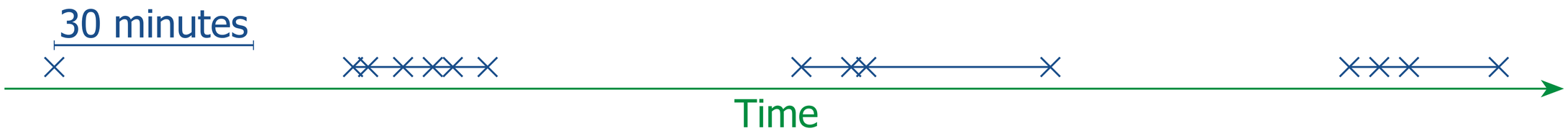

```
SELECT COUNT(*) sessions
     , AVG(duration) avg_duration
  FROM log
       MATCH_RECOGNIZE(
        ORDER BY ts
        MEASURES
         LAST(ts) - FIRST(ts) AS duration
        ONE ROW PER MATCH
        PATTERN ( any cont* )
        DEFINE cont AS ts < PREV(ts)
                          + INTERVAL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

Same as any.ts

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 ( any 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

Endless possibilites

`GROUP BY`
- ➡ `ONE ROW PER MATCH`

`OVER ()`
- ➡ `ALL ROWS PER MATCH, FINAL, RUNNING`

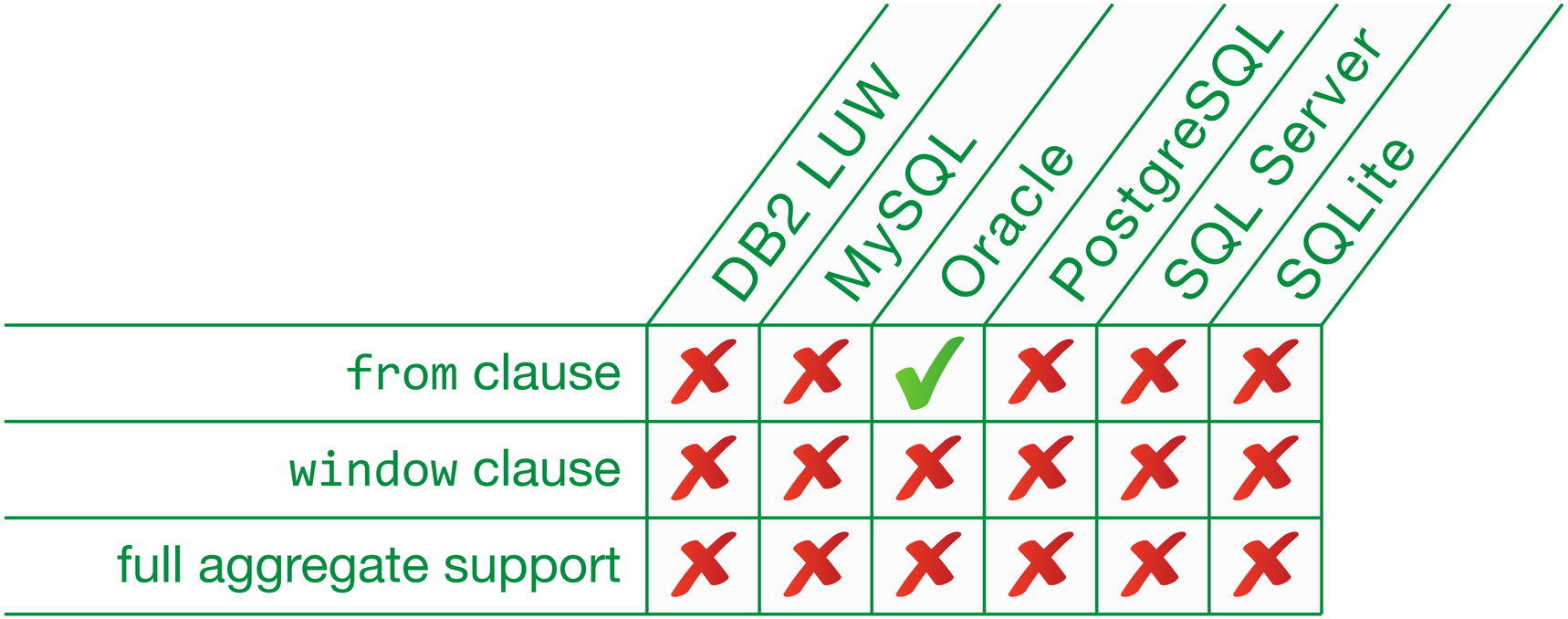
`HAVING, WHERE`
- ➡ `PATTERN` (unmatched, suppressed `{- … -}`)

Mixing `GROUP BY` and `OVER()`
- ➡ `ALL ROWS PER MATCH` + all-but-one rows suppressed

Data-driven match length
- ➡ `SUM`, `COUNT`, … in `DEFINE`

Duplicating rows (to some extend)
- ➡ `ALL ROWS PER MATCH` + `AFTER MATCH SKIP TO …`

Row pattern matching — `match_recognize`



Free technical report by ISO:

http://standards.iso.org/ittf/PubliclyAvailableStandards/c065143_ISO_IEC_TR_19075-5_2016.zip

Row Pattern Matching

Since SQL:2016

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

Stew Ashton has a lot material on this too:
https://stewashton.wordpress.com/category/match_recognize/

Temporal and bi-temporal tables

(T180, T181)

Temporal and bi-temporal tables

First appeared in SQL:2011.

There is an excellent free paper on it:

Temporal features in SQL:2011

https://sigmodrecord.org/publications/sigmodRecord/1209/pdfs/07.industry.kulkarni.pdf

If you don’t have access to the standard,
this is the next best resource on it.

Temporal and bi-temporal tables

There are two versioning features:

➡ System Versioning

Mostly transparent (done by the system).

Models when changes happened in the DB.

➡ Application Versioning

Managed by the application (with SQL support).

Can model when changes happened in the real world.

Both can be applied on per table level as needed.

Temporal and bi-temporal tables

Both require explicit datetime columns and a period:

➡ System Versioning

Generated columns
`GENERATED ALWAYS`

Period name fixed:
`SYSTEM_TIME`

➡ Application Versioning

Arbitrary columns

Arbitrary period names
(but only one per table)

Temporal and bi-temporal tables

System versioning takes care of the DMLs.

➡ <u>System Versioning</u>

Datetime columns visible (not 100% transparent)[0]

User cannot set them.

Constraints remain unchanged.

➡ <u>Application Versioning</u>

Datetime columns visible

User has to provide values.

Constraints need to consider periods (e.g. `WITHOUT OVERLAPS`).

[0] Some databases offer invisible or hidden columns for transparency.

Temporal and bi-temporal tables

For queries, they use a different syntax:

➡ System Versioning

```
FROM …
FOR SYSTEM_TIME
[AS OF|BETWEEN|FROM…TO]
```

➡ Application Versioning

In `where` clause.

New predicates for periods: `contains`, `overlaps`, `precedes`, `succeeds`,…

Temporal and bi-temporal tables

Recent discussions on -hackers:

➡ System Versioning

"AS OF Queries"
Konstantin Knizhnik
Dec 2017 - Jan 2018

➡ Application Versioning

"Periods"
Vik Fearing
May 2018

System-versioned tables

(T180)

System-versioned tables

Released
May 25, 2018

	DB2 LUW	MariaDB	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`generate always as row …`	✓[0]	✓	✗	✗	✗	✓	✗
`period for system_time`	✓[1]	✓	✗	✗	✗	✓	✗
Add system versioning to table	✓[2]	✓[2]	✗	✗	✗	✓[2]	✗
Drop system versioning from table	✓[2]	✓[2]	✗	✗	✗	✓[2]	✗
`for system_time as of …`	✓	✓	✗	✗	✗	✓[3]	✗
`for system_time between …`	✓[4]	✓[4]	✗	✗	✗	✓[5]	✗
`for system_time from …`	✓	✓	✗	✗	✗	✓[3]	✗
Immutable transaction time	✓	✗[6]	✗	✗	✗	✓	✗

[0] Requires `row begin` instead of `row start`
[1] Without keyword `for` (`period system_time (…)`)
[2] Syntax varies widely
[3] Expressions not supported.
[4] Without `between symmetric`
[5] Expressions not supported. Without `between symmetric`
[6] `Row [start|end]` uses statement time, not transaction time.

Oracle has similar (yet different) syntax to access undo data (“flashback”).

System-versioned tables

Limitations and gaps in the standard:

➡ Schema changes are not supported
(Most `ALTER` statements on system-versioned tables fail)

➡ No functionality for retention
(also: `delete` cannot delete historic rows—GDPR right of erasure ;)

➡ `FOR SYSTEM_TIME` only works for base tables
(not for views, for example. Also no session setting in the standard).

➡ Based on “transaction time” (!= commit time)

System-versioned tables

Notes from current implementations:

➡ **History tables** are most popular
Db2 (LUW) and SQL Server use separate tables for old data.

➡ **Partitions** let the user choose
MariaDB 10.3 use a single logical table that can optionally be partitioned so that current and historic data are segregated.

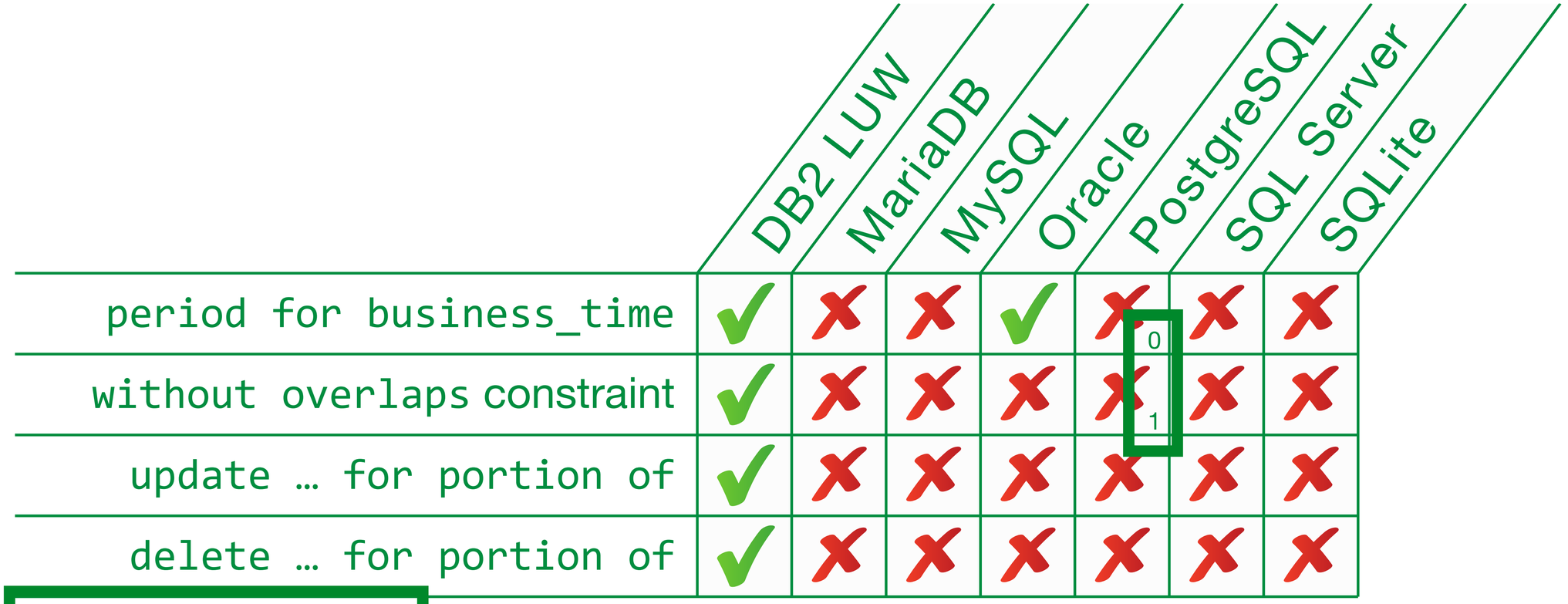
➡ Finding history data in **UNDO** (data kept for rollback)
Oracle uses the UNDO segment to access historic data.
Automatic retention, configurable up to 2^{32} seconds (136yrs)[0].

[0] Don't know if there is a way to retire selected rows (GDPR)

Application-versioned tables

(T181)

Application-versioned tables — model the real world



[0] Use range type.
[1] Use exclusion constraint.

Functionality is available, only the standard syntax is missing

“Periods” Patch from May 26 2018

Period Predicates

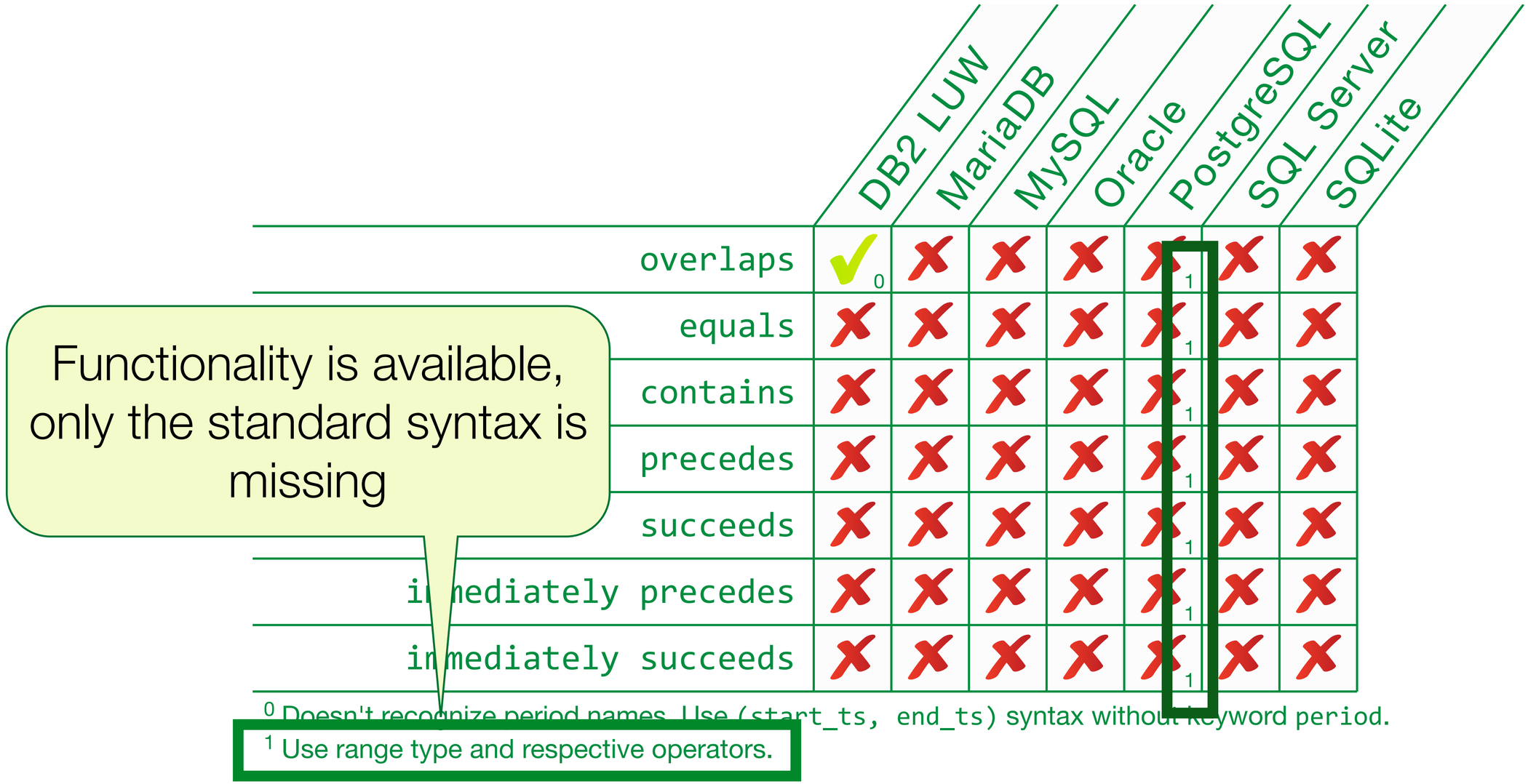
(T502)

Period Predicates — like range type operators

	DB2 LUW	MariaDB	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`overlaps`	✓[0]	✗	✗	✗	✗[1]	✗	✗
`equals`	✗	✗	✗	✗	✗[1]	✗	✗
`contains`	✗	✗	✗	✗	✗[1]	✗	✗
`precedes`	✗	✗	✗	✗	✗[1]	✗	✗
`succeeds`	✗	✗	✗	✗	✗[1]	✗	✗
`immediately precedes`	✗	✗	✗	✗	✗[1]	✗	✗
`immediately succeeds`	✗	✗	✗	✗	✗[1]	✗	✗

[0] Doesn't recognize period names. Use `(start_ts, end_ts)` syntax without keyword `period`.
[1] Use range type and respective operators.

Period Predicates — like range type operators

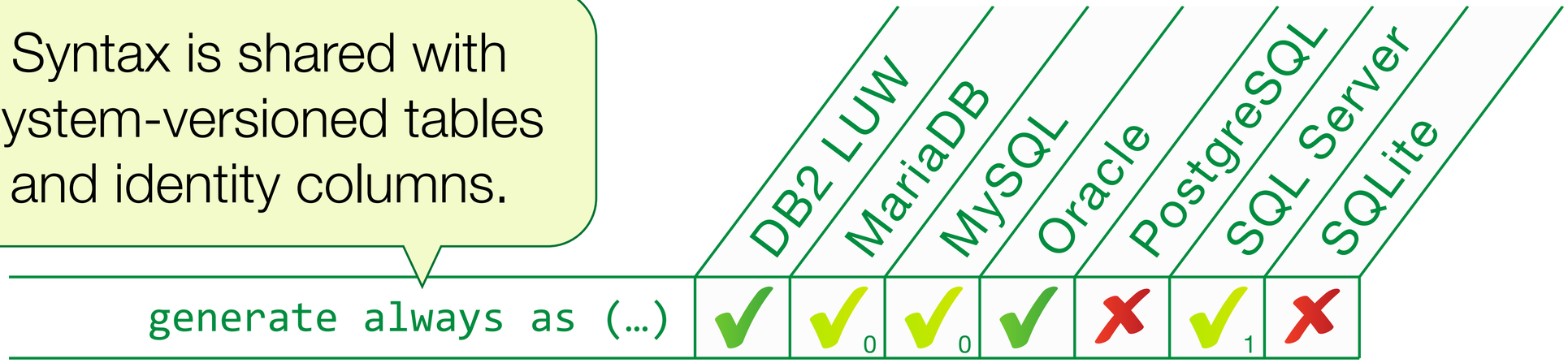


Generated Columns

(T175)

Generated Columns

Syntax is shared with system-versioned tables and identity columns.



[0] Requires data type declaration.
[1] Requires data type declaration. Without keywords `generated always`.

From standards perspective:

➡ Generated columns can be used almost like base columns (e.g. in constraint definitions)

Other use cases:

➡ Function-based indexes (MariaDB/MySQL, SQL Server)

Combined data change and retrieval

Similar to writeable CTEs

(T495)

Combined Data Change and Retrieval

```
INSERT INTO target
SELECT *
  FROM OLD TABLE (DELETE FROM source)
```

```
WITH cte AS (
     DELETE FROM source
     RETURNING *
)
INSERT INTO target
SELECT *
  FROM cte
```

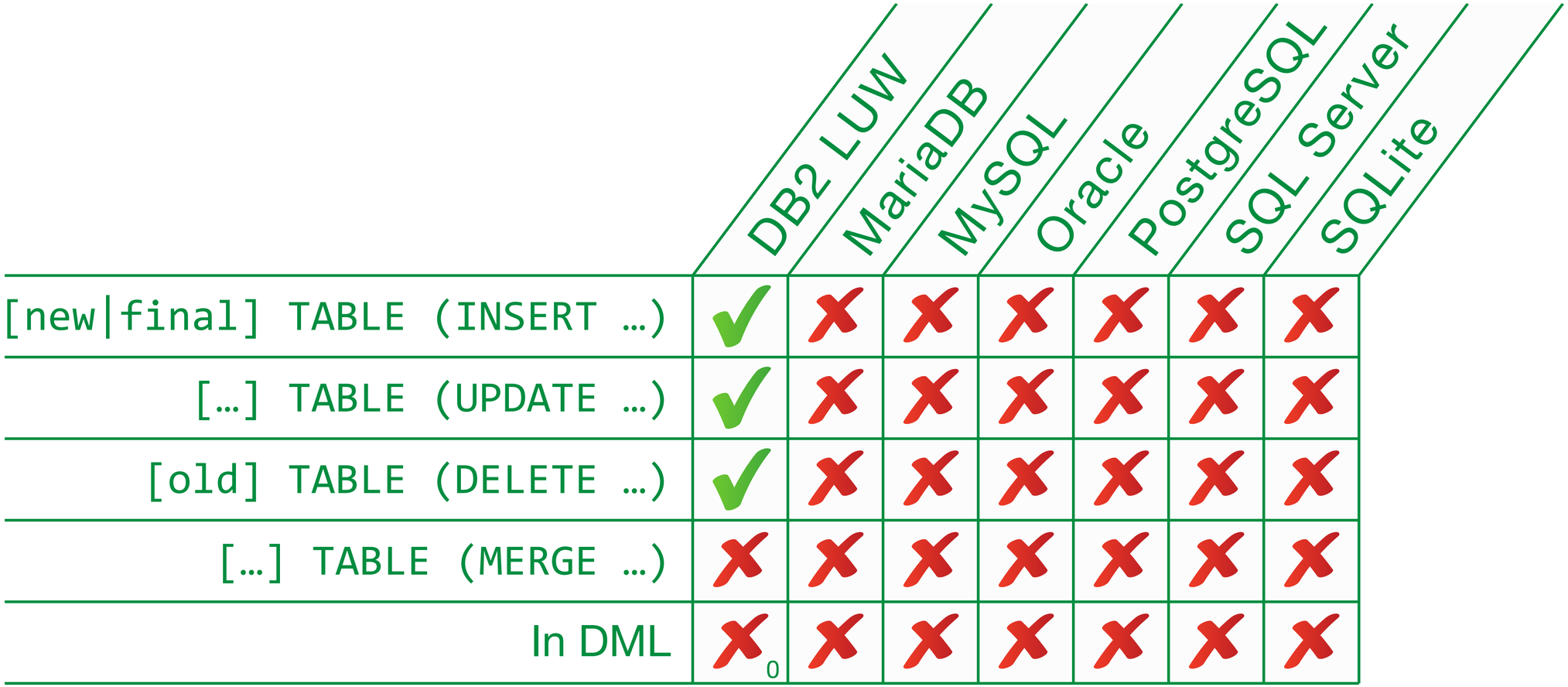
Combined Data Change and Retrieval

```
INSERT INTO demo_t495_c
SELECT *
  FROM OLD TABLE (DELETE FROM demo_t495)
```

Differences to writeable CTEs:

- ➡ Three modes: `OLD`, `NEW`, `FINAL` (similar to triggers)
- ➡ `NEW` and `FINAL` is still before `AFTER` triggers
- ➡ `FINAL` fails in case the target table is further modified by
  - ➡ constraints (cascade)
  - ➡ `AFTER` triggers

Combined Data Change and Retrieval



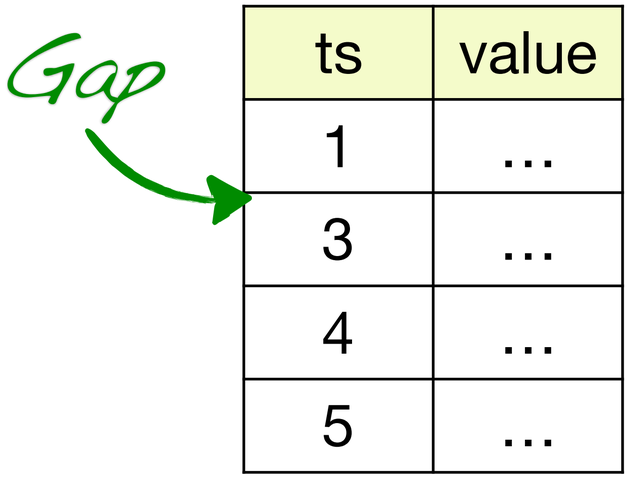
	DB2 LUW	MariaDB	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`[new|final] TABLE (INSERT …)`	✓	✗	✗	✗	✗	✗	✗
`[…] TABLE (UPDATE …)`	✓	✗	✗	✗	✗	✗	✗
`[old] TABLE (DELETE …)`	✓	✗	✗	✗	✗	✗	✗
`[…] TABLE (MERGE …)`	✗	✗	✗	✗	✗	✗	✗
In DML	✗[0]	✗	✗	✗	✗	✗	✗

[0] Main statement must be `select`. Workaround via chained `with` clause.

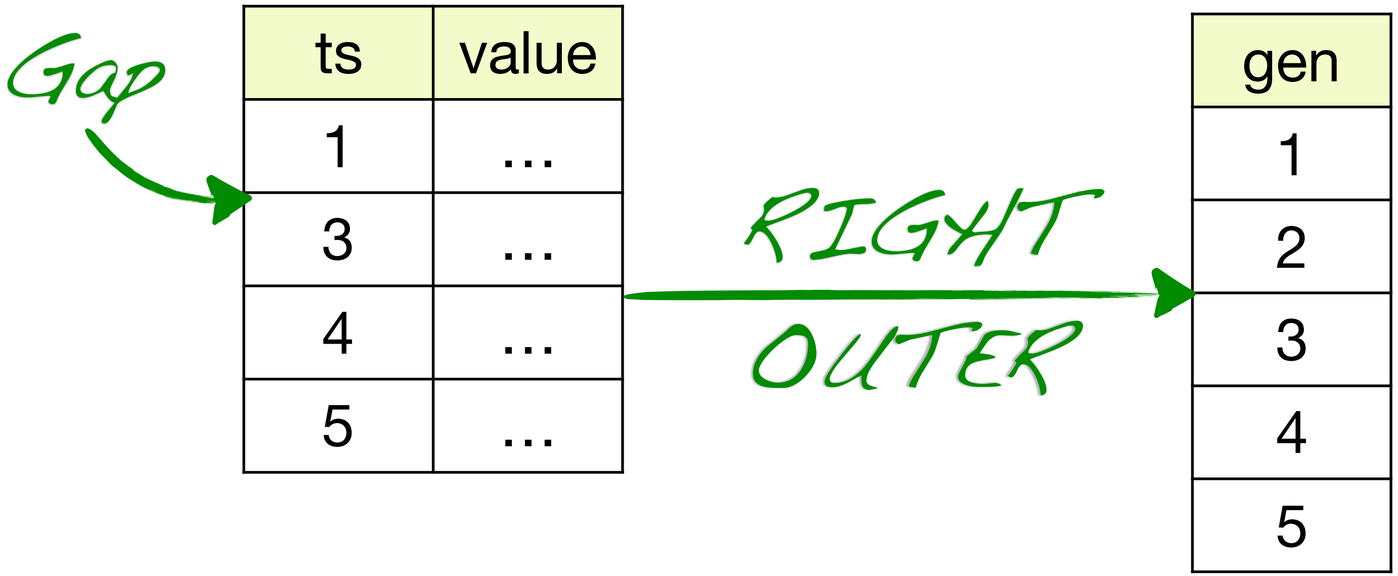
Partitioned Join
(not related to partitioned tables)

(F403)

Partitioned Join — Filling gaps in time series

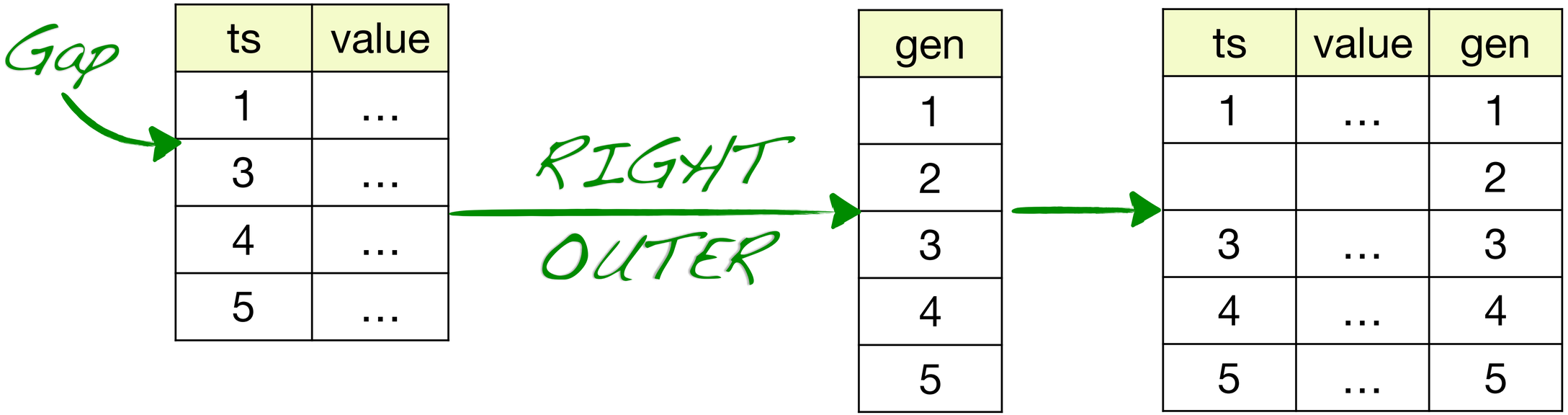


Partitioned Join — Filling gaps in time series




```
SELECT *
  FROM data
 RIGHT JOIN generate_series(…)
    ON …
```

Partitioned Join — Filling gaps in time series




```
SELECT *
  FROM data
 RIGHT JOIN generate_series(…)
    ON …
```

Partitioned Join — Filling gaps in time series

grp	ts	value
A	1	…
A	3	…
A	4	…
A	5	…
B	2	…
B	4	…
B	5	…

What if you have several time series, all of them to be padded?

```
SELECT *
  FROM (SELECT DISTINCT grp
          FROM data) dist
 CROSS JOIN LATERAL
       (SELECT *
          FROM data
         RIGHT JOIN generate_series(…)
            ON …
           AND data.grp = dist.grp
       )
```

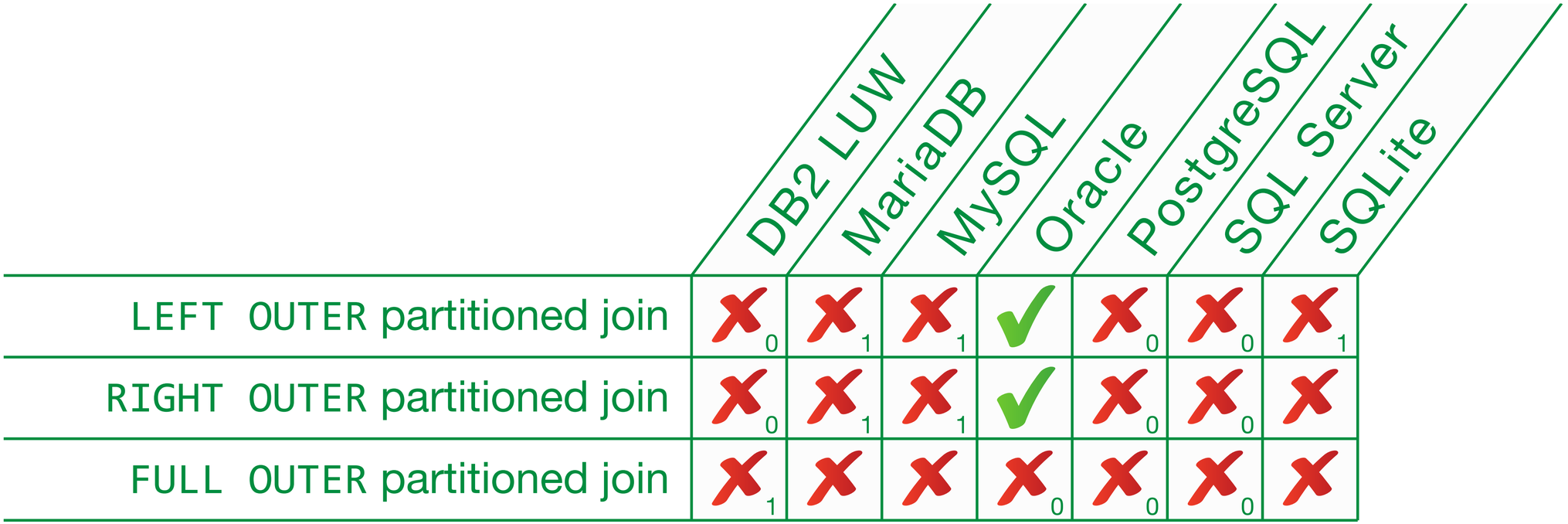
Partitioned Join — Filling gaps in time series

grp	ts	value
A	1	…
A	3	…
A	4	…
A	5	…
B	2	…
B	4	…
B	5	…

What if you have several time series, all of them to be padded?

```
SELECT *
  FROM data PARTITION BY (key)
 RIGHT JOIN generate_series(…)
         ON …
```

Partitioned Join — Filling gaps in time series



	DB2 LUW	MariaDB	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`LEFT OUTER` partitioned join	✗[0]	✗[1]	✗[1]	✓	✗[0]	✗[0]	✗[1]
`RIGHT OUTER` partitioned join	✗[0]	✗[1]	✗[1]	✓	✗[0]	✗[0]	✗
`FULL OUTER` partitioned join	✗[1]	✗	✗	✗[0]	✗[0]	✗[0]	✗

[0] Alternative: Select distinct partition key and `join lateral` for each partition.

[1] Alternative: join to cross join of distinct partition key and gap-filler.

LISTAGG

Like `STRING_AGG`

(T625)

LISTAGG

	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`listagg(…) within group (…)`	✓	✗	✓	✗	✗	✗
`listagg(… on overflow …)`	✗	✗	✓[0]	✗	✗	✗
`listagg(distinct …)`	✓[1]	✗	✗	✗	✗	✗
`SQLSTATE 22001 on truncation`	✗[2]	✗	✗[3]	✗	✗	✗
`listagg with grouping sets`	✓	✗	✓	✗	✗	✗
`stagg… within group… filter…`	✗	✗	✗	✗	✗	✗
`listagg… within group… over…`	✗	✗	✓	✗	✗	✗

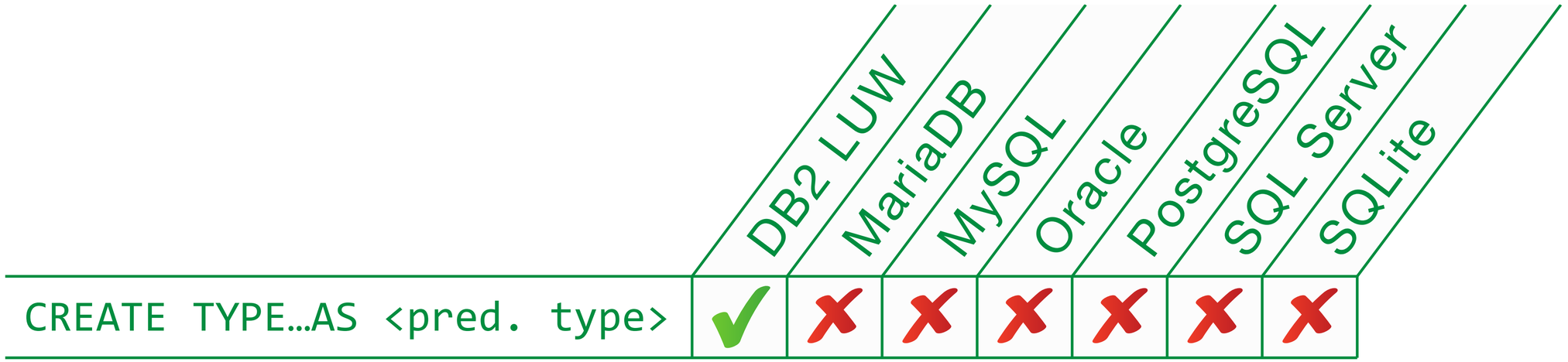
[0] Since 12.2
[1] If ordered by the aggregated values: `listagg(distinct X,…) within group (order by X)`
[2] `SQLSTATE 54006`
[3] `SQLSTATE 72000`

Distinct data types

`CREATE TYPE … AS <predefined types>`

(S011 - Core SQL)

Distinct Data Types

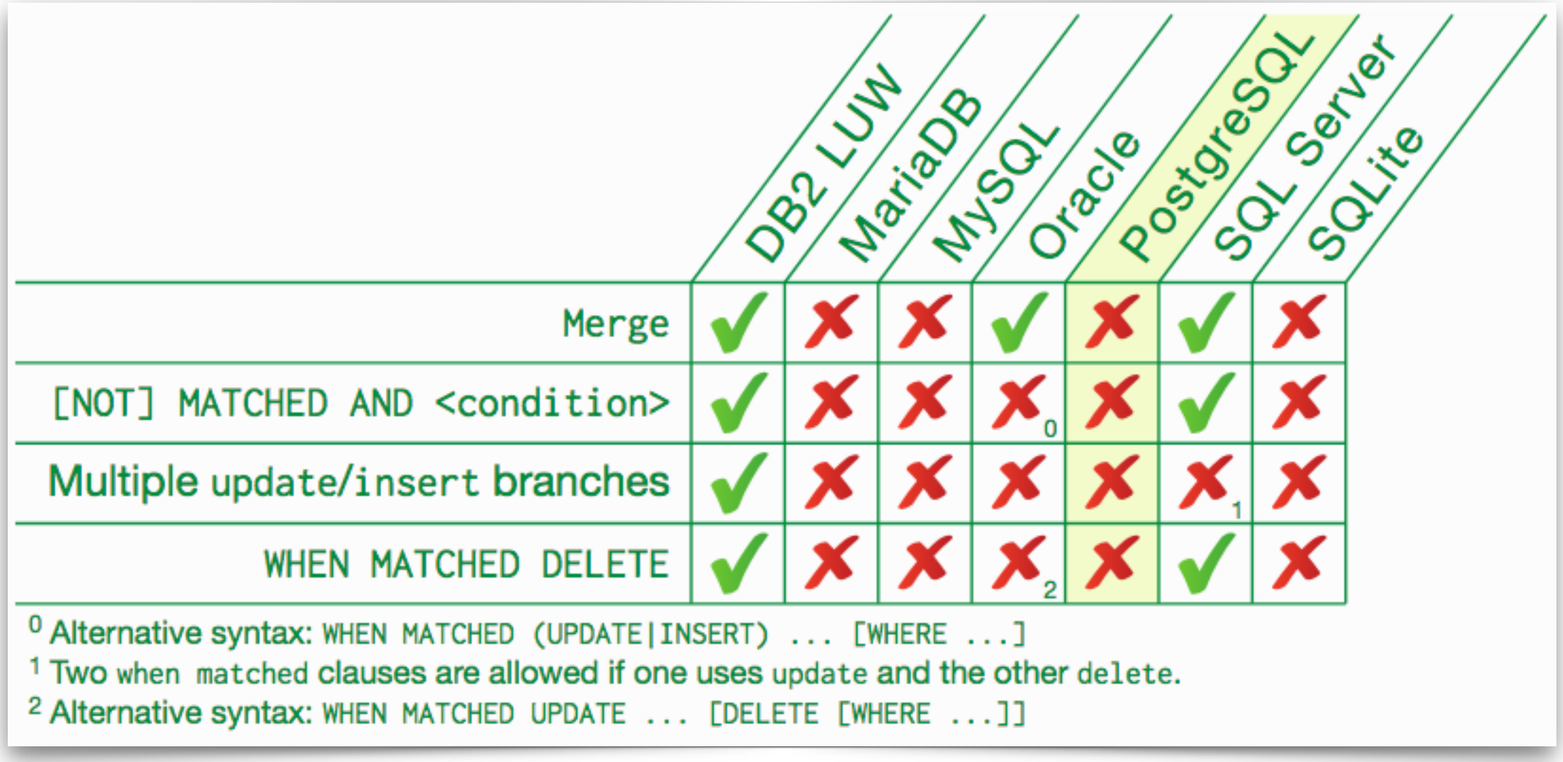


Work in progress

MERGE

(F312, F313, F314)

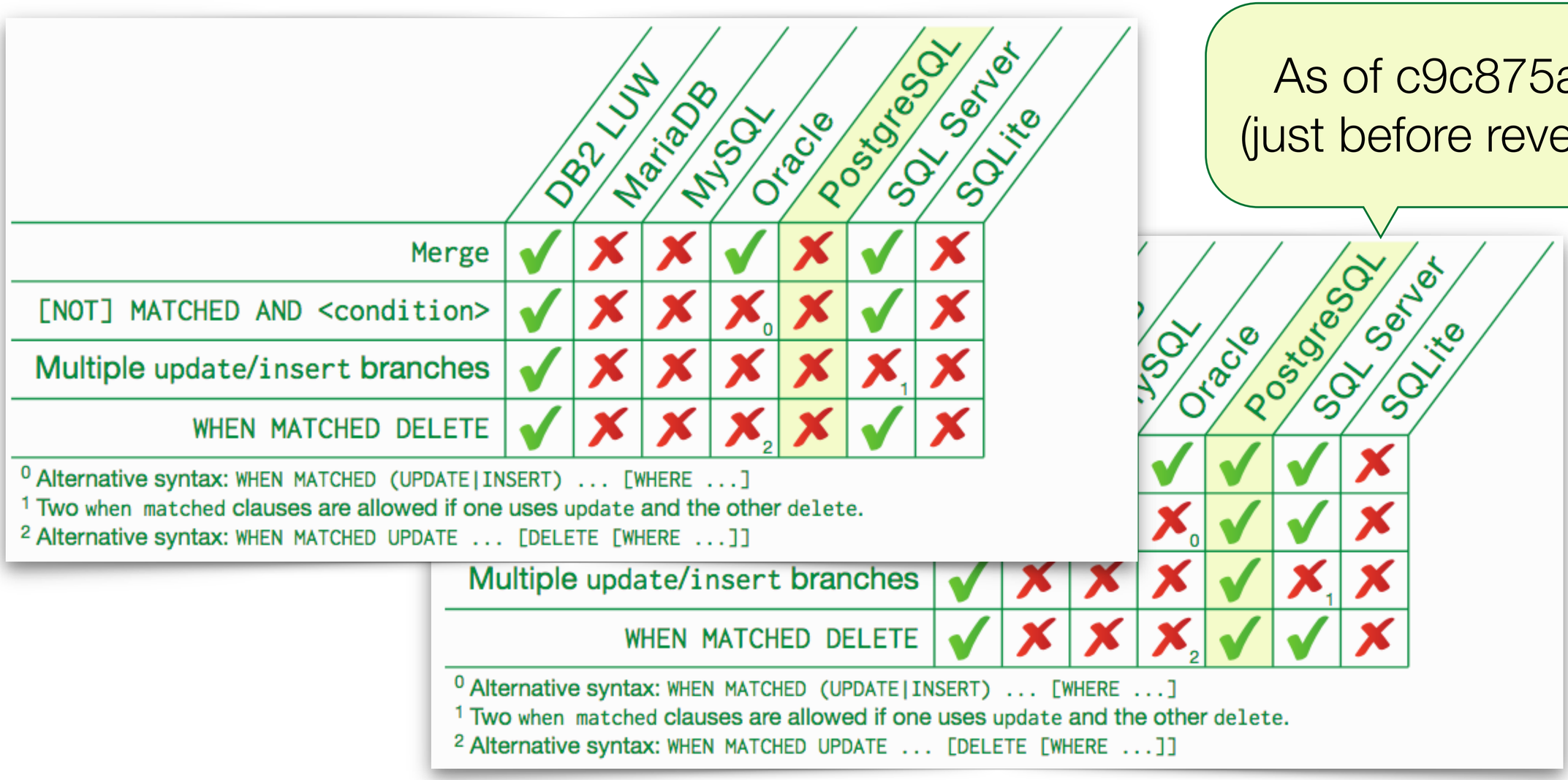
MERGE — conditional insert/update/delete



	DB2 LUW	MariaDB	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
Merge	✓	✗	✗	✓	✗	✓	✗
[NOT] MATCHED AND <condition>	✓	✗	✗	✗[0]	✗	✓	✗
Multiple update/insert branches	✓	✗	✗	✗	✗	✗[1]	✗
WHEN MATCHED DELETE	✓	✗	✗	✗[2]	✗	✓	✗

[0] Alternative syntax: `WHEN MATCHED (UPDATE|INSERT) ... [WHERE ...]`
[1] Two `when matched` clauses are allowed if one uses `update` and the other `delete`.
[2] Alternative syntax: `WHEN MATCHED UPDATE ... [DELETE [WHERE ...]]`

MERGE — conditional insert/update/delete



JSON

(T811–T838)

JSON

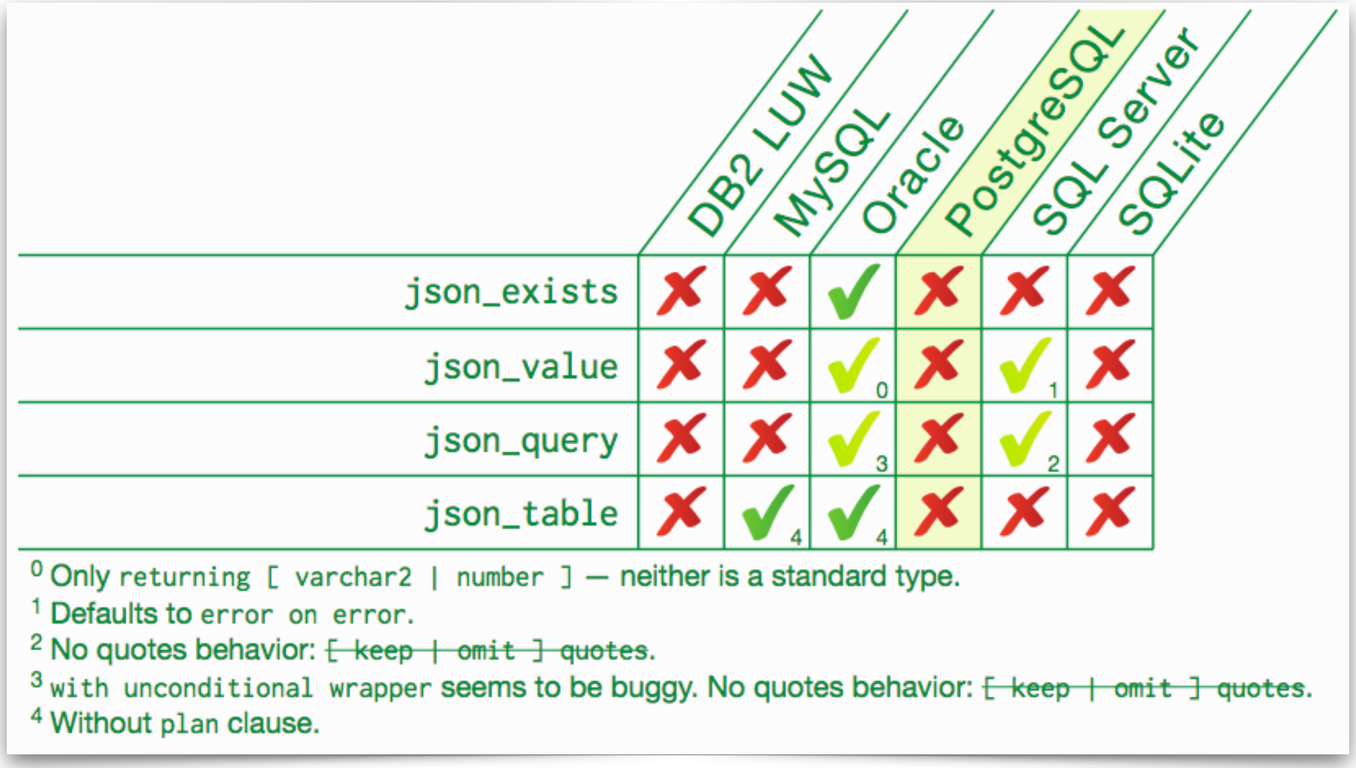
	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`is [not] json predicate`	✗	✗	✓[0]	✗	✗	✗
`on [ error | empty ] clauses`	✗	✗	✓[1]	✗	✗	✗

[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 7fe04ce92 as basis, applied those patches on top:

0001-strict-do_to_timestamp-v14.patch
0002-pass-cstring-to-do_to_timestamp-v14.patch
0003-add-to_datetime-v14.patch
0004-jsonpath-v14.patch
0005-jsonpath-gin-v14.patch
0006-jsonpath-json-v14.patch
0007-remove-PG_TRY-in-jsonpath-arithmetics-v14.patch

} SQL/JSON: jsonpath

0010-add-invisible-coercion-form-v13.patch
0011-add-function-formats-v13.patch
0012-sqljson-v13.patch
0013-sqljson-json-v13.patch

} SQL/JSON: functions

0014-json_table-v13.patch
0015-json_table-json-v13.patch

} SQL/JSON: JSON_TABLE

JSON — Preliminary testing of patches

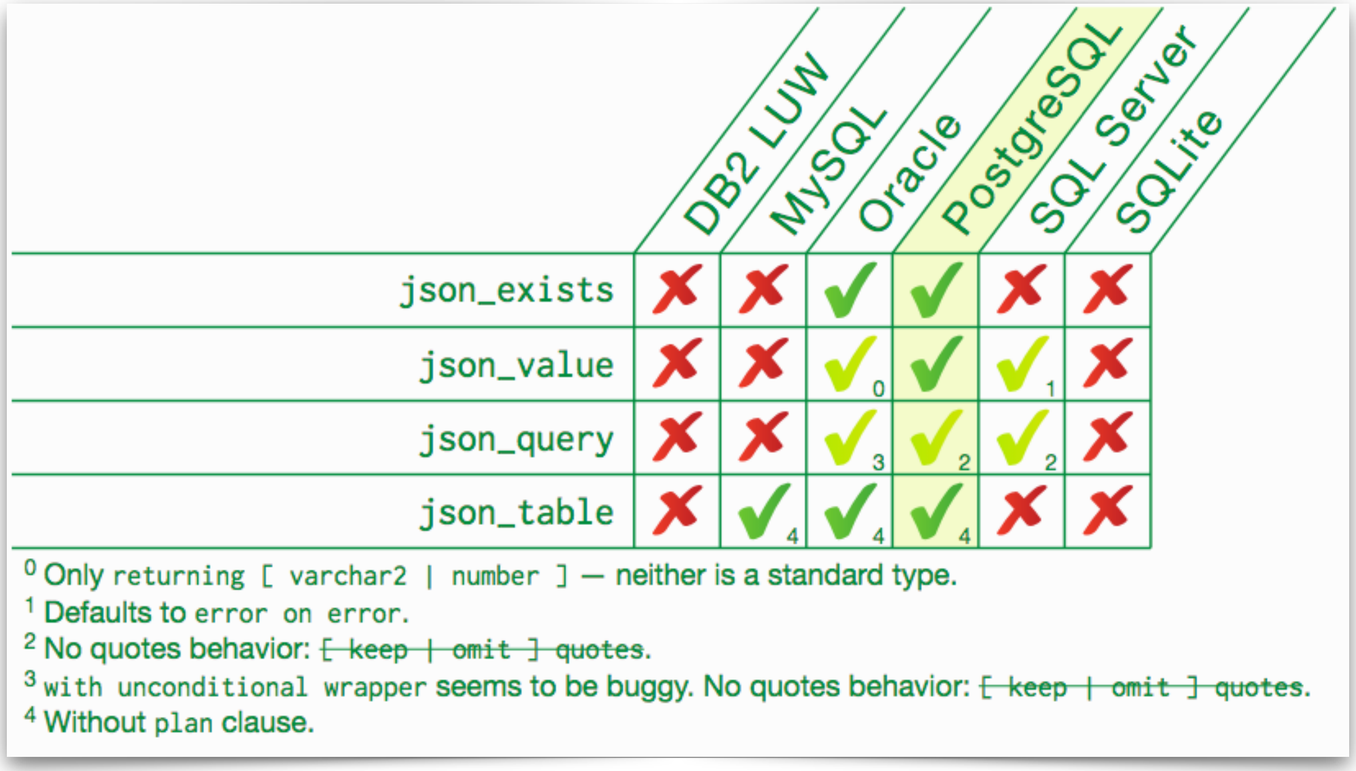
	DB2 LUW	MySQL	Oracle	PostgreSQL	SQL Server	SQLite
`is [not] json` predicate	✗	✗	✓[0]	✓[1]	✗	✗
`on [ error | empty ]` clauses	✗	✗	✓[2]	✓	✗	✗

[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 …)`	✗	✗	✓[5]	✓[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 — 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`.

Standard SQL Gap Analysis

Incomplete or “wrong”:

- `extract` (declared type)
- `ignore nulls`
- `agg(distinct) over()`
- `fetch…percent,with ties`
- Functional dependencies

Work in progress

- `merge`
- JSON

Missing

- Row pattern recognition
- Temporal tables
- Generated Columns
- Combined data change and retrieval
- Partitioned join
- `listagg`
- Distinct data types
- … (this list is not exhaustive)

How can I help?

- ➡I publish an article on each new version once it is released (pretty late for helpful feedback)
- ➡I start preparing for this article once a public beta is available (but it is often pushed by higher priority tasks -> no guarantee)
- ➡I do **not** monitor -hackers, but Depesz's “waiting for” (This is typically the first time I notice a new feature is coming up)
- ➡If you have questions on the standard or would like to get conformance test results at a earlier stage, **ping me**.

Twitter: @MarkusWinand — markus.winand@winand.at