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
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.
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**

<table>
<thead>
<tr>
<th>Function</th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract(... from &lt;datetime&gt;)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>extract(... from &lt;interval&gt;)</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>cast(&lt;timestamp&gt; as date)</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>cast(&lt;timestamp&gt; as time)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>PostgreSQL</th>
<th>MySQL</th>
<th>Oracle</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>extract(field from timestamp)</code></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>extract(field from interval)</code></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>cast(&lt;timestamp&gt; as date)</code></td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td><code>cast(&lt;timestamp&gt; as time)</code></td>
<td>✓</td>
<td>✓</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

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

Table:

<table>
<thead>
<tr>
<th>Field Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>field from timestamp</td>
<td>double precision</td>
</tr>
<tr>
<td>field from interval</td>
<td>double precision</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Window Functions</th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEAD and LAG</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>FIRST_VALUE, LAST_VALUE</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>NTH_VALUE</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Nested window functions</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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.)

<table>
<thead>
<tr>
<th>Nested window functions</th>
<th>NTH_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![Table]</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregates (count, sum, min, ...)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Distinct Aggregates</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
FETCH [FIRST|NEXT] ... 

The standard’s answer to LIMIT, but more options

(T866, T867)
**FETCH FIRST: no percent, no with ties**

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-level fetch first</td>
<td>✔️</td>
<td>❌ 0</td>
<td>✔️</td>
<td>✔️</td>
<td>❌ 1 ❌</td>
<td></td>
</tr>
<tr>
<td>Subqueries with fetch first</td>
<td>✔️</td>
<td>❌ 0</td>
<td>✔️</td>
<td>✔️</td>
<td>❌ 1 ❌</td>
<td></td>
</tr>
<tr>
<td>Top-level fetch first in views</td>
<td>❌ 2</td>
<td>❌ 0</td>
<td>✔️</td>
<td>✔️</td>
<td>❌ 1 ❌</td>
<td></td>
</tr>
<tr>
<td>Dynamic quantity</td>
<td>✔️</td>
<td>❌</td>
<td>✔️</td>
<td>✔️ 3</td>
<td>❌ ❌</td>
<td></td>
</tr>
<tr>
<td>fetch first ... percent</td>
<td>❌</td>
<td>❌</td>
<td>✔️</td>
<td>❌</td>
<td>❌</td>
<td></td>
</tr>
<tr>
<td>fetch first ... with ties</td>
<td>❌</td>
<td>❌</td>
<td>✔️</td>
<td>❌</td>
<td>❌</td>
<td></td>
</tr>
<tr>
<td>SQL State 2201W if quantity &lt; 1</td>
<td>✔️ 6</td>
<td>❌</td>
<td>❌</td>
<td>✔️ 6</td>
<td>❌</td>
<td></td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>Feature</th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch first in views</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Dynamic quantity</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Fetch first ... percent</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Fetch first ... with ties</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SQL State 2201W if quantity &lt; 1</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base table PRIMARY KEY</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Base table UNIQUE</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Joined tables</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓&lt;sub&gt;0&lt;/sub&gt;</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WHERE clause</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>GROUP BY clause</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

<sup>0</sup> Not following joins to PRIMARY KEYs or UNIQUE constraints

**Docs: unsupported features:**

<table>
<thead>
<tr>
<th>T301</th>
<th>Functional dependencies</th>
<th>partially supported</th>
</tr>
</thead>
</table>
**Functional dependencies:** only simplest cases

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base table PRIMARY KEY</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Base table UNIQUE</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Joined tables</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>WHERE clause</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>GROUP BY clause</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

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

Docs: unsupported features:

<table>
<thead>
<tr>
<th>Docs: unsupported features:</th>
<th>T301</th>
<th>Functional dependencies</th>
<th>partially supported</th>
</tr>
</thead>
</table>
### Functional dependencies: only simplest cases

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.24</td>
<td>Functional dependencies.</td>
<td>97</td>
</tr>
<tr>
<td>4.24.1</td>
<td>Overview of functional dependency rules and notations.</td>
<td>97</td>
</tr>
<tr>
<td>4.24.2</td>
<td>General rules and definitions.</td>
<td>98</td>
</tr>
<tr>
<td>4.24.3</td>
<td>Known functional dependencies in a base table.</td>
<td>99</td>
</tr>
<tr>
<td>4.24.4</td>
<td>Known functional dependencies in a viewed table.</td>
<td>99</td>
</tr>
<tr>
<td>4.24.5</td>
<td>Known functional dependencies in a transition table.</td>
<td>100</td>
</tr>
<tr>
<td>4.24.6</td>
<td>Known functional dependencies in <code>&lt;table value constructor&gt;</code></td>
<td>100</td>
</tr>
<tr>
<td>4.24.7</td>
<td>Known functional dependencies in a <code>&lt;joined table&gt;</code></td>
<td>100</td>
</tr>
<tr>
<td>4.24.8</td>
<td>Known functional dependencies in a <code>&lt;table primary&gt;</code></td>
<td>102</td>
</tr>
<tr>
<td>4.24.9</td>
<td>Known functional dependencies in a <code>&lt;table factor&gt;</code></td>
<td>103</td>
</tr>
<tr>
<td>4.24.10</td>
<td>Known functional dependencies in a <code>&lt;table reference&gt;</code></td>
<td>103</td>
</tr>
<tr>
<td>4.24.11</td>
<td>Known functional dependencies in the result of a <code>&lt;from clause&gt;</code></td>
<td>103</td>
</tr>
<tr>
<td>4.24.12</td>
<td>Known functional dependencies in the result of a <code>&lt;where clause&gt;</code></td>
<td>104</td>
</tr>
<tr>
<td>4.24.13</td>
<td>Known functional dependencies in the result of a <code>&lt;group by clause&gt;</code></td>
<td>104</td>
</tr>
<tr>
<td>4.24.14</td>
<td>Known functional dependencies in the result of a <code>&lt;having clause&gt;</code></td>
<td>105</td>
</tr>
<tr>
<td>4.24.15</td>
<td>Known functional dependencies in a <code>&lt;query specification&gt;</code></td>
<td>105</td>
</tr>
<tr>
<td>4.24.16</td>
<td>Known functional dependencies in a <code>&lt;query expression&gt;</code></td>
<td>106</td>
</tr>
</tbody>
</table>
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)
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
SELECT COUNT(*) as sessions,
    AVG(duration) as 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
)

Since SQL:2016

Row Pattern Matching

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

Since SQL:2016
Row Pattern Matching

Time

30 minutes

undefined pattern variable: matches any row

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

Row Pattern Matching
Since SQL:2016

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

Since SQL:2016
Row Pattern Matching

Very much like GROUP BY

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

Since SQL:2016

Very much like SELECT

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

Since SQL:2016

Row Pattern Matching

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

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`  

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>from clause</strong></td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>window clause</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>full aggregate support</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Free technical report by ISO:  
Row Pattern Matching

Since SQL:2016


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.

- **System Versioning**
  - Datetime columns visible (not 100% transparent)[0]
  - User cannot set them.
  - Constraints remain unchanged.

- **Application Versioning**
  - 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

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate always as row ...</td>
<td>✓0</td>
<td>✓1</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>period for system_time</td>
<td>✓1</td>
<td>✓1</td>
<td>✗</td>
<td>✗</td>
<td>✓2</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Add system versioning to table</td>
<td>✓2</td>
<td>✓2</td>
<td>✗</td>
<td>✗</td>
<td>✓2</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Drop system versioning from table</td>
<td>✓2</td>
<td>✓2</td>
<td>✗</td>
<td>✗</td>
<td>✓2</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>for system_time as of ...</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✗</td>
<td>✗</td>
<td>✓ ✓ ✓ ✓</td>
<td>✗ ✓ ✓ ✓</td>
<td>✗</td>
</tr>
<tr>
<td>for system_time between ...</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✗</td>
<td>✗ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✗ ✓ ✓ ✓</td>
<td>✗</td>
</tr>
<tr>
<td>Immutable transaction time</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✗</td>
<td>✗</td>
<td>✓ ✓ ✓ ✓</td>
<td>✗ ✓ ✓ ✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

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").
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 (136 yrs)

[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

<table>
<thead>
<tr>
<th>Action</th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>period for business_time</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>without overlaps constraint</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>update ... for portion of</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>delete ... for portion of</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>overlaps</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>equals</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>contains</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>precedes</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>succeeds</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>immediately precedes</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>immediately succeeds</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>overlaps</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>equals</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>contains</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>precedes</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>succeeds</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>immediately precedes</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>immediately succeeds</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Functionality is available, only the standard syntax is missing.

0 Doesn't recognize period names. Use (start_ts, end_ts) syntax without keyword period.

1 Use range type and respective operators.
Generated Columns

(T175)
# Generated Columns

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

<table>
<thead>
<tr>
<th>Database</th>
<th><code>generate always as (…)</code></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 LUW</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MariaDB</td>
<td>✓ 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oracle</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td></td>
<td>✓ 1</td>
<td></td>
</tr>
<tr>
<td>SQL Server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

### New/Combined Data Change and Retrieval

<table>
<thead>
<tr>
<th>Table Type</th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>[new</td>
<td>final] TABLE (INSERT ...)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[...] TABLE (UPDATE ...)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[old] TABLE (DELETE ...)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[...] TABLE (MERGE ...)</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>In DML</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ts</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>...</td>
</tr>
</tbody>
</table>
Partitioned Join — Filling gaps in time series

```sql
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

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

<table>
<thead>
<tr>
<th>grp</th>
<th>ts</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>...</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>...</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>...</td>
</tr>
</tbody>
</table>

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

```
SELECT *
FROM data
PARTITION BY (grp)
RIGHT JOIN generate_series(…)
ON ...
```
## Partitioned Join — Filling gaps in time series

<table>
<thead>
<tr>
<th>Join Type</th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFT OUTER partitioned join</td>
<td>✗ 0</td>
<td>✗ 1</td>
<td>✗ 1</td>
<td>✔ 1</td>
<td>✗ 0</td>
<td>✗ 0</td>
<td>✗ 1</td>
</tr>
<tr>
<td>RIGHT OUTER partitioned join</td>
<td>✗ 0</td>
<td>✗ 1</td>
<td>✗ 1</td>
<td>✔ 1</td>
<td>✗ 0</td>
<td>✗ 0</td>
<td>✗ 1</td>
</tr>
<tr>
<td>FULL OUTER partitioned join</td>
<td>✗ 1</td>
<td>✗</td>
<td>✗</td>
<td>✗ 0</td>
<td>✗ 0</td>
<td>✗ 0</td>
<td>✗ 1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>listagg(...) within group (...)</code></td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><code>listagg(...) on overflow ...</code></td>
<td>✗</td>
<td>✗</td>
<td>✔️́₀</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><code>listagg(distinct ...)</code></td>
<td>✔️́₁</td>
<td>✗</td>
<td>✔️́₀</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><code>SQLSTATE 22001 on truncation</code></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><code>listagg with grouping sets</code></td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><code>stagg... within group... filter...</code></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><code>listagg... within group... over...</code></td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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

CREATE TYPE...AS <pred. type>  

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TYPE...AS &lt;pred. type&gt;</td>
<td>✔️</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>
Work in progress
MERGE

(F312, F313, F314)
MERGE — conditional insert/update/delete

<table>
<thead>
<tr>
<th></th>
<th>DB2</th>
<th>LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>[NOT] MATCHED AND &lt;condition&gt;</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Multiple update/insert branches</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WHEN MATCHED DELETE</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>DB2</th>
<th>LUW</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>[NOT] MATCHED AND &lt;condition&gt;</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Multiple update/insert branches</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>WHEN MATCHED DELETE</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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 ...]]

As of c9c875a (just before revert)
JSON

(T811–T838)
### JSON

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>is [not] json predicate</strong></td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>**on [ error</td>
<td>empty ] clauses**</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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 ]

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>json_object</strong></td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>json_array</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

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].`

5 Absent on null is buggy.
<table>
<thead>
<tr>
<th></th>
<th>DB2</th>
<th>LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>json_exists</code></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><code>json_value</code></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><code>json_query</code></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><code>json_table</code></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>JSON path</th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSON path: lax mode (default)</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JSON path: strict mode</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JSON path: item method</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JSON path: multiple subscripts</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JSON path: .* member accessor</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JSON path: filter expressions</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JSON path: starts with</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>JSON path: like_regex</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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.
5. Not in json_query. Only as last step of expression.
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-jspath-v14.patch
- 0005-jspath-gin-v14.patch
- 0006-jspath-json-v14.patch
- 0007-remove-PG_TRY-in-jspath-arithmetic-v14.patch
- 0010-add-invisible-coercion-form-v13.patch
- 0011-add-function-formats-v13.patch
- 0012-sqljson-v13.patch
- 0013-sqljson-json-v13.patch
- 0014-json_table-v13.patch
- 0015-json_table-json-v13.patch

---

**SQL/JSON: jsonpath**

- 0001-strict-do_to_timestamp-v14
- 0002-pass-cstring-to-do_to_timestamp-v14
- 0003-add-to_datetime-v14
- 0004-jspath-v14
- 0005-jspath-gin-v14
- 0006-jspath-json-v14
- 0007-remove-PG_TRY-in-jspath-arithmetic-v14

**SQL/JSON: functions**

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

**SQL/JSON: JSON_TABLE**

- 0014-json_table-v13
- 0015-json_table-json-v13
**JSON — Preliminary testing of patches**

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is [not] json predicate</code></td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>`on [ error</td>
<td>empty ] clauses`</td>
<td>×</td>
<td>×</td>
<td>✓&lt;sup&gt;1&lt;/sup&gt;</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

---

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 ]

---

<table>
<thead>
<tr>
<th></th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>json_object</code></td>
<td>×</td>
<td>×</td>
<td>✓&lt;sup&gt;0&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;0&lt;/sup&gt;</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><code>json_array</code></td>
<td>×</td>
<td>✓&lt;sup&gt;1&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;2&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;2&lt;/sup&gt;</td>
<td>×&lt;sup&gt;1&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td><code>json_objectagg</code></td>
<td>×&lt;sup&gt;3&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;4&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;0&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;0&lt;/sup&gt;</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><code>json_arrayagg(… order by …)</code></td>
<td>×&lt;sup&gt;5&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;5&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;0&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;0&lt;/sup&gt;</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>Function</th>
<th>DB2 LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>json_exists</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>json_value</td>
<td>✗</td>
<td>✗</td>
<td>✓&lt;sub&gt;0&lt;/sub&gt;</td>
<td>✓</td>
<td>✓&lt;sub&gt;1&lt;/sub&gt;</td>
<td>✗</td>
</tr>
<tr>
<td>json_query</td>
<td>✗</td>
<td>✗</td>
<td>✓&lt;sub&gt;3&lt;/sub&gt;</td>
<td>✓&lt;sub&gt;2&lt;/sub&gt;</td>
<td>✓&lt;sub&gt;2&lt;/sub&gt;</td>
<td>✗</td>
</tr>
<tr>
<td>json_table</td>
<td>✗</td>
<td>✓&lt;sub&gt;4&lt;/sub&gt;</td>
<td>✓&lt;sub&gt;4&lt;/sub&gt;</td>
<td>✓&lt;sub&gt;4&lt;/sub&gt;</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

1. Only returning [ varchar2 | number ] — neither is a standard type.
2. Defaults to error on error.
3. No quotes behavior: ['keep' | 'omit'] quotes.
4. With unconditional wrapper seems to be buggy. No quotes behavior: ['keep' | 'omit'] quotes.
4. Without plan clause.
### JSON — Preliminary testing of patches

<table>
<thead>
<tr>
<th>JSON path: lax mode (default)</th>
<th>DB2</th>
<th>LUW</th>
<th>MySQL</th>
<th>Oracle</th>
<th>PostgreSQL</th>
<th>SQL Server</th>
<th>SQLite</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>JSON path: strict mode</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>JSON path: item method</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>JSON path: multiple subscripts</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>JSON path: .* member accessor</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>JSON path: filter expressions</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>JSON path: starts with</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>JSON path: like_regex</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
</tbody>
</table>

---

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.
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