Still using Windows 3.1?

So why stick with SQL-92?

@ModernSQL - https://modern-sql.com/
@MarkusWinand
SQL: 1999
WITH

(Common Table Expressions)
The Problem

Nested queries are hard to read:

```
SELECT ...
FROM (SELECT ...
       FROM t1
       JOIN (SELECT ... FROM ...
              ) a ON (...
       ) b
JOIN (SELECT ... FROM ...
      ) c ON (...
```
**WITH** (non-recursive)

The Problem

Nested queries are hard to read:

```
SELECT ... 
FROM (SELECT ...
    FROM t1
    JOIN (SELECT ... FROM ...
        ) a ON (...
    ) b
JOIN (SELECT ... FROM ...
    ) c ON (...
) 
```
WITH (non-recursive)

The Problem

Nested queries are hard to read:

```
SELECT ...
FROM (SELECT ...
    FROM t1
    JOIN (SELECT ... FROM ...)
    a ON (....)
) b
JOIN (SELECT ... FROM ...)
    c ON (....)
```
WITH (non-recursive)

The Problem

Nested queries are hard to read:

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

Then this...
Finally the first line makes sense

SELECT ...
FROM (SELECT ...
    FROM t1
    JOIN (SELECT ... FROM ...
    ) a ON (...
) b
JOIN (SELECT ... FROM ...
    ) c ON (...
)
CTEs are statement-scoped views:

\[
\begin{align*}
\text{WITH} & \quad \text{Keyword} \\
\text{WITH} & \quad \text{WITH} \\
\text{a (c1, c2, c3)} & \quad \text{a (c1, c2, c3)} \\
\text{AS (SELECT c1, c2, c3 FROM \ldots)} & \quad \text{AS (SELECT c1, c2, c3 FROM \ldots)},
\end{align*}
\]
CTEs are statement-scoped views:

**WITH**

(*Name of CTE and (here optional) column names*)

WITH

```sql
a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM ...),
```

Since SQL:1999
WITH (non-recursive) Since SQL:1999

CTEs are statement-scoped views:

WITH
  a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM ...)

Definition
**WITH** (non-recursive) **Since SQL:1999**

CTEs are statement-scoped views:

WITH
a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM ...),

Introduces another CTE
Don't repeat WITH
CTEs are statement-scoped views:

WITH (non-recursive) Since SQL:1999

WITH
  a (c1, c2, c3)
  AS (SELECT c1, c2, c3 FROM ...),

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

  c (...)
WITH (non-recursive) AS (SELECT c4, ... FROM t1 JOIN a ON (...),

Third CTE

SELECT ...
FROM b JOIN c ON (...)

Since SQL:1999
WITH (non-recursive) AS (SELECT c4, ...
    FROM t1
    JOIN a
    ON (...)
),

    AS (SELECT ... FROM ...
    SELECT ...
    FROM b JOIN c ON (...)

No comma!
WITH (non-recursive) AS (SELECT c4, ... FROM t1 JOIN a ON (...), c (...) AS (SELECT ... FROM ...)

SELECT ...
FROM b JOIN c ON (...)

Main query

Since SQL:1999
WITH (non-recursive) Since SQL:1999

WITH
  a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM ...),
  b (c4, ...)
AS (SELECT c4, ...
    FROM t1
    JOIN a
    ON (...)
  ),
  c (...)
AS (SELECT ... FROM ...)
SELECT ...
  FROM b JOIN c ON (...)

Read top down
**WITH** (non-recursive)

- **Literate SQL**
  Organize SQL code to improve maintainability

- **Assign column names**
  to tables produced by `values` or `unnest`.

- **Overload tables (for testing)**
  with queries hide tables of the same name.
  [https://modern-sql.com/use-case/unit-tests-on-transient-data](https://modern-sql.com/use-case/unit-tests-on-transient-data)

**Use-Cases**
**WITH** (non-recursive)  

**WITH** are the "private methods" of SQL

**WITH** is a prefix to **SELECT**

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

**WITH** in detail:

https://modern-sql.com/feature/with
WITH (non-recursive)

Availability

- MariaDB 5.1, 10.2
- MySQL 8.0
- PostgreSQL 8.4
- SQLite 3.8.3
- DB2 LUW 7.0
- Oracle 9iR2
- SQL Server 2005

[0] Only for top-level SELECT statements
[1] Only allowed at the very begin of a statement. E.g. WITH...INSERT...SELECT.
WITH RECURSIVE

(Common Table Expressions)
WITH RECURSIVE

Coping with hierarchies in the Adjacency List Model

CREATE TABLE t (  
id NUMERIC NOT NULL,  
parent_id NUMERIC,  
...
PRIMARKEY (id)
)
WITH RECURSIVE

Coping with hierarchies in the Adjacency List Model[0]

CREATE TABLE t (  
id NUMERIC NOT NULL,
parent_id NUMERIC,
...  
PRIMARY KEY (id)  
)
WITH RECURSIVE

Coping with hierarchies in the Adjacency List Model

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

RECURSIVE

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

Coping with hierarchies in the Adjacency List Model

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

[0] Hierarchies implemented using a “parent id” — see “Joe Celko’s Trees and Hierarchies in SQL for Smarties”
SELECT *
FROM t AS d0
LEFT JOIN t AS d1
  ON (d1.parent_id=d0.id)
LEFT JOIN t AS d2
  ON (d2.parent_id=d1.id)
WHERE d0.id = ?

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

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

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

SELECT *
FROM subtree
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

```
WITH RECURSIVE cte (n)
AS (SELECT 1
    UNION ALL
    SELECT n+1
    FROM cte
    WHERE n < 3)
SELECT * FROM cte
```
Recursive common table expressions may refer to themselves in a leg of a \texttt{UNION \ [ALL]}:

\begin{verbatim}
WITH RECURSIVE cte (n) AS (SELECT 1
UNION ALL
SELECT n+1
FROM cte
WHERE n < 3)
SELECT * FROM cte
\end{verbatim}
Recursive common table expressions may refer to themselves in a leg of a **UNION [ALL]**:

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

Since SQL:1999
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

```sql
WITH RECURSIVE cte (n)
AS (SELECT 1
    UNION ALL
    SELECT n+1
    FROM cte
    WHERE n < 3)
SELECT * FROM cte
```
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

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

Since SQL:1999
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

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

Once it becomes part of the final result, it is included in the final output.
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

```
WITH RECURSIVE cte (n) AS
    (SELECT 1
     UNION ALL
     SELECT n+1
     FROM cte
     WHERE n < 3)
SELECT * FROM cte
```
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

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

Second leg of `UNION` is executed

Second leg of `UNION [ALL]` is executed
WITH RECURSIVE

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

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

Result sent there again

Since SQL:1999
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

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

Since SQL:1999
Recursive common table expressions may refer to themselves in a leg of a **UNION [ALL]**:

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

It's a loop!

Since SQL:1999
Recursive common table expressions may refer to themselves in a leg of a UNION [ALL]:

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

It's a loop!
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

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

It's a loop!
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

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

Since SQL:1999
Recursive common table expressions may refer to themselves in a leg of a `UNION [ALL]`:

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

Since SQL:1999
Use Cases

- **Row generators**
  To fill gaps (e.g., in time series), generate test data.

- **Processing graphs**
  Shortest route from person A to B in LinkedIn/Facebook/Twitter/…

- **Finding distinct values**
  with $n \log(N)$ time complexity.

  [...many more…]

---

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

WITH RECURSIVE is the “while” of SQL

WITH RECURSIVE "supports" infinite loops

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

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

Availability

- MariaDB 5.1, 10.2
- MySQL 8.0
- PostgreSQL 8.4
- SQLite 3.8.3[0]
- DB2 LUW 7.0
- Oracle 11gR2
- SQL Server 2005

[0] Only for top-level SELECT statements
GROUPING SETS
GROUPING SETS

Only one \texttt{GROUP BY} operation at a time:

Monthly revenue

\begin{verbatim}
SELECT year, month, sum(revenue)
FROM tbl
GROUP BY year, month
\end{verbatim}

Yearly revenue

\begin{verbatim}
SELECT year, sum(revenue)
FROM tbl
GROUP BY year
\end{verbatim}

Before SQL:1999
GROUPING SETS

Before SQL: 1999

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

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

Before SQL:1999

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

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

SELECT year
  , month
  , sum(revenue)
FROM tbl
GROUP BY
  GROUPING SETS (  
    (year, month)  
    , (year)  
  )
GROUPING SETS are multiple GROUP BYs in one go

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

GROUPING (function) disambiguates the meaning of NULL
(was the grouped data NULL or is this column not currently grouped?)

Permutations can be created using ROLLUP and CUBE
(ROLLUP(a,b,c) = GROUPING SETS ((a,b,c), (a,b),(a),()))
GROUPING SETS

Availability

MariaDB
MySQL
PostgreSQL
SQLite
DB2 LUW
Oracle
SQL Server

5.1

5.0

9.5

2008

[0] Only ROLLUP (proprietary syntax).
SQL:2003
OVER

and

PARTITION BY
Two distinct concepts could not be used independently:

- **Merge rows with the same key properties**
  - `GROUP BY` to specify key properties
  - `DISTINCT` to use full row as key

- **Aggregate data from related rows**
  - Requires `GROUP BY` to segregate the rows
  - `COUNT, SUM, AVG, MIN, MAX` to aggregate grouped rows
The Problem

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

SELECT DISTINCT c1,
c2
FROM t

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

The Problem

Yes ⇠ Merge rows ⇢ No

No ← Aggregate → Yes

SELECT c1,
c2
FROM t

SELECT c1
JOIN ( ) ta
ON (t.c1=ta.c1)
OVER (PARTITION BY)

The Problem

<table>
<thead>
<tr>
<th>Merge rows</th>
<th>Aggregate</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

SELECT c1, c2 FROM t

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

SELECT c1, c2 FROM t

SELECT c1, c2 FROM t

SELECT c1, SUM(c2) tot FROM t GROUP BY c1
### OVER (PARTITION BY)

<table>
<thead>
<tr>
<th>Merge rows</th>
<th>Aggregate</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

#### SQL Queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT c1, c2 FROM t</td>
<td>No Merge rows, No Aggregate</td>
</tr>
<tr>
<td>SELECT DISTINCT c1, c2 FROM t</td>
<td>Yes Merge rows, No Aggregate</td>
</tr>
<tr>
<td>SELECT c1, SUM(c2) tot FROM t GROUP BY c1</td>
<td>Yes Aggregate, Yes</td>
</tr>
<tr>
<td>SELECT c1 FROM t JOIN ta ON (t.c1 = ta.c1)</td>
<td>No Aggregate, No</td>
</tr>
</tbody>
</table>

#### The Problem

- Merge rows
- Aggregate
### OVER (PARTITION BY)

<table>
<thead>
<tr>
<th>No</th>
<th>Aggregate</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT <code>c1</code> , <code>c2</code> FROM <code>t</code></td>
<td>Yes</td>
<td>SELECT <code>c1</code> , <code>c2</code>, <code>SUM(c2)</code> OVER (PARTITION BY <code>c1</code>) FROM <code>t</code></td>
</tr>
<tr>
<td>SELECT DISTINCT <code>c1</code> , <code>c2</code> FROM <code>t</code></td>
<td>← Merge rows</td>
<td>SELECT <code>c1</code> , <code>SUM(c2)</code> <code>tot</code> FROM <code>t</code> GROUP BY <code>c1</code></td>
</tr>
</tbody>
</table>
SELECT dep, salary
FROM emp

<table>
<thead>
<tr>
<th>dep</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
</tr>
</tbody>
</table>

**OVER (PARTITION BY)**

How it works
SELECT dep, salary, 
FROM emp

<table>
<thead>
<tr>
<th>dep</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
</tr>
</tbody>
</table>
### OVER (PARTITION BY)

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

<table>
<thead>
<tr>
<th>dep</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
</tr>
</tbody>
</table>

How it works
SELECT dep, salary, SUM(salary) OVER() FROM emp

### How it works

<table>
<thead>
<tr>
<th>dep</th>
<th>salary</th>
<th>OVER()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>6000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
<td>6000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
<td>6000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
<td>6000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
<td>6000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
<td>6000</td>
</tr>
</tbody>
</table>
SELECT dep, salary, SUM(salary) OVER(PARTITION BY dep) FROM emp

<table>
<thead>
<tr>
<th>dep</th>
<th>salary</th>
<th>SUM(salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>22</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
<td>3000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
<td>3000</td>
</tr>
<tr>
<td>333</td>
<td>1000</td>
<td>3000</td>
</tr>
</tbody>
</table>
OVER
and
ORDER BY
(Framing & Ranking)
### The Problem

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

<table>
<thead>
<tr>
<th>id</th>
<th>value</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+10</td>
<td>+10</td>
</tr>
<tr>
<td>2</td>
<td>+20</td>
<td>+30</td>
</tr>
<tr>
<td>3</td>
<td>-10</td>
<td>+20</td>
</tr>
<tr>
<td>4</td>
<td>+50</td>
<td>+70</td>
</tr>
<tr>
<td>5</td>
<td>-30</td>
<td>+40</td>
</tr>
<tr>
<td>6</td>
<td>-20</td>
<td>+20</td>
</tr>
</tbody>
</table>
OVER (ORDER BY)

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

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

<table>
<thead>
<tr>
<th>id</th>
<th>value</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+10</td>
<td>+10</td>
</tr>
<tr>
<td>2</td>
<td>+20</td>
<td>+30</td>
</tr>
<tr>
<td>3</td>
<td>-10</td>
<td>+20</td>
</tr>
<tr>
<td>4</td>
<td>+50</td>
<td>+70</td>
</tr>
<tr>
<td>5</td>
<td>-30</td>
<td>+40</td>
</tr>
<tr>
<td>6</td>
<td>-20</td>
<td>+20</td>
</tr>
</tbody>
</table>
OVER (ORDER BY)

Since SQL:2003

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

**Since SQL:2003**

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

<table>
<thead>
<tr>
<th></th>
<th>value</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+10</td>
<td>+10</td>
</tr>
<tr>
<td>2</td>
<td>+20</td>
<td>+30</td>
</tr>
<tr>
<td>3</td>
<td>-10</td>
<td>+20</td>
</tr>
<tr>
<td>4</td>
<td>+50</td>
<td>+70</td>
</tr>
<tr>
<td>5</td>
<td>-30</td>
<td>+40</td>
</tr>
<tr>
<td>6</td>
<td>-20</td>
<td>+20</td>
</tr>
</tbody>
</table>
SELECT id,
    value,
    SUM(value) OVER (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM transactions t
SELECT id, value, SUM(value) OVER (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM transactions t
SELECT id, value, SUM(value) OVER ( ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ) FROM transactions t
SELECT id, value, SUM(value) OVER (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM transactions t
SELECT id, value, SUM(value) OVER (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM transactions t
SELECT id,
    value,
    SUM(value)
OVER (  
    ORDER BY id 
    ROWS BETWEEN 
    UNBOUNDED PRECEDING 
    AND CURRENT ROW 
) 
FROM transactions t

<table>
<thead>
<tr>
<th>acnt</th>
<th>id</th>
<th>value</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>+10</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>+20</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>-10</td>
<td></td>
</tr>
<tr>
<td>333</td>
<td>4</td>
<td>+50</td>
<td></td>
</tr>
<tr>
<td>333</td>
<td>5</td>
<td>-30</td>
<td></td>
</tr>
<tr>
<td>333</td>
<td>6</td>
<td>-20</td>
<td></td>
</tr>
</tbody>
</table>
**OVER** (ORDER BY)

Since SQL:2003

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

<table>
<thead>
<tr>
<th>acnt</th>
<th>id</th>
<th>value</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>+10</td>
<td>+10</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>+20</td>
<td>+20</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>-10</td>
<td>+10</td>
</tr>
<tr>
<td>333</td>
<td>4</td>
<td>+50</td>
<td>+50</td>
</tr>
<tr>
<td>333</td>
<td>5</td>
<td>-30</td>
<td>+20</td>
</tr>
<tr>
<td>333</td>
<td>6</td>
<td>-20</td>
<td>0</td>
</tr>
</tbody>
</table>
Since SQL:2003

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

<table>
<thead>
<tr>
<th>n</th>
<th>ROW_NUMBER</th>
<th>RANK</th>
<th>DENSE_RANK</th>
<th>PERCENT_RANK</th>
<th>CUME_DIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0.33...</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>0.33...</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
OVER (SQL:2003)

Use Cases

- Aggregates without `GROUP BY`
- Running totals, moving averages
- Ranking
  - Top-N per Group
- Avoiding self-joins

[... many more ...]

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

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

OVER may follow any aggregate function

OVER defines which rows are visible at each row

OVER() makes all rows visible at every row

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

OVER(ORDER BY ... BETWEEN) segregates using <, >
OVER (SQL:2003)

Availability

MariaDB
MySQL
PostgreSQL
SQLite
DB2 LUW
Oracle
SQL Server

Hive
Impala
Spark
NuoDB

[0] Expected for release 3.25.0 (available in snapshot release).
NULLS FIRST/LAST
The sorting of NULL is implementation defined (some DBs sort NULL as great, others as very small value).

NULLS FIRST/LAST

Before SQL:2003

If you know a value larger/smaller than any actual value...

SELECT ...
FROM ...
ORDER BY COALESCE(nullable, ?);
The sorting of **NULL** is implementation defined
(some DBs sort **NULL** as great, others as very small value)

```sql
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

- MariaDB[0]
- MySQL[0]
- PostgreSQL
- SQLite[0]
- DB2 LUW
- Oracle
- SQL Server[0]

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

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

Optional:

```sql
ELSE NULL
```

Aggregates ignore NULL*

*Exceptions: array_agg, json_objectagg, xmlagg

See: https://modern-sql.com/concept/null#aggregates
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

Pivot table: Years on the Y axis, month on X:
SELECT YEAR,
    SUM(revenue) FILTER (WHERE MONTH = 1) AS JAN,
    SUM(revenue) FILTER (WHERE MONTH = 2) AS FEB,
    ...
FROM sales
GROUP BY YEAR;
Pivot in SQL

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

See: https://modern-sql.com/use-case/pivot
FILTER

Availability

1999  2001  2003  2005  2007  2009  2011  2013  2015  2017

MariaDB
MySQL
PostgreSQL
SQLite[^0]
DB2 LUW
Oracle
SQL Server

[^0]Only with OVER clause
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)
```
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)

SELECT d1.val
FROM data
gROUP BY d1.val
HAVING count(*) = (SELECT FLOOR(COUNT(*)/2) FROM data)

Number rows

Pick middle one

(d1.val < d2.val OR (d1.val = d2.val AND d1.id < d2.id))

2)
Invers Distribution Functions \textbf{Since SQL:2003}

\begin{verbatim}
SELECT \textbf{PERCENTILE\_DISC(0.5 WITHIN GROUP (ORDER BY \textbf{val})}
FROM data
\end{verbatim}

Which value?
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

- MariaDB
- MySQL
- PostgreSQL
- SQLite
- DB2 LUN
- Oracle
- SQL Server

- Only as window function (requires OVER clause)
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

*Standard SQL allows XQuery
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

Stored in tbl.x:
<d>
  <e id="42">
    ...</c1>
  </e>
</d>

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

*Standard SQL allows XQuery XPath expressions to extract data*

Stored in tbl.x:

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

Since SQL:2006

Stored in tbl.x:
<d>
  <e id="42">...
  </e>
</d>

*Standard SQL allows XQuery

Row number (like for unnest)
XMLTABLE

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

<table>
<thead>
<tr>
<th>id</th>
<th>c1</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>...</td>
<td>1</td>
</tr>
</tbody>
</table>

Stored in tbl.x:

```xml
<d>
  <e id="42">
  <c1>...</c1>
  </e>
</d>
```

*Standard SQL allows XQuery
<table>
<thead>
<tr>
<th>Year</th>
<th>MariaDB</th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>SQLite</th>
<th>DB2 LUW</th>
<th>Oracle</th>
<th>SQL Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2001</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2003</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2005</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2007</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2009</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2011</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2013</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2015</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2017</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

[0] No XQuery (only XPath). No default namespace declaration.
SQL: 2008
FETCH FIRST
FETCH FIRST

The Problem

Limit the result to a number of rows.  
(LIMIT, TOP and ROWNUM are all proprietary)

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

SQL:2003 introduced ROW_NUMBER() to number rows.  
But this still requires wrapping to limit the result.  
And how about databases not supporting ROW_NUMBER()?
SELECT *
FROM (SELECT *
      , ROW_NUMBER() OVER (ORDER BY x) as rn
      FROM data)
      numbered_data
WHERE rn <= 10

The Problem

Limit the result to a number of rows.

SQL:2003 introduced ROW_NUMBER().

But this still requires wrapping to LIMIT.

And how about databases not supporting ROW_NUMBER?

Dammit! Let's take LIMIT.
SELECT *
FROM data
ORDER BY x
FETCH FIRST 10 ROWS ONLY
Earliest mention of LIMIT. Probably inherited from mSQL
Functionality available using LIMIT
SELECT TOP n ... SQL Server 2000 also supports expressions and bind parameters
SQL:2011
OFFSET
How to fetch the rows after a limit? (pagination anybody?)

SELECT *
FROM (SELECT *
      , ROW_NUMBER() OVER(ORDER BY x) rn
      FROM data) numbered_data
WHERE rn > 10 and rn <= 20
SELECT *
FROM data
ORDER BY x
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY

Since SQL:2011

SQL:2011 introduced OFFSET, unfortunately!
SELECT *
FROM data
ORDER BY x
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY

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

OFFSET

Grab coasters & stickers!
https://use-the-index-luke.com/no-offset
### OFFSET

<table>
<thead>
<tr>
<th>Year</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>MariaDB</td>
</tr>
<tr>
<td>2001</td>
<td>MySQL</td>
</tr>
<tr>
<td>2003</td>
<td>PostgreSQL</td>
</tr>
<tr>
<td>2005</td>
<td>SQLite</td>
</tr>
<tr>
<td>2007</td>
<td>DB2 LUW</td>
</tr>
<tr>
<td>2009</td>
<td>Oracle</td>
</tr>
<tr>
<td>2011</td>
<td>SQL Server</td>
</tr>
</tbody>
</table>

**Since SQL:2011**

- **MariaDB**
- **MySQL**
- **PostgreSQL**
- **SQLite**
- **DB2 LUW**
- **Oracle**
- **SQL Server**

---

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

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

**[2]** Requires enabling the MySQL compatibility vector: `db2set DB2_COMPATABILITY_VECTOR=MYS`
OVER
Direct access of other rows of the same window is not possible.
(E.g., calculate the difference to the previous rows)

```sql
SELECT * 
FROM t
```

<table>
<thead>
<tr>
<th>balance</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>...</td>
</tr>
<tr>
<td>90</td>
<td>...</td>
</tr>
<tr>
<td>70</td>
<td>...</td>
</tr>
<tr>
<td>30</td>
<td>...</td>
</tr>
</tbody>
</table>
The Problem

Direct access of other rows of the same window is not possible.
(E.g., calculate the difference to the previous rows)

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

<table>
<thead>
<tr>
<th>balance</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>90</td>
<td>2</td>
</tr>
<tr>
<td>70</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
</tr>
</tbody>
</table>
WITH numbered_t AS (SELECT *
, ROW_NUMBER() OVER(ORDER BY x) rn
FROM t)
SELECT curr.*
FROM numbered_t curr

The Problem

Direct access of other rows of the same window is not possible.
(E.g., calculate the difference to the previous rows)
### OVER (SQL:2011)

**The Problem**

Direct access of other rows of the same window is not possible. (E.g., calculate the difference to the previous rows)

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

<table>
<thead>
<tr>
<th>curr balance</th>
<th>prev balance</th>
<th>curr rn</th>
<th>prev rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>50</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>90</td>
<td>90</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>70</td>
<td>70</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
```
WITH numbered_t AS (SELECT *
    , ROW_NUMBER() OVER(ORDER BY x) rn
FROM t)
SELECT curr.*
FROM numbered_t curr
LEFT JOIN numbered_t prev
ON (curr.rn = prev.rn+1)

The Problem

Direct access of other rows of the same window is not possible.
(E.g., calculate the difference to the previous rows)

<table>
<thead>
<tr>
<th></th>
<th>curr</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>rn</td>
<td>balance</td>
<td>m</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>90</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>70</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>4</td>
</tr>
</tbody>
</table>
OVER (SQL:2011)

The Problem

Direct access of other rows of the same window is not possible. (E.g., calculate the difference to the previous rows)

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

SELECT curr.*
       , curr.balance
       - COALESCE(prev.balance,0)
FROM numbered_t curr
LEFT JOIN numbered_t prev
ON (curr.rn = prev.rn+1)

<table>
<thead>
<tr>
<th>curr</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>balance</td>
<td>m</td>
</tr>
<tr>
<td>-------</td>
<td>---</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>90</td>
<td>2</td>
</tr>
<tr>
<td>70</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
</tr>
</tbody>
</table>
Since SQL:2011 introduced LEAD, LAG, NTH_VALUE, ... for that:

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

Available functions:
- **LEAD** / **LAG**
- **FIRST_VALUE** / **LAST_VALUE**
- **NTH_VALUE**(col, n) FROM **FIRST/LAST** RESPECT/IGNORE NULLS
OVER (LEAD, LAG, ...) Since SQL:2011

- MariaDB: 5.1 → 10.2[0]
- MySQL: 8.0[0]
- PostgreSQL: 8.4[0]
- SQLite[0]
- DB2 LUW: 9.5[1] → 11.1
- Oracle: 11gR2 → 11.1
- SQL Server: 2012[1]

[0] No IGNORE NULLS and FROM LAST
[1] No NTH_VALUE
System Versioning
(Time Traveling)
System Versioning

The Problem

INSERT
UPDATE
DELETE

are

DESTRUCTIVE
CREATE TABLE t (...,
    start_ts TIMESTAMP(9) GENERATED ALWAYS AS ROW START,
    end_ts  TIMESTAMP(9) GENERATED ALWAYS AS ROW END,
    PERIOD FOR SYSTEM_TIME (start_ts, end_ts)
) WITH SYSTEM VERSIONING
### System Versioning

Since SQL:2011

**INSERT ... (ID, DATA) VALUES (1, 'X')**

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>start_ts</th>
<th>end_ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>10:00:00</td>
<td></td>
</tr>
</tbody>
</table>
System Versioning

**INSERT** ... (ID, DATA) VALUES (1, 'X')

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>start_ts</th>
<th>end_ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>10:00:00</td>
<td></td>
</tr>
</tbody>
</table>

**UPDATE** ... SET DATA = 'Y' ...

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>start_ts</th>
<th>end_ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>10:00:00</td>
<td>11:00:00</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>11:00:00</td>
<td></td>
</tr>
</tbody>
</table>
System Versioning

Since SQL:2011

UPDATE ... SET DATA = 'Y' ...

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>start_ts</th>
<th>end_ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>10:00:00</td>
<td>11:00:00</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>11:00:00</td>
<td></td>
</tr>
</tbody>
</table>

DELETE ... WHERE ID = 1

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>start_ts</th>
<th>end_ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>10:00:00</td>
<td>11:00:00</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>11:00:00</td>
<td>12:00:00</td>
</tr>
</tbody>
</table>
System Versioning

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>start_ts</th>
<th>end_ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>10:00:00</td>
<td>11:00:00</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>11:00:00</td>
<td>12:00:00</td>
</tr>
</tbody>
</table>

Although multiple versions exist, only the “current” one is visible per default.

After 12:00:00, `SELECT * FROM t` doesn’t return anything anymore.
System Versioning

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>start_ts</th>
<th>end_ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>10:00:00</td>
<td>11:00:00</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>11:00:00</td>
<td>12:00:00</td>
</tr>
</tbody>
</table>

With `FOR ... AS OF` you can query anything you like:

```
SELECT *
FROM t FOR SYSTEM_TIME AS OF
  TIMESTAMP '2018-09-12 10:30:00'
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>start_ts</th>
<th>end_ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>10:00:00</td>
<td>11:00:00</td>
</tr>
</tbody>
</table>
System Versioning

Since SQL:2011


MariaDB
MySQL
PostgreSQL
SQLite
DB2 LUW
Oracle
SQL Server

[0] Transaction time not immutable. See MDEV-16236.
[1] Third column required (tx id), history table required.
SQL: 2016

(released: 2016-12-15)
MATCH_RECOGNIZE

(Row Pattern Matching)
Row Pattern Matching

Example: Logfile

30 minutes

Time
Row Pattern Matching

Example: Logfile

Session 1  Session 2  Session 3  Session 4

30 minutes

Time
Row Pattern Matching

Example: Logfile

Example problem:
- Average session duration

Two approaches:
- Row pattern matching
- Start-of-group tagging
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
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

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

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
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

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

30 minutes

Time

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

Oracle doesn’t support avg on intervals — query doesn’t work as shown
Now, let’s try using window functions
SELECT count(*) sessions, avg(duration) avg_duration
FROM (SELECT MAX(ts) - MIN(ts) duration
FROM (SELECT ts, COUNT(grp_start) OVER(ORDER BY ts) session_no
FROM (SELECT ts, CASE WHEN ts >= LAG(ts, 1, DATE'1900-01-1')
OVER( ORDER BY ts )
+ INTERVAL '30' minute THEN 1
END grp_start
FROM log)
tagged
) numbered
GROUP BY session_no
) grouped

Row Pattern Matching

Before SQL:2016
Row Pattern Matching

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

Before SQL:2016
SELECT count(*) sessions, avg(duration) avg_duration
FROM (SELECT MAX(ts) - MIN(ts) duration
      FROM (SELECT ts, COUNT(grp_start) OVER(ORDER BY ts) session_no
            FROM (SELECT ts, CASE WHEN ts >= LAG(ts, 1, DATE'1900-01-1') OVER(ORDER BY ts)
                  + INTERVAL '30' minute THEN 1
                  ELSE 0 END grp_start
            FROM log)
            ) tagged
      ) numbered
GROUP BY session_no
) grouped
Row Pattern Matching

Row Pattern Matching

<table>
<thead>
<tr>
<th>1999</th>
<th>2001</th>
<th>2003</th>
<th>2005</th>
<th>2007</th>
<th>2009</th>
<th>2011</th>
<th>2013</th>
<th>2015</th>
<th>2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>MariaDB</td>
<td>MySQL</td>
<td>PostgreSQL</td>
<td>SQLite</td>
<td>DB2 LUW</td>
<td>Oracle</td>
<td>SQL Server</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Availability

12cR1
Modern SQL?

SQL has evolved *beyond the relational idea*. 
Modern SQL?

SQL has evolved *beyond the relational idea*.

If you are using SQL like 25 years ago, *you are doing it wrong*!
Modern SQL?

SQL has evolved beyond the relational idea.

If you are using SQL like 25 years ago, you are doing it wrong!

A lot has happened since SQL-92.
I have shown you a few features today
I have shown you a few features today.

There are hundreds more to discover.

https://www.flickr.com/photos/mfoubister/25367243054/
@ModernSQL

modern-sql.com

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

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