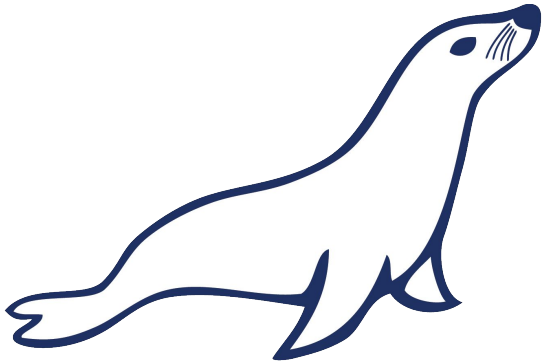


MariaDB®
FOUNDATION

Optimizing Queries Using CTEs and Window Functions

Vicențiu Ciorbaru
Software Engineer @ MariaDB Foundation

Agenda

- What are Common Table Expressions (CTEs)?
- What are Window Functions?
- Practical use cases
- Why are window functions fast?
- Development status in MariaDB

What are CTEs?

Syntax

```
WITH engineers AS (
      SELECT *
      FROM employees
      WHERE dept=”Engineering”
)
SELECT *
FROM engineers
WHERE ...
```

What are CTEs?

Syntax

```
WITH engineers AS (
      SELECT *
      FROM employees
      WHERE dept=”Engineering”
)
SELECT *
FROM engineers
WHERE ...
```

Keyword

What are CTEs?

Syntax

```
WITH engineers AS (
      SELECT *
      FROM employees
      WHERE dept=”Engineering”
)
SELECT *
FROM engineers
WHERE ...
```

CTE Name

What are CTEs?

Syntax

```
WITH engineers AS (
      SELECT *
      FROM employees
      WHERE dept=”Engineering”
)
SELECT *
FROM engineers
WHERE ...
```

CTE Body

What are CTEs?

Syntax

```
WITH engineers AS (
      SELECT *
      FROM employees
      WHERE dept=”Engineering”
)
SELECT *
FROM engineers
WHERE ...
```

CTE Usage

What are CTEs?

CTEs are similar to derived tables.

```
WITH engineers AS (
      SELECT *
      FROM employees
      WHERE dept=”Engineering”
)
SELECT *
FROM engineers
WHERE ...
```

```
SELECT *
FROM (SELECT *
      FROM employees
      WHERE dept=”Engineering”) AS engineers
WHERE ...
```

What are CTEs?

CTEs are more readable than derived tables.

```
WITH engineers AS (
      SELECT *
      FROM employees
      WHERE dept=”Engineering”
),
eu_engineers AS (
      SELECT *
      FROM engineers
      WHERE country IN (”NL”,...)
)
SELECT *
FROM eu_engineers
WHERE ...
```

```
SELECT *
FROM (SELECT *
      FROM (SELECT *
            FROM employees
            WHERE dept=”Engineering”) AS engineers
      WHERE country IN (”NL”,...))
WHERE ...
```

What are CTEs?

CTEs are more readable than derived tables.

```
WITH engineers AS (
      SELECT *
      FROM employees
      WHERE dept=”Engineering”
),
eu_engineers AS (
      SELECT *
      FROM engineers
      WHERE country IN (”NL”,...)
)
SELECT *
FROM eu_engineers
WHERE ...
```

Linear View

```
SELECT *
FROM (SELECT *
      FROM (SELECT *
            FROM employees
            WHERE dept=”Engineering”) AS engineers
      WHERE country IN (”NL”,...))
WHERE ...
```

Nested View

What are CTEs?

Example: Year-over-year comparisons

```
WITH sales_product_year AS (
  SELECT
    product,
    year(ship_date) as year,
    SUM(price) as total_amt
  FROM
    item_sales
  GROUP BY
    product, year
)
```

```
SELECT *
FROM
  sales_product_year CUR,
  sales_product_year PREV,
WHERE
  CUR.product = PREV.product AND
  CUR.year = PREV.year + 1 AND
  CUR.total_amt > PREV.total_amt
```

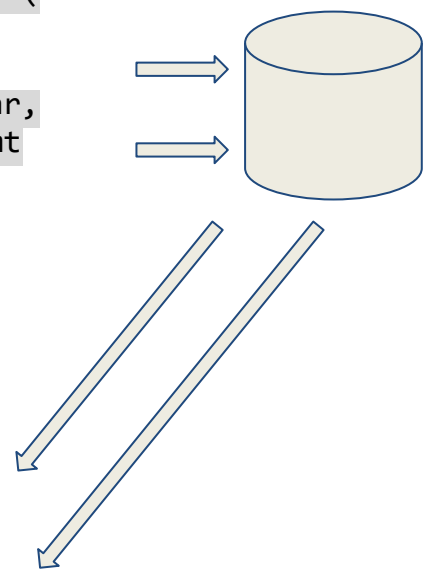
Summary on CTEs

- Identified by the WITH clause.
- Similar to derived tables in the FROM clause.
- More expressive and provide cleaner code.
- Can produce more efficient query plans.

CTE execution

Basic algorithm

```
WITH sales_product_year AS (
  SELECT
    product,
    year(ship_date) as year,
    SUM(price) as total_amt
  FROM
    item_sales
  GROUP BY
    product, year
)

SELECT *
FROM
  sales_product_year CUR,

  sales_product_year PREV,
WHERE
  CUR.product = PREV.product AND
  CUR.year = PREV.year + 1 AND
  CUR.total_amt > PREV.total_amt
```

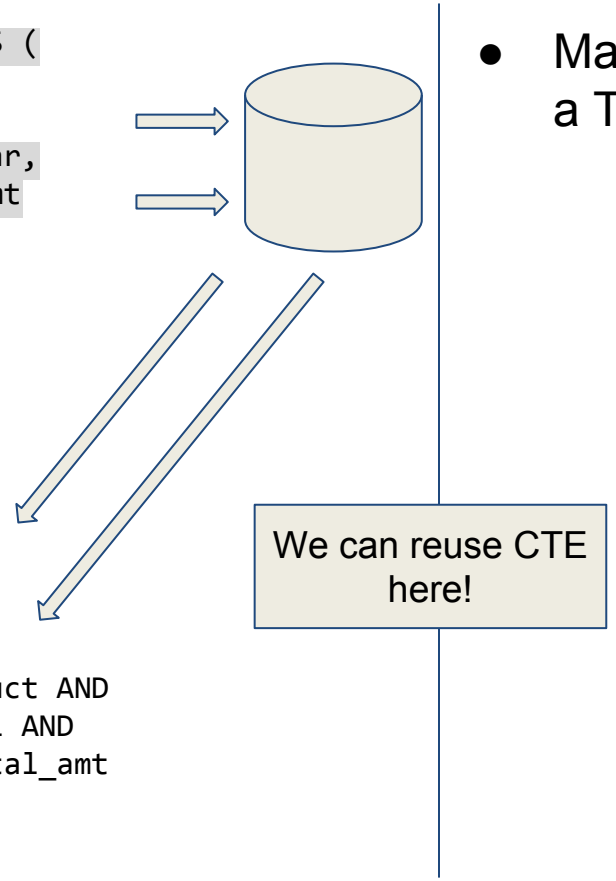


- Materialize each CTE occurrence into a Temporary Table
- Often Not optimal!

CTE optimization #1

CTE reuse

```
WITH sales_product_year AS (
  SELECT
    product,
    year(ship_date) as year,
    SUM(price) as total_amt
  FROM
    item_sales
  GROUP BY
    product, year
)

SELECT *
FROM
  sales_product_year CUR,

  sales_product_year PREV,
WHERE
  CUR.product = PREV.product AND
  CUR.year = PREV.year + 1 AND
  CUR.total_amt > PREV.total_amt
```

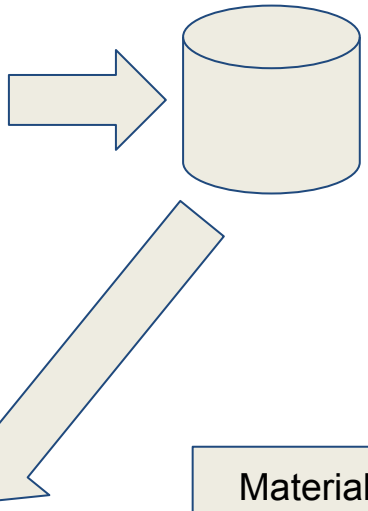


- Materialize each CTE occurrence into a Temporary Table

CTE optimization #1

CTE reuse

```
WITH sales_product_year AS (
  SELECT
    product,
    year(ship_date) as year,
    SUM(price) as total_amt
  FROM
    item_sales
  GROUP BY
    product, year
)

SELECT *
FROM
  sales_product_year CUR,

  sales_product_year PREV,
WHERE
  CUR.product = PREV.product AND
  CUR.year = PREV.year + 1 AND
  CUR.total_amt > PREV.total_amt
```



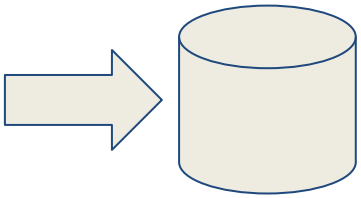
Materialize only once!

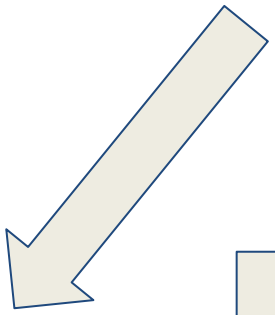
- Materialize each distinct CTE occurrence into a Temporary Table

CTE optimization #1

CTE reuse

```
WITH sales_product_year AS (
  SELECT
    product,
    year(ship_date) as year,
    SUM(price) as total_amt
  FROM
    item_sales
  GROUP BY
    product, year
)

SELECT *
FROM
  sales_product_year CUR,

  sales_product_year PREV,
WHERE
  CUR.product = PREV.product AND
  CUR.year = PREV.year + 1 AND
  CUR.total_amt > PREV.total_amt
```





Materialize only once!

- Materialize each distinct CTE occurrence into a Temporary Table
- Not compatible with other optimizations.

CTE optimization #2

CTE merging

```
WITH engineers AS (
  SELECT * FROM EMPLOYEES
  WHERE
    dept='Development'
)
SELECT
  ...
FROM
  engineers E,
  support_cases SC
WHERE
  E.name=SC.assignee and
  SC.created='2017-04-10' and
  E.location='New York'
```

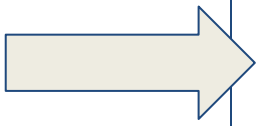
Requirements:

- CTE is used in a JOIN, no GROUP BY, DISTINCT, etc.

CTE optimization #2

CTE merging

```
WITH engineers AS (
  SELECT * FROM EMPLOYEES
  WHERE
    dept='Development'
)
SELECT
  ...
FROM
  engineers E,
  support_cases SC
WHERE
  E.name=SC.assignee and
  SC.created='2017-04-10' and
  E.location='New York'
```



```
SELECT
  ...
FROM
  employees E,
  support_cases SC
WHERE
  E.name=SC.assignee and
  SC.created='2017-04-10' and
  E.location='New York'
  E.dept='Development'
```

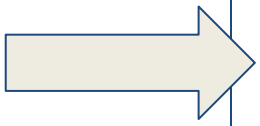
Requirements:

- CTE is used in a JOIN, no GROUP BY, DISTINCT, etc.

CTE optimization #2

CTE merging

```
WITH engineers AS (
  SELECT * FROM EMPLOYEES
  WHERE
    dept='Development'
)
SELECT
  ...
FROM
  engineers E,
  support_cases SC
WHERE
  E.name=SC.assignee and
  SC.created='2017-04-10' and
  E.location='New York'
```



```
SELECT
  ...
FROM
  employees E,
  support_cases SC
WHERE
  E.name=SC.assignee and
  SC.created='2017-04-10' and
  E.location='New York'
  E.dept='Development'
```

Requirements:

- CTE is used in a JOIN, no GROUP BY, DISTINCT, etc.

- CTE merged into parent join.
- Now optimizer can pick any query plan.
- Same algorithm is used for VIEWS (ALGORITHM = MERGE)

CTE optimization #3

Condition pushdown

```
WITH sales_per_year AS (
  SELECT
    year(order.date) AS year
    sum(order.amount) AS sales
  FROM
    order
  GROUP BY
    year
)
SELECT *
FROM sales_per_year
WHERE
  year in ('2015','2016')
```

CTE optimization #3

Condition pushdown

```
WITH sales_per_year AS (
  SELECT
    year(order.date) AS year
    sum(order.amount) AS sales
  FROM
    order
  GROUP BY
    year
)
SELECT *
FROM sales_per_year
WHERE
  year in ('2015','2016')
```

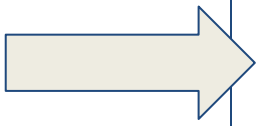
Requirements:

- Merging is not possible (GROUP BY exists)
- Conditions in outer select

CTE optimization #3

Condition pushdown

```
WITH sales_per_year AS (
  SELECT
    year(order.date) AS year
    sum(order.amount) AS sales
  FROM
    order
  GROUP BY
    year
)
SELECT *
FROM sales_per_year
WHERE
  year in ('2015','2016')
```



```
WITH sales_per_year AS (
  SELECT
    year(order.date) as year
    sum(order.amount) as sales
  FROM
    order
  WHERE
     year in ('2015','2016')
  GROUP BY
    year
)
SELECT *
FROM sales_per_year
```

Requirements:

- Merging is not possible (GROUP BY exists)
- Conditions in outer select

CTE optimization #3

Condition pushdown

- Makes temporary tables smaller.
- Can filter out whole groups.
- Works for derived tables and views.

- Implemented as a GSoC project:

"Pushing conditions into non-mergeable views and derived tables in MariaDB"

```
WITH sales_per_year AS (
  SELECT
    year(order.date) as year
    sum(order.amount) as sales
  FROM
    order
  WHERE
     year in ('2015','2016')
  GROUP BY
    year
)
SELECT *
FROM sales_per_year
```

CTE Optimizations Summary

	CTE Merge	**Condition pushdown**	CTE reuse
MariaDB 10.2	✔	✔	✘
MS SQL Server	✔	✔	✘
PostgreSQL	✘	✘	✔
MySQL 8.0.0-labs-optimizer	✔	✘	✔*

- Merge and condition pushdown are most important
  - Can not be used at the same time as CTE reuse
- PostgreSQL considers CTEs optimization barriers
- MySQL (8.0) tries merging, otherwise reuse

What are window functions?

- Similar to aggregate functions
  - Computed over a sequence of rows
- But they provide one result per row
  - Like regular functions!
- Identified by the OVER clause.

What are window functions?

Let's start with a "function like" example

```
SELECT

      email, first_name,
      last_name, account_type
FROM users
ORDER BY email;
```

```
+-------------------------+------------+-----------+--------------+
| email                   | first_name | last_name | account_type |
+-------------------------+------------+-----------+--------------+
| admin@boss.org          | Admin      | Boss      | admin        |
| bob.carlsen@foo.bar     | Bob        | Carlsen   | regular      |
| eddie.stevens@data.org  | Eddie      | Stevens   | regular      |
| john.smith@xyz.org      | John       | Smith     | regular      |
| root@boss.org           | Root       | Chief     | admin        |
+-------------------------+------------+-----------+--------------+
```

What are window functions?

Let's start with a "function like" example

```
SELECT
      row_number() over () as rnum,
      email, first_name,
      last_name, account_type
FROM users
ORDER BY email;
```

```
+------+-------------------------+------------+-----------+--------------+
| rnum | email                   | first_name | last_name | account_type |
+------+-------------------------+------------+-----------+--------------+
|    1 | admin@boss.org          | Admin      | Boss      | admin        |
|    2 | bob.carlsen@foo.bar     | Bob        | Carlsen   | regular      |
|    3 | eddie.stevens@data.org  | Eddie      | Stevens   | regular      |
|    4 | john.smith@xyz.org      | John       | Smith     | regular      |
|    5 | root@boss.org           | Root       | Chief     | admin        |
+------+-------------------------+------------+-----------+--------------+
```

What are window functions?

Let's start with a "function like" example

```
SELECT
      row_number() over () as rnum,
      email, first_name,
      last_name, account_type
FROM users
ORDER BY email;
```

This order is not deterministic!

rnum	email	first_name	last_name	account_type
1	admin@boss.org	Admin	Boss	admin
2	bob.carlsen@foo.bar	Bob	Carlsen	regular
3	eddie.stevens@data.org	Eddie	Stevens	regular
4	john.smith@xyz.org	John	Smith	regular
5	root@boss.org	Root	Chief	admin

What are window functions?

Let's start with a "function like" example

```
SELECT
      row_number() over () as rnum,
      email, first_name,
      last_name, account_type
FROM users
ORDER BY email;
```

This is also valid!

```
+------+-----------------------+------------+-----------+--------------+
| rnum | email                 | first_name | last_name | account_type |
+------+-----------------------+------------+-----------+--------------+
|    2 | admin@boss.org        | Admin      | Boss      | admin        |
|    1 | bob.carlsen@foo.bar   | Bob        | Carlsen   | regular      |
|    3 | eddie.stevens@data.org| Eddie      | Stevens   | regular      |
|    5 | john.smith@xyz.org    | John       | Smith     | regular      |
|    4 | root@boss.org         | Root       | Chief     | admin        |
+------+-----------------------+------------+-----------+--------------+
```

What are window functions?

Let's start with a "function like" example

```
SELECT
      row_number() over () as rnum,
      email, first_name,
      last_name, account_type
FROM users
ORDER BY email;
```

And this one...

```
+------+-----------------------+------------+-----------+--------------+
| rnum | email                 | first_name | last_name | account_type |
+------+-----------------------+------------+-----------+--------------+
|    5 | admin@boss.org        | Admin      | Boss      | admin        |
|    4 | bob.carlsen@foo.bar   | Bob        | Carlsen   | regular      |
|    3 | eddie.stevens@data.org| Eddie      | Stevens   | regular      |
|    2 | john.smith@xyz.org    | John       | Smith     | regular      |
|    1 | root@boss.org         | Root       | Chief     | admin        |
+------+-----------------------+------------+-----------+--------------+
```

What are window functions?

Let's start with a "function like" example

```
SELECT
      row_number() over (ORDER BY email) as rnum,
      email, first_name,
      last_name, account_type
FROM users
ORDER BY email;
```

Now only this one is valid!

```
+------+-----------------------+------------+-----------+--------------+
| rnum | email                 | first_name | last_name | account_type |
+------+-----------------------+------------+-----------+--------------+
|    1 | admin@boss.org        | Admin      | Boss      | admin        |
|    2 | bob.carlsen@foo.bar   | Bob        | Carlsen   | regular      |
|    3 | eddie.stevens@data.org| Eddie      | Stevens   | regular      |
|    4 | john.smith@xyz.org    | John       | Smith     | regular      |
|    5 | root@boss.org         | Root       | Chief     | admin        |
+------+-----------------------+------------+-----------+--------------+
```

What are window functions?

Let's start with a "function like" example

```
SELECT
      row_number() over (ORDER BY email) as rnum,
      email, first_name,
      last_name, account_type
FROM users
ORDER BY email;
```

How do we "group" by account type?

rnum	email	first_name	last_name	account_type
1	admin@boss.org	Admin	Boss	admin
2	bob.carlsen@foo.bar	Bob	Carlsen	regular
3	eddie.stevens@data.org	Eddie	Stevens	regular
4	john.smith@xyz.org	John	Smith	regular
5	root@boss.org	Root	Chief	admin

What are window functions?

Let's start with a "function like" example

```
SELECT
      row_number() over (PARTITION BY account_type ORDER BY email) as rnum,
      email, first_name,
      last_name, account_type
FROM users
ORDER BY account_type, email;
```

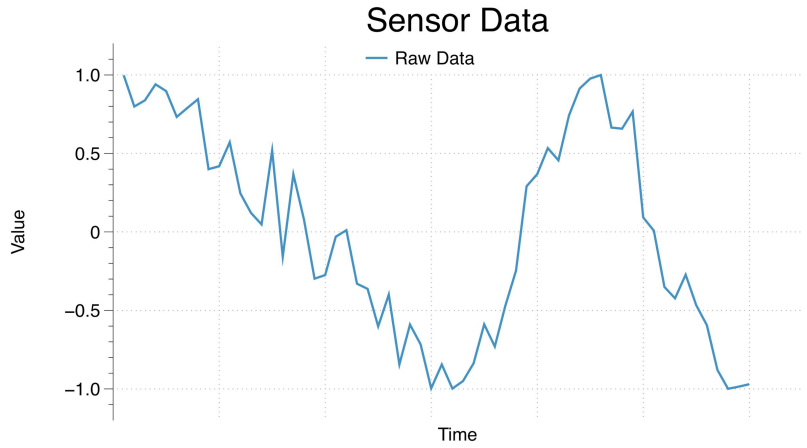
row_number() resets for every partition

```
+------+-----------------------+------------+-----------+--------------+
| rnum | email                 | first_name | last_name | account_type |
+------+-----------------------+------------+-----------+--------------+
|    1 | admin@boss.org        | Admin      | Boss      | admin        |
|    2 | root@boss.org         | Root       | Chief     | admin        |
|    1 | bob.carlsen@foo.bar   | Bob        | Carlsen   | regular      |
|    2 | eddie.stevens@data.org| Eddie      | Stevens   | regular      |
|    3 | john.smith@xyz.org    | John       | Smith     | regular      |
+------+-----------------------+------------+-----------+--------------+
```

What are window functions?

How about that aggregate similarity?

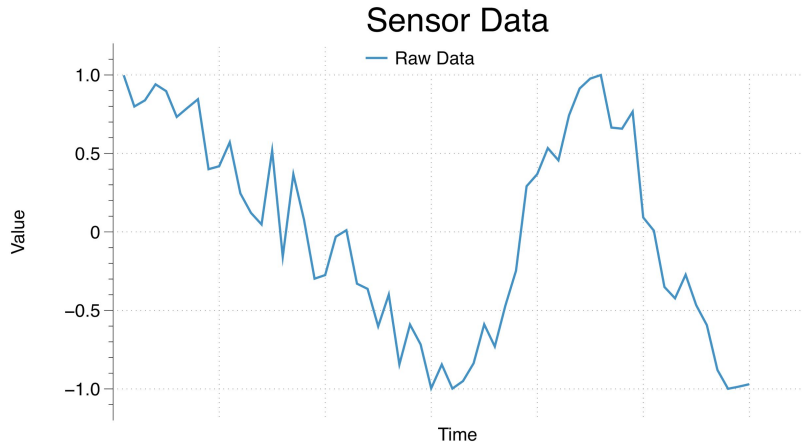
```
SELECT
      time, value
FROM data_points
ORDER BY time;
```



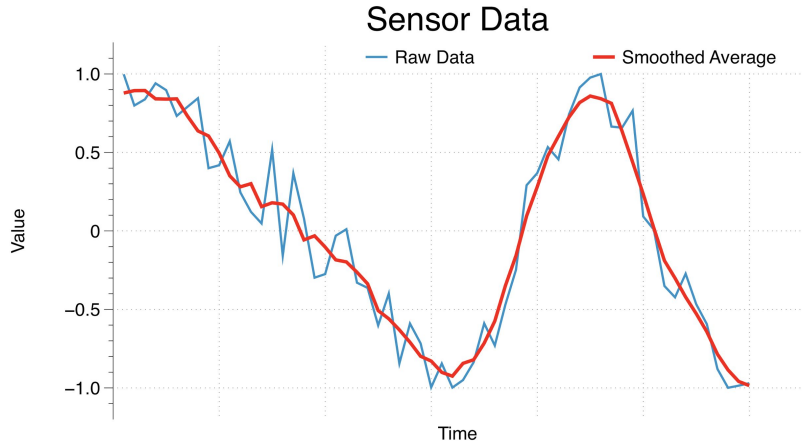
What are window functions?

How about that aggregate similarity?

```
SELECT
      time, value
FROM data_points
ORDER BY time;
```



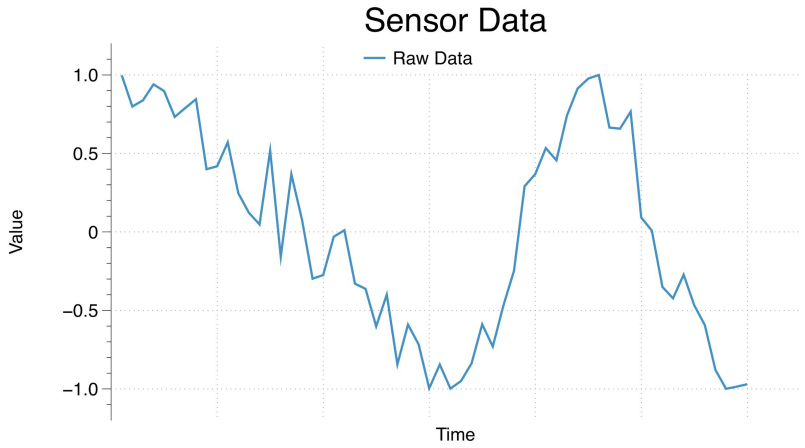

```
SELECT
      time, value
      avg(value) over (ORDER BY time
                       ),

FROM data_points
ORDER BY time;
```



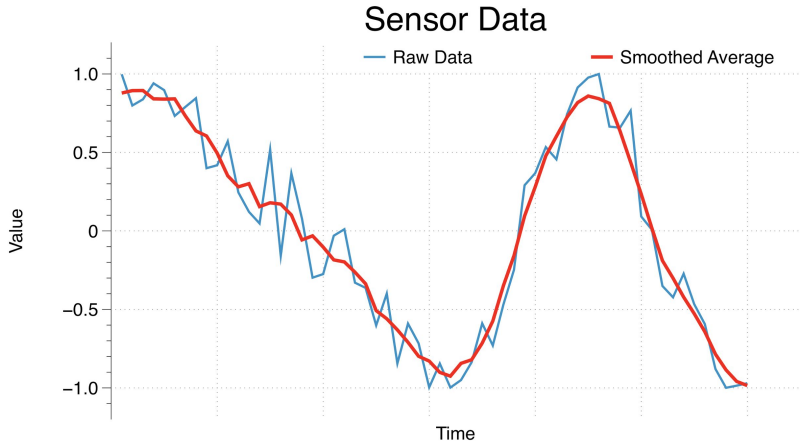
What are window functions?

How about that aggregate similarity?

```
SELECT
      time, value
FROM data_points
ORDER BY time;
```



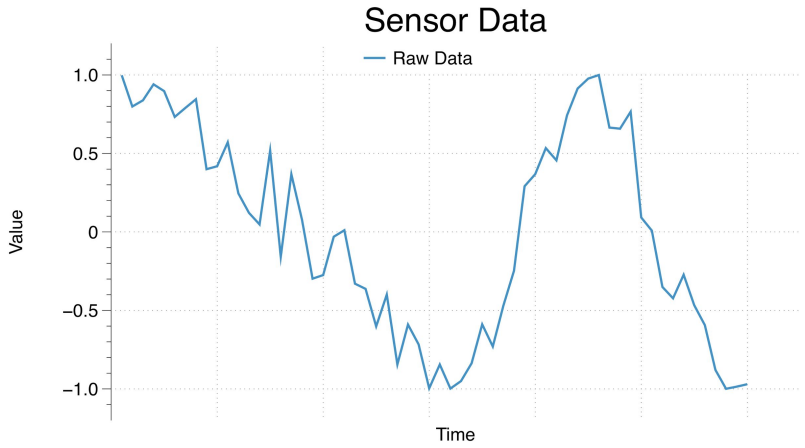

```
SELECT
      time, value
      avg(value) over (ORDER BY time
                       ROWS BETWEEN 3 PRECEDING
                                AND 3 FOLLOWING),
FROM data_points
ORDER BY time;
```



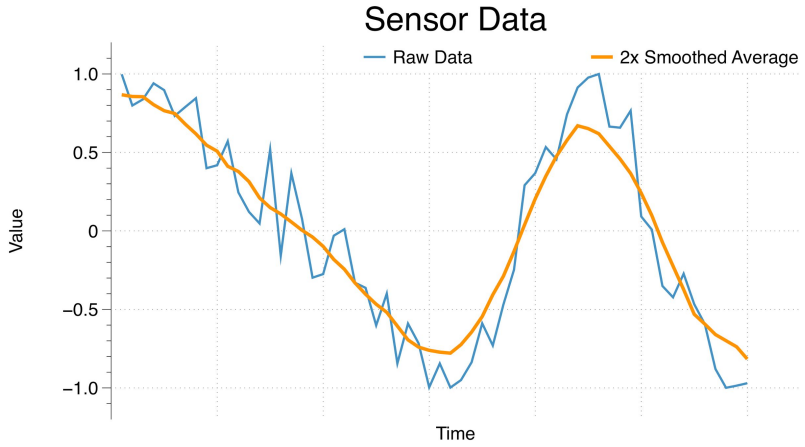
What are window functions?

How about that aggregate similarity?

```
SELECT
      time, value
FROM data_points
ORDER BY time;
```




```
SELECT
      time, value
      avg(value) over (ORDER BY time
                       ROWS BETWEEN 6 PRECEDING
                                AND 6 FOLLOWING),
FROM data_points
ORDER BY time;
```



What are window functions?

So how do frames work?

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 1 PRECEDING
                     AND 1 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |      |
| 11:00:00 |     5 |      |
| 12:00:00 |     4 |      |
| 13:00:00 |     4 |      |
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 2 PRECEDING
                     AND 2 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |      |
| 11:00:00 |     5 |      |
| 12:00:00 |     4 |      |
| 13:00:00 |     4 |      |
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

What are window functions?

So how do frames work?

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 1 PRECEDING
                     AND 1 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |    7 |   (2 + 5)
| 11:00:00 |     5 |      |
| 12:00:00 |     4 |      |
| 13:00:00 |     4 |      |
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 2 PRECEDING
                     AND 2 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |   11 |   (2 + 5 + 4)
| 11:00:00 |     5 |      |
| 12:00:00 |     4 |      |
| 13:00:00 |     4 |      |
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

What are window functions?

So how do frames work?

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 1 PRECEDING
                     AND 1 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |    7 |   (2 + 5)
| 11:00:00 |     5 |   11 |   (2 + 5 + 4)
| 12:00:00 |     4 |      |
| 13:00:00 |     4 |      |
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 2 PRECEDING
                     AND 2 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |   11 |   (2 + 5 + 4)
| 11:00:00 |     5 |   15 |   (2 + 5 + 4 + 4)
| 12:00:00 |     4 |      |
| 13:00:00 |     4 |      |
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

What are window functions?

So how do frames work?

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 1 PRECEDING
                     AND 1 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |    7 |   (2 + 5)
| 11:00:00 |     5 |   11 |   (2 + 5 + 4)
| 12:00:00 |     4 |   13 |   (5 + 4 + 4)
| 13:00:00 |     4 |      |
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 2 PRECEDING
                     AND 2 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |   11 |   (2 + 5 + 4)
| 11:00:00 |     5 |   15 |   (2 + 5 + 4 + 4)
| 12:00:00 |     4 |   16 |   (2 + 5 + 4 + 4 + 1)
| 13:00:00 |     4 |      |
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

What are window functions?

So how do frames work?

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 1 PRECEDING
                     AND 1 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |    7 |   (2 + 5)
| 11:00:00 |     5 |   11 |   (2 + 5 + 4)
| 12:00:00 |     4 |   13 |   (5 + 4 + 4)
| 13:00:00 |     4 |    9 |   (4 + 4 + 1)
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 2 PRECEDING
                     AND 2 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |   11 |   (2 + 5 + 4)
| 11:00:00 |     5 |   15 |   (2 + 5 + 4 + 4)
| 12:00:00 |     4 |   16 |   (2 + 5 + 4 + 4 + 1)
| 13:00:00 |     4 |   19 |   (5 + 4 + 4 + 1 + 5)
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

What are window functions?

So how do frames work?

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 1 PRECEDING
                     AND 1 FOLLOWING)
FROM data_points
ORDER BY time;
```

Every new row adds a value and removes a value!

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |    7 |   (2 + 5)
| 11:00:00 |     5 |   11 |   (2 + 5 + 4)
| 12:00:00 |     4 |   13 |   (5 + 4 + 4)
| 13:00:00 |     4 |    9 |   (4 + 4 + 1)
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 2 PRECEDING
                     AND 2 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |   11 |   (2 + 5 + 4)
| 11:00:00 |     5 |   15 |   (2 + 5 + 4 + 4)
| 12:00:00 |     4 |   16 |   (2 + 5 + 4 + 4 + 1)
| 13:00:00 |     4 |   19 |   (5 + 4 + 4 + 1 + 5)
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

What are window functions?

So how do frames work?

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 1 PRECEDING
                     AND 1 FOLLOWING)
FROM data_points
ORDER BY time;
```

We can do “on-line” computation!

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |    7 |   (2 + 5)
| 11:00:00 |     5 |   11 |   (2 + 5 + 4)
| 12:00:00 |     4 |   13 |   (5 + 4 + 4)
| 13:00:00 |     4 |    9 |   (4 + 4 + 1)
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 2 PRECEDING
                     AND 2 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |   11 |   (2 + 5 + 4)
| 11:00:00 |     5 |   15 |   (2 + 5 + 4 + 4)
| 12:00:00 |     4 |   16 |   (2 + 5 + 4 + 4 + 1)
| 13:00:00 |     4 |   19 |   (5 + 4 + 4 + 1 + 5)
| 14:00:00 |     1 |      |
| 15:00:00 |     5 |      |
| 15:00:00 |     2 |      |
| 15:00:00 |     2 |      |
+----------+-------+------+
```

What are window functions?

So how do frames work?

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 1 PRECEDING
                     AND 1 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |    7 |   (2 + 5)
| 11:00:00 |     5 |   11 |   (2 + 5 + 4)
| 12:00:00 |     4 |   13 |   (5 + 4 + 4)
| 13:00:00 |     4 |    9 |   (4 + 4 + 1)
| 14:00:00 |     1 |   10 |   (4 + 1 + 5)
| 15:00:00 |     5 |    8 |   (1 + 5 + 2)
| 15:00:00 |     2 |    9 |   (5 + 2 + 2)
| 15:00:00 |     2 |    4 |   (2 + 2)
+----------+-------+------+
```

```
SELECT
      time, value
      sum(value) OVER (
            ORDER BY time
            ROWS BETWEEN 2 PRECEDING
                     AND 2 FOLLOWING)
FROM data_points
ORDER BY time;
```

```
+----------+-------+------+
| time     | value | sum  |
+----------+-------+------+
| 10:00:00 |     2 |   11 |   (2 + 5 + 4)
| 11:00:00 |     5 |   15 |   (2 + 5 + 4 + 4)
| 12:00:00 |     4 |   16 |   (2 + 5 + 4 + 4 + 1)
| 13:00:00 |     4 |   19 |   (5 + 4 + 4 + 1 + 5)
| 14:00:00 |     1 |   16 |   (4 + 4 + 1 + 5 + 2)
| 15:00:00 |     5 |   14 |   (4 + 1 + 5 + 2 + 2)
| 15:00:00 |     2 |   10 |   (1 + 5 + 2 + 2)
| 15:00:00 |     2 |    9 |   (5 + 2 + 2)
+----------+-------+------+
```

Scenario 1 - Regular SQL

Given a set of bank transactions,
compute the account balance after each transaction.

```
SELECT timestamp, transaction_id, customer_id, amount,
FROM transactions
ORDER BY customer_id, timestamp;
```

timestamp	transaction_id	customer_id	amount
2016-09-01 10:00:00	1	1	1000
2016-09-01 11:00:00	2	1	-200
2016-09-01 12:00:00	3	1	-600
2016-09-01 13:00:00	5	1	400
2016-09-01 12:10:00	4	2	300
2016-09-01 14:00:00	6	2	500
2016-09-01 15:00:00	7	2	400

Scenario 1 - Regular SQL

Given a set of bank transactions,
compute the account balance after each transaction.

```
SELECT timestamp, transaction_id, customer_id, amount,
       (SELECT sum(amount)
        FROM transactions AS t2
        WHERE t2.customer_id = t1.customer_id AND
              t2.timestamp <= t1.timestamp) AS balance
FROM transactions AS t1
ORDER BY customer_id, timestamp;
```

```
+---------------------+----------------+-------------+--------+---------+
| timestamp           | transaction_id | customer_id | amount | balance |
+---------------------+----------------+-------------+--------+---------+
| 2016-09-01 10:00:00 |              1 |           1 |   1000 |    1000 |
| 2016-09-01 11:00:00 |              2 |           1 |   -200 |     800 |
| 2016-09-01 12:00:00 |              3 |           1 |   -600 |     200 |
| 2016-09-01 13:00:00 |              5 |           1 |    400 |     600 |
| 2016-09-01 12:10:00 |              4 |           2 |    300 |     300 |
| 2016-09-01 14:00:00 |              6 |           2 |    500 |     800 |
| 2016-09-01 15:00:00 |              7 |           2 |    400 |    1200 |
+---------------------+----------------+-------------+--------+---------+
```

Scenario 1 - Window Functions

Given a set of bank transactions,
compute the account balance after each transaction.

```
SELECT timestamp, transaction_id, customer_id, amount,
       sum(amount) OVER (PARTITION BY customer_id
                         ORDER BY timestamp
                         ROWS BETWEEN UNBOUNDED PRECEDING AND
                                      CURRENT ROW) AS balance
FROM transactions AS t1
ORDER BY customer_id, timestamp;
```

timestamp	transaction_id	customer_id	amount	balance
2016-09-01 10:00:00	1	1	1000	1000
2016-09-01 11:00:00	2	1	-200	800
2016-09-01 12:00:00	3	1	-600	200
2016-09-01 13:00:00	5	1	400	600
2016-09-01 12:10:00	4	2	300	300
2016-09-01 14:00:00	6	2	500	800
2016-09-01 15:00:00	7	2	400	1200

Scenario 1 - Performance

Given a set of bank transactions,
compute the account balance after each transaction.

#Rows	Regular SQL (seconds)	Regular SQL + Index (seconds)	**Window Functions (seconds)**
10 000	0.29	0.01	**0.02**
100 000	2.91	0.09	**0.16**
1 000 000	29.1	2.86	**3.04**
10 000 000	346.3	90.97	**43.17**
100 000 000	4357.2	813.2	**514.24**

Practical Use Cases - Scenario 2

- “Top-N” queries
- Retrieve the top 5 earners by department.

Scenario 2 - Regular SQL

Retrieve the top 5 earners by department.

```
SELECT dept, name, salary
FROM employee_salaries
ORDER BY dept;
```

```
+-------+----------+--------+
| dept  | name     | salary |
+-------+----------+--------+
| Sales | John     |    200 |
| Sales | Tom      |    300 |
| Sales | Bill     |    150 |
| Sales | Jill     |    400 |
| Sales | Bob      |    500 |
| Sales | Axel     |    250 |
| Sales | Lucy     |    300 |
| Eng   | Tim      |   1000 |
| Eng   | Michael  |   2000 |
| Eng   | Andrew   |   1500 |
| Eng   | Scarlett |   2200 |
| Eng   | Sergei   |   3000 |
| Eng   | Kristian |   3500 |
| Eng   | Arnold   |   2500 |
| Eng   | Sami     |   2800 |
+-------+----------+--------+
```

Scenario 2 - Regular SQL

Retrieve the top 5 earners by department.

```
SELECT dept, name, salary
FROM employee_salaries AS t1
WHERE (SELECT count(*)
       FROM employee_salaries AS t2
       WHERE t1.name != t2.name AND
             t1.dept = t2.dept AND
             t2.salary > t1.salary) < 5
ORDER BY dept, salary DESC;
```

```
+-------+----------+--------+
| dept  | name     | salary |
+-------+----------+--------+
| Eng   | Kristian |   3500 |
| Eng   | Sergei   |   3000 |
| Eng   | Sami     |   2800 |
| Eng   | Arnold   |   2500 |
| Eng   | Scarlett |   2200 |
| Sales | Bob      |    500 |
| Sales | Jill     |    400 |
| Sales | Lucy     |    300 |
| Sales | Tom      |    300 |
| Sales | Axel     |    250 |
+-------+----------+--------+
```

Scenario 2 - Regular SQL

Retrieve the top 5 earners by department.

```
SELECT dept, name, salary
FROM employee_salaries AS t1
WHERE (SELECT count(*)
       FROM employee_salaries AS t2
       WHERE t1.name != t2.name AND
             t1.dept = t2.dept AND
             t2.salary > t1.salary) < 5
ORDER BY dept, salary DESC;
```

```
+-------+----------+--------+
| dept  | name     | salary |
+-------+----------+--------+
| Eng   | Kristian |   3500 |
| Eng   | Sergei   |   3000 |
| Eng   | Sami     |   2800 |
| Eng   | Arnold   |   2500 |
| Eng   | Scarlett |   2200 |
| Sales | Bob      |    500 |
| Sales | Jill     |    400 |
| Sales | Lucy     |    300 |
| Sales | Tom      |    300 |
| Sales | Axel     |    250 |
+-------+----------+--------+
```

What if I want a “rank” column?

Scenario 2 - Regular SQL

Retrieve the top 5 earners by department.

```
SELECT
     (SELECT count(*) + 1
      FROM employee_salaries as t2
      WHERE t1.name != t2.name and
            t1.dept = t2.dept and
            t2.salary > t1.salary)
            AS ranking,
     dept, name, salary
FROM employee_salaries AS t1
WHERE (SELECT count(*)
       FROM employee_salaries AS t2
       WHERE t1.name != t2.name AND
             t1.dept = t2.dept AND
             t2.salary > t1.salary) < 5
ORDER BY dept, salary DESC;
```

ranking	dept	name	salary
1	Eng	Kristian	3500
2	Eng	Sergei	3000
3	Eng	Sami	2800
4	Eng	Arnold	2500
5	Eng	Scarlett	2200
1	Sales	Bob	500
2	Sales	Jill	400
3	Sales	Lucy	300
3	Sales	Tom	300
5	Sales	Axel	250

What if I want a “rank” column?

Scenario 2 - Window Functions

Retrieve the top 5 earners by department.

```
SELECT
    rank() OVER (
          PARTITION BY dept
          ORDER BY salary DESC)
     AS ranking,
    dept, name, salary
FROM employee_salaries;
```

ranking	dept	name	salary
1	Eng	Kristian	3500
2	Eng	Sergei	3000
3	Eng	Sami	2800
4	Eng	Arnold	2500
5	Eng	Scarlett	2200
6	Eng	Michael	2000
7	Eng	Andrew	1500
8	Eng	Tim	1000
1	Sales	Bob	500
2	Sales	Jill	400
3	Sales	Tom	300
3	Sales	Lucy	300
5	Sales	Axel	250
6	Sales	John	200
7	Sales	Bill	150

Scenario 2 - Window Functions

Retrieve the top 5 earners by department.

```
SELECT
    rank() OVER (
          PARTITION BY dept
          ORDER BY salary DESC)
     AS ranking,
    dept, name, salary
FROM employee_salaries
WHERE ranking <= 5;
```

ranking	dept	name	salary
1	Eng	Kristian	3500
2	Eng	Sergei	3000
3	Eng	Sami	2800
4	Eng	Arnold	2500
5	Eng	Scarlett	2200
6	Eng	Michael	2000
7	Eng	Andrew	1500
8	Eng	Tim	1000
1	Sales	Bob	500
2	Sales	Jill	400
3	Sales	Tom	300
3	Sales	Lucy	300
5	Sales	Axel	250
6	Sales	John	200
7	Sales	Bill	150

Scenario 2 - Window Functions

Retrieve the top 5 earners by department.

```
SELECT
    rank() OVER (
          PARTITION BY dept
          ORDER BY salary DESC)
     AS ranking,
    dept, name, salary
FROM employee_salaries
WHERE ranking <= 5;
```

No Window Functions in the WHERE clause :(

ranking	dept	name	salary
1	Eng	Kristian	3500
2	Eng	Sergei	3000
3	Eng	Sami	2800
4	Eng	Arnold	2500
5	Eng	Scarlett	2200
6	Eng	Michael	2000
7	Eng	Andrew	1500
8	Eng	Tim	1000
1	Sales	Bob	500
2	Sales	Jill	400
3	Sales	Tom	300
3	Sales	Lucy	300
5	Sales	Axel	250
6	Sales	John	200
7	Sales	Bill	150

Scenario 2 - Window Functions

Retrieve the top 5 earners by department.

```
WITH salary_ranks AS (
    SELECT
        rank() OVER (
              PARTITION BY dept
              ORDER BY salary DESC)
         AS ranking,
        dept, name, salary
    FROM employee_salaries
)
SELECT *
FROM salary_ranks
WHERE ranking <= 5
ORDER BY dept, ranking;
```

```
+---------+-------+----------+--------+
| ranking | dept  | name     | salary |
+---------+-------+----------+--------+
|       1 | Eng   | Kristian |   3500 |
|       2 | Eng   | Sergei   |   3000 |
|       3 | Eng   | Sami     |   2800 |
|       4 | Eng   | Arnold   |   2500 |
|       5 | Eng   | Scarlett |   2200 |
|       1 | Sales | Bob      |    500 |
|       2 | Sales | Jill     |    400 |
|       3 | Sales | Lucy     |    300 |
|       3 | Sales | Tom      |    300 |
|       5 | Sales | Axel     |    250 |
+---------+-------+----------+--------+
```

Scenario 2 - Performance

Retrieve the top 5 earners by department.

#Rows	Regular SQL (seconds)	Regular SQL + Index (seconds)	**Window Functions (seconds)**
2 000	1.31	0.14	**0.00**
20 000	123.6	12.6	**0.02**
200 000	10000+	1539.79	**0.21**
2 000 000	...	...	**5.61**
20 000 000	...	...	**76.04**

Window functions summary

- Can help eliminate expensive subqueries.
- Can help eliminate self-joins.
- Make queries more readable.
- Make (some) queries faster.

Window Functions in MariaDB

- We support:
  - ROW_NUMBER, RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST, NTILE
  - FIRST_VALUE, LAST_VALUE, NTH_VALUE, LEAD, LAG
  - All regular aggregate functions except GROUP_CONCAT

Window Functions in MariaDB

- We do not (yet) support:
  - Time interval range-type frames
  - DISTINCT clause
  - GROUP_CONCAT function
  - Advanced window functions such as:

    PERCENTILE_CONT, PERCENTILE_DISC

Thank You!

Contact me at:
vicentiu@mariadb.org
vicentiu@ciorbaru.io

Blog: vicentiu.ciorbaru.io