Common Table Expressions (CTE) & Window Functions in MySQL 8.0

Øystein Grøvlen
Senior Principal Software Engineer
MySQL Optimizer Team, Oracle
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
Program Agenda

1. Common table expression
2. Window functions
Program Agenda

1. Common table expression
2. Window functions
Common Table Expression

Alternative to derived table

• A derived table is a subquery in the FROM clause

  SELECT ... FROM (subquery) AS derived, t1 ...

• Common Table Expression (CTE) is just like a derived table, but its declaration is put before the query block instead of in FROM clause

  WITH derived AS (subquery)
  SELECT ... FROM derived, t1 ...

• A CTE may precede SELECT/UPDATE/DELETE including sub-queries

  WITH derived AS (subquery)
  DELETE FROM t1 WHERE t1.a IN (SELECT b FROM derived);
Common Table Expression (CTE)

WITH cte_name [(<list of column names>)] AS (  
  SELECT ...  # Definition  
)  
[,] (<any number of other CTE definitions> ]  
<SELECT/UPDATE/DELETE statement>

WITH qn AS (SELECT a FROM t1)  
SELECT * from qn;

INSERT INTO t2  
WITH qn AS (SELECT 10*a AS a FROM t1)  
SELECT * from qn;

SELECT * FROM t1 WHERE t1.a IN  
(WITH cte as (SELECT * FROM t1 AS t2 LIMIT 1)  
SELECT a + 0 FROM cte);
Common Table Expression versus Derived Table

- Better readability
- Can be referenced multiple times
- Can refer to other CTEs
- Improved performance
Better readability

• Derived table:

```sql
SELECT ... 
FROM t1 LEFT JOIN ((SELECT ... FROM ...) AS dt JOIN t2 ON ...) ON ...
```

• CTE:

```sql
WITH dt AS (SELECT ... FROM ...) 
SELECT ... 
FROM t1 LEFT JOIN (dt JOIN t2 ON ...) ON ...
```
Can be referenced multiple times

- Derived table can not be referenced twice:

  ```sql
  SELECT ...
  FROM (SELECT a, b, SUM(c) s FROM t1 GROUP BY a, b) AS d1
       JOIN (SELECT a, b, SUM(c) s FROM t1 GROUP BY a, b) AS d2 ON d1.b = d2.a;
  ```

- CTE can:

  ```sql
  WITH d AS (SELECT a, b, SUM(c) s FROM t1 GROUP BY a, b)
  SELECT ... FROM d AS d1 JOIN d AS d2 ON d1.b = d2.a;
  ```
Can refer to other CTEs

• Derived tables can not refer to other derived tables:

```
SELECT ...
FROM (SELECT ... FROM ...) AS d1, (SELECT ... FROM d1 ...) AS d2 ...
```

ERROR: 1146 (42S02): Table ‘db.d1’ doesn’t exist

• CTEs can refer other CTEs:

```
WITH d1 AS (SELECT ... FROM ...),
    d2 AS (SELECT ... FROM d1 ...)
SELECT
FROM d1, d2 ...
```
Chained CTEs

Neat, but not very useful example

WITH cte1(txt) AS (SELECT "This "),
    cte2(txt) AS (SELECT CONCAT(cte1.txt,"is a ") FROM cte1),
    cte3(txt) AS (SELECT "nice query" UNION
                   SELECT "query that rocks" UNION
                   SELECT "query"),
    cte4(txt) AS (SELECT concat(cte2.txt, cte3.txt) FROM cte2, cte3)
SELECT MAX(txt), MIN(txt) FROM cte4;

<table>
<thead>
<tr>
<th>MAX(txt)</th>
<th>MIN(txt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a query that rocks</td>
<td>This is a nice query</td>
</tr>
</tbody>
</table>

1 row in set (0,00 sec)

Neat, but not very useful example
Better performance

• Derived table:
  – For derived tables that are materialized, two identical derived tables will be materialized. Performance problem (more space, more time, longer locks)
  – Similar with view references

• CTE:
  – Will be materialized once, regardless of how many references
DBT3 Query 15 Top Supplier Query

Using view

CREATE VIEW revenue0 (supplier_no, total_revenue) AS SELECT l_suppkey, SUM(l_extendedprice * (1 - l_discount)) FROM lineitem WHERE l_shipdate >= '1996-07-01' AND l_shipdate < DATE_ADD('1996-07-01', INTERVAL '90' day) GROUP BY l_suppkey;

SELECT s_suppkey, s_name, s_address, s_phone, total_revenue FROM supplier, revenue0 WHERE s_suppkey = supplier_no AND total_revenue = (SELECT MAX(total_revenue) FROM revenue0) ORDER BY s_suppkey;

Using CTE

WITH revenue0 (supplier_no, total_revenue) AS (SELECT l_suppkey, SUM(l_extendedprice * (1 - l_discount)) FROM lineitem WHERE l_shipdate >= '1996-07-01' AND l_shipdate < DATE_ADD('1996-07-01', INTERVAL '90' day) GROUP BY l_suppkey)

SELECT s_suppkey, s_name, s_address, s_phone, total_revenue FROM supplier, revenue0 WHERE s_suppkey = supplier_no AND total_revenue = (SELECT MAX(total_revenue) FROM revenue0) ORDER BY s_suppkey;
DBT-3 Query 15

Query Performance

Query Execution Time (seconds)

View

CTE
Recursive CTE

- A recursive CTE refers to itself in a subquery
- The “seed” SELECT is executed once to create the initial data subset, the recursive SELECT is repeatedly executed to return subsets of data until the complete result set is obtained.
- Recursion stops when an iteration does not generate any new rows
- Useful to dig in hierarchies (parent/child, part/subpart)
Recursive CTE

A simple example

Print 1 to 10:

WITH RECURSIVE qn AS
  ( SELECT 1 AS a
    UNION ALL
    SELECT 1+a FROM qn WHERE a<10
  )
SELECT * FROM qn;
Recursive CTE

**INSERT**

Insert 1 to 10:

**INSERT INTO numbers**
WITH RECURSIVE qn AS
( SELECT 1 AS a
  UNION ALL
  SELECT 1+a FROM qn WHERE a<10
)
SELECT * FROM qn;

**SELECT * FROM numbers;**

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
Date sequence

Missing dates

```
SELECT orderdate, SUM(totalprice) sales
FROM orders
GROUP BY orderdate
ORDER BY orderdate;
```

<table>
<thead>
<tr>
<th>orderdate</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-09-01</td>
<td>43129.83</td>
</tr>
<tr>
<td>2016-09-03</td>
<td>218347.61</td>
</tr>
<tr>
<td>2016-09-04</td>
<td>142568.40</td>
</tr>
<tr>
<td>2016-09-05</td>
<td>299244.83</td>
</tr>
<tr>
<td>2016-09-07</td>
<td>185991.79</td>
</tr>
</tbody>
</table>
```
WITH RECURSIVE dates(date) AS
  ( SELECT '2016-09-01'
      UNION ALL
      SELECT DATE_ADD(date, INTERVAL 1 DAY)
      FROM dates
      WHERE date < '2016-09-07'
    )
SELECT dates.date,
       COALESCE(SUM(totalprice), 0) sales
FROM dates LEFT JOIN orders
       ON dates.date = orders.orderdate
GROUP BY dates.date
ORDER BY dates.date;

<table>
<thead>
<tr>
<th>date</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-09-01</td>
<td>43129.83</td>
</tr>
<tr>
<td>2016-09-02</td>
<td>0.00</td>
</tr>
<tr>
<td>2016-09-03</td>
<td>218347.61</td>
</tr>
<tr>
<td>2016-09-04</td>
<td>142568.40</td>
</tr>
<tr>
<td>2016-09-05</td>
<td>299244.83</td>
</tr>
<tr>
<td>2016-09-06</td>
<td>0.00</td>
</tr>
<tr>
<td>2016-09-07</td>
<td>185991.79</td>
</tr>
</tbody>
</table>
Hierarchy Traversal
Employee database

CREATE TABLE employees (  
id INT PRIMARY KEY,  
name VARCHAR(100),  
manager_id INT,  
FOREIGN KEY (manager_id)  
REFERENCES employees(id) );

INSERT INTO employees VALUES  
(333, "Yasmina", NULL), # CEO  
(198, "John", 333), # John reports to 333  
(692, "Tarek", 333),  
(29, "Pedro", 198),  
(4610, "Sarah", 29),  
(72, "Pierre", 29),  
(123, "Adil", 692);
## Hierarchy Traversal

### List reporting chain

WITH RECURSIVE emp_ext (id, name, path) AS (  
  SELECT id, name, CAST(id AS CHAR(200))  
  FROM employees  
  WHERE manager_id IS NULL  
  UNION ALL  
  SELECT s.id, s.name,  
         CONCAT(m.path, ",", s.id)  
  FROM emp_ext m JOIN employees s  
       ON m.id=s.manager_id )  
SELECT * FROM emp_ext ORDER BY path;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>333</td>
<td>Yasmina</td>
<td>333</td>
</tr>
<tr>
<td>198</td>
<td>John</td>
<td>333,198</td>
</tr>
<tr>
<td>692</td>
<td>Tarek</td>
<td>333,692</td>
</tr>
<tr>
<td>29</td>
<td>Pedro</td>
<td>333,198,29</td>
</tr>
<tr>
<td>123</td>
<td>Adil</td>
<td>333,692,123</td>
</tr>
<tr>
<td>4610</td>
<td>Sarah</td>
<td>333,198,29,4610</td>
</tr>
<tr>
<td>72</td>
<td>Pierre</td>
<td>333,198,29,72</td>
</tr>
</tbody>
</table>
## Hierarchy Traversal

### List reporting chain

WITH RECURSIVE emp_ext (id, name, path) AS (  
  SELECT id, name, CAST(id AS CHAR(200))  
  FROM employees  
  WHERE manager_id IS NULL  
  UNION ALL  
  SELECT s.id, s.name,  
  CONCAT(m.path, ",", s.id)  
  FROM emp_ext m JOIN employees s  
  ON m.id=s.manager_id  
)  
SELECT * FROM emp_ext ORDER BY path;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>333</td>
<td>Yasmina</td>
<td>333</td>
</tr>
<tr>
<td>198</td>
<td>John</td>
<td>333,198</td>
</tr>
<tr>
<td>29</td>
<td>Pedro</td>
<td>333,198,29</td>
</tr>
<tr>
<td>4610</td>
<td>Sarah</td>
<td>333,198,29,4610</td>
</tr>
<tr>
<td>72</td>
<td>Pierre</td>
<td>333,198,29,72</td>
</tr>
<tr>
<td>692</td>
<td>Tarek</td>
<td>333,692</td>
</tr>
<tr>
<td>123</td>
<td>Adil</td>
<td>333,692,123</td>
</tr>
</tbody>
</table>
Program Agenda

1. Non recursive common table expression
2. Window functions
Window functions: what are they?

• A window function performs a calculation across a set of rows that are related to the current row, similar to an aggregate function.

• But unlike aggregate functions, a window function does not cause rows to become grouped into a single output row.

• Window functions can access values of other rows “in the vicinity” of the current row
Window function example

Sum up total salary for each department:

```
SELECT name, dept_id, salary, 
    SUM(salary) OVER (PARTITION BY dept_id) AS dept_total
FROM employee
ORDER BY dept_id, name;
```

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>dept_total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>75000</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>370000</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>370000</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>60000</td>
<td>370000</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>370000</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>370000</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>370000</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>130000</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>130000</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>370000</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>370000</td>
</tr>
</tbody>
</table>

The `OVER` keyword signals a window function

`PARTITION` == disjoint set of rows in result set
With GROUP BY

SELECT name, dept_id, salary,
    SUM(salary) AS dept_total
FROM employee GROUP BY dept_id
ORDER BY dept_id, name;

ERROR 1055 (42000): Expression #1 of SELECT list is not in GROUP BY clause and contains nonaggregated column 'mysql.employee.name' which is not functionally dependent on columns in GROUP BY clause; this is incompatible with sql_mode=only_full_group_by
**With GROUP BY**

```
SELECT /* name, */ dept_id, /* salary,*/ SUM(salary) AS dept_total
FROM employee GROUP BY dept_id
ORDER BY dept_id /*, name */;
```

<table>
<thead>
<tr>
<th>dept_id</th>
<th>dept_total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>75000</td>
</tr>
<tr>
<td>10</td>
<td>370000</td>
</tr>
<tr>
<td>20</td>
<td>130000</td>
</tr>
<tr>
<td>30</td>
<td>370000</td>
</tr>
</tbody>
</table>
## Window function example, with frame

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>75000</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>100000</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>60000</td>
<td>160000</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>220000</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>190000</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>210000</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>65000</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>130000</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>300000</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>370000</td>
</tr>
</tbody>
</table>

**SELECT** name, dept_id, salary, SUM(salary) 
**OVER** (PARTITION BY dept_id 
ORDER BY name 
ROWS 2 PRECEDING) total 
FROM employee 
ORDER BY dept_id, name;
Window function example, with frame

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>75000</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>100000</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>60000</td>
<td>160000</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>220000</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>190000</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>210000</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>65000</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>130000</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>300000</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>370000</td>
</tr>
</tbody>
</table>

SELECT name, dept_id, salary, SUM(salary) OVER (PARTITION BY dept_id ORDER BY name ROWS 2 PRECEDING) total
FROM employee
ORDER BY dept_id, name;
Window function example

```
SELECT name, dept_id, salary, 
AVG(salary) OVER w AS `avg`,
salary - AVG(salary) OVER w AS diff
FROM employee
WINDOW w AS (PARTITION BY dept_id)
ORDER BY diff DESC;
```

- i.e. find the employees with the largest difference between their wage and that of the department average
- Note: explicit window definition of “w”

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>average</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>185000</td>
<td>115000</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>74000</td>
<td>26000</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>74000</td>
<td>6000</td>
</tr>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>75000</td>
<td>0</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>65000</td>
<td>0</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>65000</td>
<td>0</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>74000</td>
<td>-4000</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>74000</td>
<td>-14000</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>60000</td>
<td>74000</td>
<td>-14000</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>185000</td>
<td>-115000</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>74000</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Implicit and explicit windows

• Windows can be implicit and unnamed:

  \[ \text{COUNT}(*) \text{ OVER (PARTITION BY dept\_ID)} \]

• Windows can be defined and named via the windows clause:

  \[ \text{SELECT COUNT}(*) \text{ OVER w FROM t} \]
  \[ \text{WINDOW w as (PARTITION BY dept\_id)} \]

• Allows sharing of windows between several window functions

• Avoids redundant windowing steps since more functions can be evaluated in the same step
Types of window functions

• Aggregates
  – COUNT, SUM, AVG, MAX, MIN  + more to come

• Ranking
  – RANK, DENSE_RANK, PERCENT_RANK,
  – CUME_DIST, ROW_NUMBER

• Analytical
  – NTILE, LEAD, LAG
  – NTH, FIRST_VALUE, LAST_VALUE

Blue ones use frames, all obey partitions
Syntax for window specification

window specification ::=  
  [ existing window name ]  
  [PARTITION BY expr-1, ... ]  
  [ORDER BY expr-1, ... [DESC] ]  
  [ frame clause ]

frame clause ::= { ROWS | RANGE } { start | between }

start ::= { CURRENT ROW | UNBOUNDED PRECEDING | n PRECEDING}

between ::= BETWEEN bound-1 AND bound-2

bound ::= start | UNBOUNDED FOLLOWING | n FOLLOWING
Frame clause bound

partition

n PRECEDING

CURRENT ROW

m FOLLOWING

UNBOUNDED PRECEDING

UNBOUNDED FOLLOWING
RANGE frame example

<table>
<thead>
<tr>
<th>date</th>
<th>amount</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-01-01</td>
<td>100.50</td>
<td>300.50</td>
</tr>
<tr>
<td>2017-01-01</td>
<td>200.00</td>
<td>300.50</td>
</tr>
<tr>
<td>2017-01-02</td>
<td>200.00</td>
<td>500.50</td>
</tr>
<tr>
<td>2017-01-03</td>
<td>200.00</td>
<td>700.50</td>
</tr>
<tr>
<td>2017-01-05</td>
<td>200.00</td>
<td>900.50</td>
</tr>
<tr>
<td>2017-01-10</td>
<td>200.00</td>
<td>700.00</td>
</tr>
<tr>
<td>2017-01-10</td>
<td>100.00</td>
<td>700.00</td>
</tr>
<tr>
<td>2017-01-11</td>
<td>200.00</td>
<td>700.00</td>
</tr>
</tbody>
</table>

Find the sum of payments within the last 8 days

```
SELECT date, amount,
    SUM(amount) OVER w AS `sum` FROM payments
WINDOW w AS
    (ORDER BY date
        RANGE BETWEEN INTERVAL 1 WEEK PRECEDING
        AND CURRENT ROW)
ORDER BY date;
```

Current row's date is the 10th, so first row in range is the 3rd. Frame cardinality is 4 due to peer in next row.

For Jan 5, the frame cardinality is 5, and sum is 900.50.
When are they evaluated?

• After GROUP BY/ HAVING
• Before final ORDER BY, DISTINCT, LIMIT
• You can have several window functions and several different windows
• To filter on window function’s value, use a subquery, e.g.

```
SELECT *
FROM (  
    SELECT SUM(salary) OVER (PARTITION BY dept_id) `sum`
    FROM employee
  ) AS s
WHERE `sum` < 100000;
```
Logical flow

Input goes into a tmp table
Sort for
PARTITION BY
and
ORDER BY

• Tmp table between each windowing step
  • (in-mem if result set can fit †)
• Streamable wfs vs buffered
  • Depends on wf and frame
• Buffered: re-read rows
  • O(rows * frame size)
  • Move frame for SUM 1 row
  • Optimization: Invert by subtraction, add new row.

† cf. variables tmp_table_size, max_heap_table_size
Streamable evaluation

Accumulate the salary in each department as sum

```
SELECT name, dept_id, salary, 
    SUM(salary) OVER (PARTITION BY dept_id ORDER BY name 
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS `sum`
FROM employee;
```

Just accumulate as we see rows
Non-streamable evaluation

Sum two preceding rows and the current row

```sql
SELECT name, dept_id, salary,
    SUM(salary) OVER (PARTITION BY dept_id ORDER BY name
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS `sum`
FROM employee;
```

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>75000</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>100000</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>60000</td>
<td>160000</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>220000</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>190000</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>210000</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>65000</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>130000</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>300000</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>370000</td>
</tr>
</tbody>
</table>

When evaluating Michael, subtract Ed's contribution, add Michael

or just evaluate entire frame over again (non-optimized). In both cases we need re-visit rows.
EXPLAIN FORMAT=JSON
SELECT name, dept_id, salary, 
SUM(salary) OVER (PARTITION BY dept_id ORDER BY name 
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS `sum`
FROM employee;

: 
"windows": [ 
  { 
    "name": "<unnamed window>", 
    "using_filesort": true, 
    "frame_buffer": { 
      "using_temporary_table": true, 
      "optimized_frame_evaluation": true 
    }, 
    "functions": [ "sum" ] 
  } 
],
RANK

SELECT name, dept_id AS dept, salary, 
  \textbf{RANK}() \text{ OVER w AS `rank`} 
FROM employee 
  \text{ WINDOW w AS (PARTITION BY dept_id 
ORDER BY salary DESC)};
DENSE_RANK

SELECT name, dept_id AS dept, salary,
  RANK() OVER w AS `rank`,
  DENSE_RANK() OVER w AS dense
FROM employee
  WINDOW w AS (PARTITION BY dept_id
    ORDER BY salary DESC);

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>rank</th>
<th>dense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>70000</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

DENSE_RANK doesn't skip
**ROW_NUMBER**

```sql
SELECT name, dept_id AS dept, salary,
    RANK() OVER w AS `rank`,
    DENSE_RANK() OVER w AS dense,
    ROW_NUMBER() OVER w AS `rowno`
FROM employee
WINDOW w AS (PARTITION BY dept_id
ORDER BY salary DESC);
```

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>rank</th>
<th>dense</th>
<th>rowno</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>70000</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Implicit and explicit windows

A window definition can inherit from another window definition in its specification, adding detail, no override

```sql
SELECT name, dept_id,
    COUNT(*) OVER w1 AS cnt1,
    COUNT(*) OVER w2 AS cnt2
FROM employee
WINDOW w1 AS (PARTITION BY dept_id),
    w2 AS (w1 ORDER BY name)
ORDER BY dept_id, name;
```

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>cnt1</th>
<th>cnt2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Levedev</td>
<td>20</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Want to learn more?

Want to learn more?

• Thursday 12:50pm: Recursive Query Throwdown in MySQL 8 (Bill Karwin)
• MySQL Server Team blog
  – http://mysqlserverteam.com/
• My blog:
  – http://oysteing.blogspot.com/
• MySQL forums:
Hardware and Software
Engineered to Work Together
LEAD, LAG

Returns value evaluated at the row that is offset rows after/before the current row within
the partition; if there is no such row, instead return default (which must be of the same
type as value).

Both offset and default are evaluated with respect to the current row. If omitted, offset
defaults to 1 and default to null

lead or lag function ::= { LEAD | LAG } ( expr [ , offset [ , default expression> ] ] )
[ RESPECT NULLS ]

Note: “IGNORE NULLS” not supported, RESPECT NULLS is default but can be specified.
FIRST_VALUE, LAST_VALUE, NTH_VALUE

Returns value evaluated at the first, last, nth in the frame of the current row within the partition; if there is no nth row (frame is too small), the NTH_VALUE returns NULL.

first or last value ::= { FIRST_VALUE | LAST_VALUE } ( expr )
[ RESPECT NULLS ]

nth_value ::= NTH_VALUE ( expr, nth-row ) [FROM FIRST]
[ RESPECT NULLS ]

Note: “IGNORE NULLS” is not supported, RESPECT NULLS is used but can be specified.
Note: For NTH_VALUE, “FROM LAST” is not supported, FROM FIRST is used but can be specified.
FIRST_VALUE “in frame”

SELECT name, dept_id AS dept, salary, 
SUM(salary) OVER w AS `sum`, 
FIRST_VALUE(salary) OVER w AS `first` 
FROM employee

WINDOW w AS (PARTITION BY dept_id 
ORDER BY name 
ROWS BETWEEN 2 PRECEDING AND 
CURRENT ROW)

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>sum</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>75000</td>
<td>75000</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>100000</td>
<td>NULL</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>60000</td>
<td>160000</td>
<td>NULL</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>220000</td>
<td>100000</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>190000</td>
<td>60000</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>210000</td>
<td>60000</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>65000</td>
<td>65000</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>130000</td>
<td>65000</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>300000</td>
<td>30000</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>370000</td>
<td>30000</td>
</tr>
</tbody>
</table>

Current row: Jon
FIRST_VALUE in frame is: Ed
LAST_VALUE “in frame”

SELECT name, dept_id AS dept, salary, SUM(salary) OVER w AS `sum`, FIRST_VALUE(salary) OVER w AS `first`, LAST_VALUE(salary) OVER w AS `last`
FROM employee
WINDOW w AS ( PARTITION BY dept_id ORDER BY name ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)

Current row: Jon
LAST_VALUE in frame is: Jon

<table>
<thead>
<tr>
<th>name</th>
<th>dept_id</th>
<th>salary</th>
<th>sum</th>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newt</td>
<td>NULL</td>
<td>75000</td>
<td>75000</td>
<td>75000</td>
<td>75000</td>
</tr>
<tr>
<td>Dag</td>
<td>10</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Ed</td>
<td>10</td>
<td>100000</td>
<td>100000</td>
<td>NULL</td>
<td>100000</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
<td>60000</td>
<td>160000</td>
<td>NULL</td>
<td>60000</td>
</tr>
<tr>
<td>Jon</td>
<td>10</td>
<td>60000</td>
<td>220000</td>
<td>100000</td>
<td>60000</td>
</tr>
<tr>
<td>Michael</td>
<td>10</td>
<td>70000</td>
<td>190000</td>
<td>60000</td>
<td>70000</td>
</tr>
<tr>
<td>Newt</td>
<td>10</td>
<td>80000</td>
<td>210000</td>
<td>60000</td>
<td>80000</td>
</tr>
<tr>
<td>Lebedev</td>
<td>20</td>
<td>65000</td>
<td>65000</td>
<td>65000</td>
<td>65000</td>
</tr>
<tr>
<td>Pete</td>
<td>20</td>
<td>65000</td>
<td>130000</td>
<td>65000</td>
<td>65000</td>
</tr>
<tr>
<td>Jeff</td>
<td>30</td>
<td>300000</td>
<td>300000</td>
<td>30000</td>
<td>300000</td>
</tr>
<tr>
<td>Will</td>
<td>30</td>
<td>70000</td>
<td>370000</td>
<td>30000</td>
<td>70000</td>
</tr>
</tbody>
</table>
NTH_VALUE "in frame"

SELECT name, dept_id AS dept, salary,
   SUM(salary) OVER w AS `sum`,
   NTH_VALUE(salary, 2) OVER w AS `nth`
FROM employee
WINDOW w AS (PARTITION BY dept_id
   ORDER BY name
   ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)

Current row: Jon
NTH_VALUE(...,2) in frame is: Fred
Logical flow

JOIN → GROUP BY → WINDOW 1 → WINDOW n → ORDER BY/DISTINCT/LIMIT

Row addressable buffer

in-mem: overflows to disk

Permits re-reading rows when frame moves