Window Functions in MySQL Notes V 1.0

RAJ SOLANKI

Contents

Window Functions in MySQL

Concepts

1. Window functions perform functions on a set of query rows and returns these rows
2. **Window** – This is the set of rows that are related to the current row where the function evaluation is taking place
3. Example of a window function

```
MySQL> SELECT
         year, country, product, profit,
         SUM(profit) OVER() AS total_profit,
         SUM(profit) OVER(PARTITION BY country) AS country_profit
       FROM sales
       ORDER BY country, year, product, profit;
```

year	country	product	profit	total_profit	country_profit
2000	Finland	Computer	1500	7535	1610
2000	Finland	Phone	100	7535	1610
2001	Finland	Phone	10	7535	1610
2000	India	Calculator	75	7535	1350
2000	India	Calculator	75	7535	1350
2000	India	Computer	1200	7535	1350
2000	USA	Calculator	75	7535	4575
2000	USA	Computer	1500	7535	4575
2001	USA	Calculator	50	7535	4575
2001	USA	Computer	1200	7535	4575
2001	USA	Computer	1500	7535	4575
2001	USA	TV	100	7535	4575
2001	USA	TV	150	7535	4575

4. **Syntax:**
    a. Window functions use an **over** clause to specify how to subset the query rows for evaluation
    b. Window functions are only permitted in the **Select** or **Order by** clauses
    c. Evaluation occurs after **Where, Group By, Having** have been processed.
    d. Over Clause
        i. There are 2 forms that this can be specified
            1. **OVER(window_spec) –** This appears between the parenthesis of the over clause.
                a. What can be specified for **window_spec?**

                i. **window_name** – You can use the named window defined elsewhere here or in combination of the following below.

                ii. **partition_clause**- Tells how to divide the query rows into groups specified by PARTITION BY expr [, expr]

iii. **order_clause –** specify ORDER BY expr [ASC|DESC] [, expr [ASC|DESC]] ... that will order within the partition

iv. **frame_clause -** Frames are subsets of the current partition.

2. **OVER window_name –** This is a named window specified elsewhere in the query.

5. What functions can be windowed?

   a. This can be broken down into 2 different classes of functions:

      i. **Aggregate functions** such as SUM (), AVG (), COUNT (), MAX (), MIN () etc. can be used as window functions
      ii. **Non-Aggregate functions** such as CUME_DIST (), DENSE_RANK (), LAG (), LEAD () etc. can only be used as window function. (Function descriptions are provided in Appendix A)

How to specify frames?

I. Frames are determined based on the "**current row**" so this allows the frame to move within the partition

II. **Real life example**

```
MySQL> SELECT
         time, subject, val,
         SUM(val) OVER (PARTITION BY subject ORDER BY time
                        ROWS UNBOUNDED PRECEDING)
           AS running_total,
         AVG(val) OVER (PARTITION BY subject ORDER BY time
                        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
           AS running_average
       FROM observations;
+----------+---------+------+--------------+-----------------+
| time     | subject | val  | running_total | running_average |
+----------+---------+------+--------------+-----------------+
| 07:00:00 | st113   |   10 |           10 |          9.5000 |
| 07:15:00 | st113   |    9 |           19 |         14.6667 |
| 07:30:00 | st113   |   25 |           44 |         18.0000 |
| 07:45:00 | st113   |   20 |           64 |         22.5000 |
| 07:00:00 | xh458   |    0 |            0 |          5.0000 |
| 07:15:00 | xh458   |   10 |           10 |          5.0000 |
| 07:30:00 | xh458   |    5 |           15 |         15.0000 |
| 07:45:00 | xh458   |   30 |           45 |         20.0000 |
| 08:00:00 | xh458   |   25 |           70 |         27.5000 |
+----------+---------+------+--------------+-----------------+
```

1. In the example above, we have a **running sum** which is adding up **all** preceding rows to the current value as indicated by the **unbounded.**
2. The **average** is being calculated using the numbers above & below the current value. The numbers in the 1st row and last row are using the available numbers. Hence 1st row average = 10+9=19/2=9.5 etc.

III. **What functions work with frames?**

1. First_Value (), Last_value (), Nth_value (), and aggregate functions such as sum () etc.

IV. **Frame Syntax**

```
frame_clause:
    frame_units frame_extent

frame_units:
    {ROWS | RANGE}

frame_extent:
    {frame_start | frame_between}

frame_between:
    BETWEEN frame_start AND frame_end

frame_start, frame_end: {
    CURRENT ROW
  | UNBOUNDED PRECEDING
  | UNBOUNDED FOLLOWING
  | expr PRECEDING
  | expr FOLLOWING
}
```

1. There are 2 components that need to be specified

    1. **frame_units** – Here you specify ***ROW*** or ***RANGE*** where **ROW** specifies start and end position for the frame based on position where as **RANGE** specifies a range of rows based on the **value** offset.

    2. **frame_extent** – After specifying frame_units we specify the start and end points for the frame IE *frame_start* and *frame_end*

        1. **CURRENT ROW** – This means that if **ROW** is specified as the unit then the bound is the current row and for **RANGE** it is the peer of the current row
        2. **UNBOUNDED PRECEDING** – This means it reaches all the way back to the 1st row
        3. **UNBOUNDED FOLLOWING** – This means it reaches all the way to the last row
        4. **Expression (expr) PRECEDING/expr Following** - If **ROW** is specified the bound is expr rows before/after the current row (# of rows determined by expr). If **RANGE** is specified, then the frame is are the rows with values in the expression range. (Note: If current row value is NULL then **RANGE** expr will consider the peer rows or rows that still in the **RANGE** of current row.
            a. Expressions can be nonnegative number, time interval in form of *INTERVAL val unit* (IE INTERVAL 5 DAY PRECEDING)

v. **Named Windows**

1. We can give windows names as shown in the example below

```
1.  SELECT
2.   DISTINCT year, country,
3.   FIRST_VALUE(year) OVER (w ORDER BY year ASC) AS first,
4.   FIRST_VALUE(year) OVER (w ORDER BY year DESC) AS last
5.  FROM sales
6.  WINDOW w AS (PARTITION BY country);
```

Window Functions Notes (Site 2)

Link: https://mysqlserverteam.com/mysql-8-0-2-introducing-window-functions/

1. **Order By**
   a. When an **order by** is used, MySQL will default to a different windows equivalent to the one below
      i. (**PARTITION** by employee **ORDER BY** date **RANGE BETWEEN UNBOUNDED PRECEDING** AND **CURRENT ROW**) /* AND PEERS */
      ii. In this case, the partition window includes all the rows from the 1st row to current row also known as an **expanding windows frame**
      iii. **Peers –** A **peer** is any row that sorts the same according to what is given in the order by.
         1. **Example –** If we sort by date then any rows with the same date are considered **peers**
   b. Order by example
      i. **SELECT** employee, sale, date, SUM (sale) **OVER (ORDER BY** date **ROWS UNBOUNDED PRECEDING**) AS sum_sales FROM sales;
   c. Omitting **Order By?**
      i. By omitting the **order by,** MySQL treats all rows within the partition as peers

2. **MA example**
   a. **MA** stands for moving average which we can use window functions for
   b. Below is the data we want to use and create a MA out of
      i. **Columns**:
         1. Name
         2. Date
         3. Sale
      ii. **INSERT INTO** sales **VALUES** ('odin', '2017-03-01', 200),
         a. ('odin', '2017-04-01', 300),
         b. ('odin', '2017-05-01', 400),
         c. ('odin', '2017-06-01', 200),
         d. ('odin', '2017-07-01', 600),
         e. ('odin', '2017-08-01', 100),
         f. ('thor', '2017-03-01', 400),
         g. ('thor', '2017-04-01', 300),

h. ('thor', '2017-05-01', 500),
i. ('thor', '2017-06-01', 400),
j. ('thor', '2017-07-01', 600),
k. ('thor', '2017-08-01', 150);

c. How we do we create a monthly **MA**?
   i. **SELECT MONTH** (date) as month, Avg(sum(sales)) **OVER** (**ORDER BY** month **RANGE BETWEEN 1 PRECEDING** and **1 FOLLOWING**) as sliding_avg from sales group by month;

Window Functions Notes (Site 3)

Link: http://www.mysqltutorial.org/mysql-window-functions/

1. **Windows Function Syntax**

```
1. window_function_name(expression)
2.     OVER (
3.         [partition_defintion]
4.         [order_definition]
5.         [frame_definition]
6.     )
```

2. How do interpret the above syntax?
   1. To use a function, we must specify 2 things
      I. ***window_function_name***
      II. **OVER clause**
         1. [partition_definition]
         2. [order_definition]
         3. [frame_definition]

   2. **Over clause options**
      I. ***partition_definition*** – breaks up the rows
         1. ***Syntax:*** *PARTITION BY <expression> [{,<expression>...}]*
      II. ***order_definition –*** *specifies how rows are ordered within partition*
         1. ***Syntax:*** *ORDER BY <expression> [ASC|DESC], [{,<expression>...}]*
      III. ***Frame_definition –*** It is a subset of the current partition with respect to the current position
         1. ***Syntax:*** *frame_unit {<frame_between>|<frame_start>}*
         2. ***frame_unit:*** *Defines the relationship between the current row and the frame. There are two units it can take*
            i. ***Row –*** *This forms the frame based on row position*

ii. ***Range –*** *This forms the frame based on row values*

IV. ***frame_start*** *– This can take 1 of the following values*

1. ***UNBOUNDED PRECEDING****: frame starts at the first row of the partition.*
2. ***N PRECEDING****: a physical N of rows before the first current row. N can be a literal number or an expression that evaluates to a number.*
3. ***CURRENT ROW****: the row of the current calculation*

V. ***frame_between*** – select between two row boundaries as shown below

1. **BETWEEN frame_boundary_1** AND **frame_boundary_2**
   i. **Frame_boundary** uses the following:
      1. **frame_start:** as mentioned previously.
      2. **UNBOUNDED FOLLOWING:** the frame ends at the final row in the partition.
      3. **N FOLLOWING:** a physical **N** of rows after the current row

VI. If you don't specify the **frame_definition** in the over clause, then MySQL uses the following frame by default:

1. **RANGE BETWEEN UNBOUNDED PRECEDING** AND **CURRENT ROW**

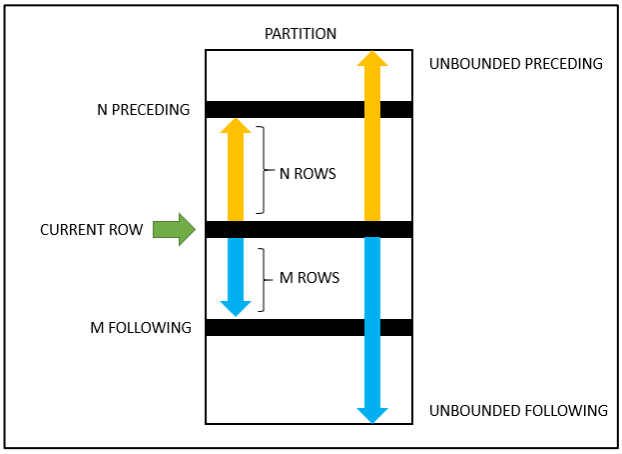


Figure 1 – Above figure represents what partitions and different **frame_definitions** look like.

Appendix A – Window Function List

1. **CUME_DIST ()** – It represents the # of rows with values less then or equal to row's value divided by total number of rows and always between 0 and 1.

```
SELECT
name,
  score,
  ROW_NUMBER() OVER (ORDER BY score) row_num,
  CUME_DIST() OVER (ORDER BY score) cume_dist
_val
FROM
scores;
```

name	score	row_num	cume_dist_val
Jones	55	1	0.2
Williams	55	2	0.2
Brown	62	3	0.4
Taylor	62	4	0.4
Thomas	72	5	0.6
Wilson	72	6	0.6
Smith	81	7	0.7
Davies	84	8	0.8
Evans	87	9	0.9
Johnson	100	10	1

2. **DENSE_RANK ()** – assigns a rank to every row within its partition based on the **ORDER BY** clause. Same rank is assigned to those with equal values.

```
DENSE_RANK () OVER (
    PARTITION BY <expressi
on> [{, <expression>...}]
    ORDER BY <expression>
[ASC|DESC], [{, <expressio
n>...}]
)
```

```
SELECT
    val,
    DENSE_RANK () OVER
(
        ORDER BY val
    ) my_rank
FROM
    t;
```

val	my_rank
1	1
2	2
2	2
3	3
4	4
4	4
5	5

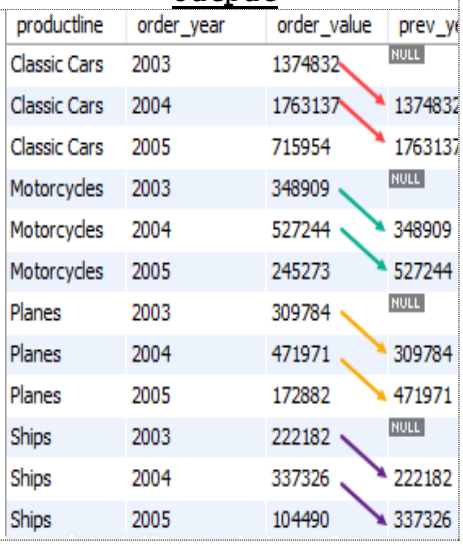
3. **FIRST_VALUE ()** – selects the 1st row from partition, frame, or result set

```
FIRST_VALUE (expre
ssion) OVER (
[partition_clause]
[order_clause]
[frame_clause]
)
```

```
 SELECT
    employee_name,
    department,
    hours,
    FIRST_VALUE(employe
e_name) OVER (
        PARTITION BY de
partment
        ORDER BY hours
    ) least_over_time
FROM
    overtime
```

employee_name	department	hours	least_over_time
Diane Murphy	Accounting	37	Diane Murphy
Jeff Firrelli	Accounting	40	Diane Murphy
Mary Patterson	Accounting	74	Diane Murphy
Gerard Bondur	Finance	47	Gerard Bondur
William Patterson	Finance	58	Gerard Bondur
Anthony Bow	Finance	66	Gerard Bondur
Leslie Thompson	IT	88	Leslie Thompson
Leslie Jennings	IT	90	Leslie Thompson

4. **LAG ()** – Allows you to access data from a selected number of rows back.

Syntax	Example	Output
LAG(<expression> [, offset [, default_value]]) OVER (PARTITION BY expr, ... ORDER BY expr [ASC|DESC], ...)	SELECT productline, order_year, order_value, **LAG (order_value, 1) OVER** (**PARTITION BY** productLine **ORDER BY** order_year) prev_year_order_value FROM productline_sales;	

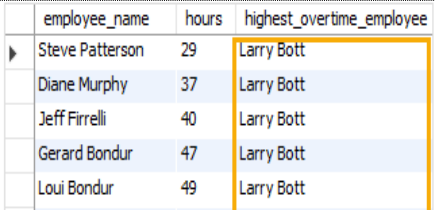
Syntax:

1. **Expression** – **Lag ()** returns the value of what's specified in the expression **offset** rows back within in the partition or result set

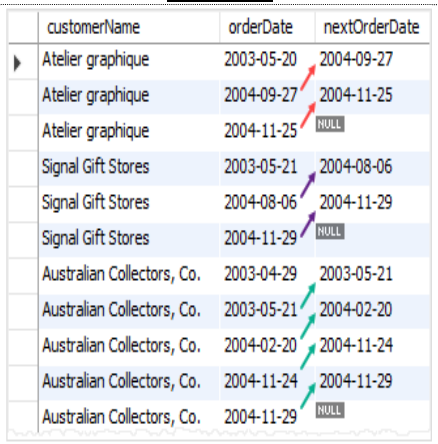
2. **Offset** – This is the number of rows back **Lag ()** will go. If 0 is specified, **Lag ()** will use the current row. If nothing is specified, **Lag ()** uses 1 by default.

3. **default value** – **Lag ()** will return the default value if there is no preceding row. If default value is not specified, then NULL will be returned.

5. **LAST_VALUE ()** – Selects the last value of an ordered set of rows within a partition.

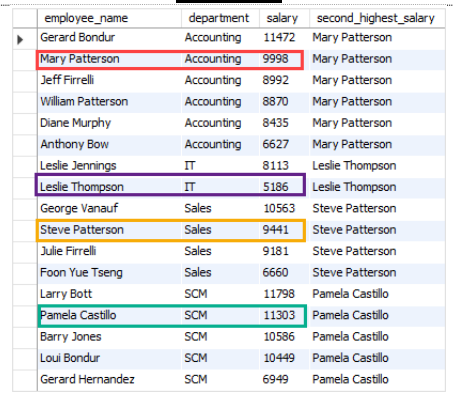
Syntax	Example	Output
LAST_VALUE (expression) OVER ([partition_clause] [order_clause] [frame_clause])	SELECT employee_name, hours, **LAST_VALUE**(employee_name) **OVER** (ORDER BY hours RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) highest_overtime_employee FROM overtime;	

6. **LEAD ()** – Selects the subsequent rows within a partition or result set

Syntax	Example	Output
LEAD(<expression> [, offset [, default_value]]) OVER (PARTITION BY (expr) ORDER BY (expr))	SELECT customerName, orderDate, LEAD (orderDate,1) OVER (PARTITION BY customerNumber ORDER BY orderDate) nextOrderDate FROM orders INNER JOIN customers USING (customerNumber);	

customerName	orderDate	nextOrderDate
Atelier graphique	2003-05-20	2004-09-27
Atelier graphique	2004-09-27	2004-11-25
Atelier graphique	2004-11-25	NULL
Signal Gift Stores	2003-05-21	2004-08-06
Signal Gift Stores	2004-08-06	2004-11-29
Signal Gift Stores	2004-11-29	NULL
Australian Collectors, Co.	2003-04-29	2003-05-21
Australian Collectors, Co.	2003-05-21	2004-02-20
Australian Collectors, Co.	2004-02-20	2004-11-24
Australian Collectors, Co.	2004-11-24	2004-11-29
Australian Collectors, Co.	2004-11-29	NULL

7. **NTH_VALUE ()** – Allows you to grab the nth row in a result set.

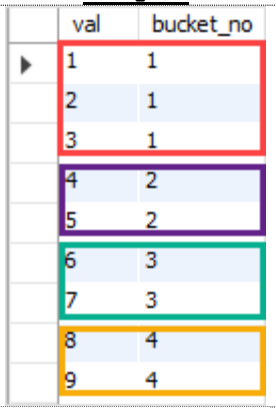
Syntax	Example	Output
NTH_VALUE (expression, N) FROM FIRST OVER (partition_clause order_clause frame_clause)	SELECT employee_name, department, salary, **NTH_VALUE** (employee_name, 2) OVER (PARTITION BY department ORDER BY salary DESC RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) second_highest_salary FROM basic_pays;	

employee_name	department	salary	second_highest_salary
Gerard Bondur	Accounting	11472	Mary Patterson
Mary Patterson	Accounting	9998	Mary Patterson
Jeff Firrelli	Accounting	8992	Mary Patterson
William Patterson	Accounting	8870	Mary Patterson
Diane Murphy	Accounting	8435	Mary Patterson
Anthony Bow	Accounting	6627	Mary Patterson
Leslie Jennings	IT	8113	Leslie Thompson
Leslie Thompson	IT	5186	Leslie Thompson
George Vanauf	Sales	10563	Steve Patterson
Steve Patterson	Sales	9441	Steve Patterson
Julie Firrelli	Sales	9181	Steve Patterson
Foon Yue Tseng	Sales	6660	Steve Patterson
Larry Bott	SCM	11798	Pamela Castillo
Pamela Castillo	SCM	11303	Pamela Castillo
Barry Jones	SCM	10586	Pamela Castillo
Loui Bondur	SCM	10449	Pamela Castillo
Gerard Hernandez	SCM	6949	Pamela Castillo

Comments

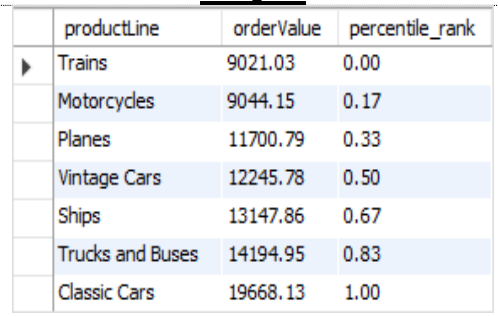
1. **Nth_VALUE()** returns NULL if the Nth row does not exist
2. **Nth_VALUE ()** grabs value from the beginning

8. **NTILE ()** – divides the rows into a specified number of groups

Syntax	Example	Output
NTILE(n) OVER (PARTITION BY <expression> [{, <expression>...}] ORDER BY <expression> [ASC|DESC], [{, <expression>...}])	SELECT val, **NTILE (4)** OVER (ORDER BY val) group FROM t;	

	val	bucket_no
▶	1	1
	2	1
	3	1
	4	2
	5	2
	6	3
	7	3
	8	4
	9	4

9. **PERCENT_RANK ()** – calculates the **%** rank of a value within a partition

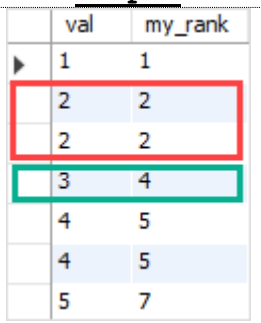
Syntax	Example	Output
PERCENT_RANK () OVER (PARTITION BY expr, ... ORDER BY expr [ASC|DESC], ...)	SELECT productLine, orderValue, ROUND(**PERCENT_RANK ()** OVER (ORDER BY orderValue) ,2) percentile_rank FROM t;	

	productLine	orderValue	percentile_rank
▶	Trains	9021.03	0.00
	Motorcycles	9044.15	0.17
	Planes	11700.79	0.33
	Vintage Cars	12245.78	0.50
	Ships	13147.86	0.67
	Trucks and Buses	14194.95	0.83
	Classic Cars	19668.13	1.00

Comments

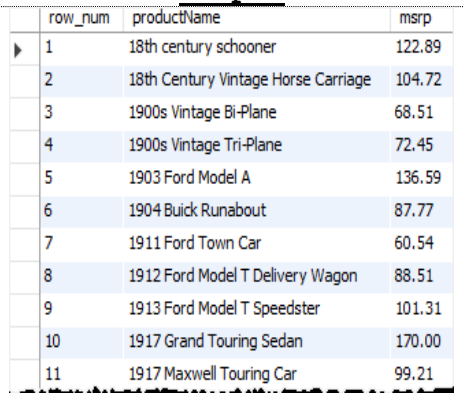
1. The rank is calculated using (rank-1)/(total_rows-1)
2. **PERCENT_RANK ()** will always return 0 for the 1st row in a partition

10. **RANK ()** – assigns a rank to each row within the partition

Syntax	Example	Output
RANK () OVER (PARTITION BY <expression> [{, <expression>...}] ORDER BY <expression> [ASC|DESC], [{, <expression>...}])	SELECT val, **RANK()** OVER (ORDER BY val) my_rank FROM t;	

	val	my_rank
▶	1	1
	2	2
	2	2
	3	4
	4	5
	4	5
	5	7

11. **ROW_NUMBER ()** – returns the row number within a partition

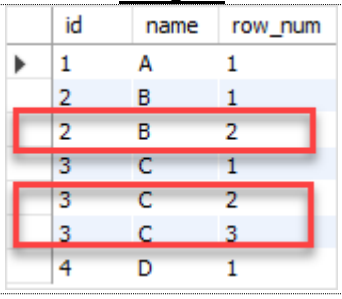
Syntax	Example	Output
ROW_NUMBER () OVER (<partition_definition> <order_definition>)	SELECT **ROW_NUMBER ()** OVER (ORDER BY productName) row_num, productName, msrp FROM products ORDER BY productName;	

	row_num	productName	msrp
▶	1	18th century schooner	122.89
	2	18th Century Vintage Horse Carriage	104.72
	3	1900s Vintage Bi-Plane	68.51
	4	1900s Vintage Tri-Plane	72.45
	5	1903 Ford Model A	136.59
	6	1904 Buick Runabout	87.77
	7	1911 Ford Town Car	60.54
	8	1912 Ford Model T Delivery Wagon	88.51
	9	1913 Ford Model T Speedster	101.31
	10	1917 Grand Touring Sedan	170.00
	11	1917 Maxwell Touring Car	99.21

Comments

1. You can use **ROW_NUMBER ()** to find the top/bottom N of every group by filtering on the row number in the where clause and using ORDER BY within the **OVER** clause.

2. Another use of **ROW_NUMBER ()** is removing duplicate rows based on some ID column example below.

Example	Output
SELECT id, name, **ROW_NUMBER()** OVER (PARTITION BY id, name ORDER BY id) AS row_num FROM t;	

	id	name	row_num
▶	1	A	1
	2	B	1
	2	B	2
	3	C	1
	3	C	2
	3	C	3
	4	D	1