12.21.1 Window Function Descriptions

This section describes nonaggregate window functions that, for each row from a query, perform a calculation using rows related to that row. Most aggregate functions also can be used as window functions; see Section 12.20.1, “Aggregate Function Descriptions”.

For window function usage information and examples, and definitions of terms such as the `OVER` clause, window, partition, frame, and peer, see Section 12.21.2, “Window Function Concepts and Syntax”.

Table 12.26 Window Functions

Name	Description
`CUME_DIST()`	Cumulative distribution value
`DENSE_RANK()`	Rank of current row within its partition, without gaps
`FIRST_VALUE()`	Value of argument from first row of window frame
`LAG()`	Value of argument from row lagging current row within partition
`LAST_VALUE()`	Value of argument from last row of window frame
`LEAD()`	Value of argument from row leading current row within partition
`NTH_VALUE()`	Value of argument from N-th row of window frame
`NTILE()`	Bucket number of current row within its partition.
`PERCENT_RANK()`	Percentage rank value
`RANK()`	Rank of current row within its partition, with gaps
`ROW_NUMBER()`	Number of current row within its partition

In the following function descriptions, ***`over_clause`*** represents the `OVER` clause, described in Section 12.21.2, “Window Function Concepts and Syntax”. Some window functions permit a ***`null_treatment`*** clause that specifies how to handle `NULL` values when calculating results. This clause is optional. It is part of the SQL standard, but the MySQL implementation permits only `RESPECT NULLS` (which is also the default). This means that `NULL` values are considered when calculating results. `IGNORE NULLS` is parsed, but produces an error.

- `CUME_DIST()` ***`over_clause`***

Returns the cumulative distribution of a value within a group of values; that is, the percentage of partition values less than or equal to the value in the current row. This represents the number of rows preceding or peer with the current row in the window ordering of the window partition divided by the total number of rows in the window partition. Return values range from 0 to 1.

This function should be used with `ORDER BY` to sort partition rows into the desired order. Without `ORDER BY`, all rows are peers and have value ***N/N*** = 1, where ***N*** is the partition size.

over_clause is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

The following query shows, for the set of values in the `val` column, the `CUME_DIST()` value for each row, as well as the percentage rank value returned by the similar `PERCENT_RANK()` function. For reference, the query also displays row numbers using `ROW_NUMBER()`:

```
mysql> SELECT
         val,
         ROW_NUMBER()   OVER w AS 'row_number',
         CUME_DIST()    OVER w AS 'cume_dist',
         PERCENT_RANK() OVER w AS 'percent_rank'
       FROM numbers
       WINDOW w AS (ORDER BY val);
+------+------------+--------------------+--------------+
| val  | row_number | cume_dist          | percent_rank |
+------+------------+--------------------+--------------+
|    1 |          1 | 0.2222222222222222 |            0 |
|    1 |          2 | 0.2222222222222222 |            0 |
|    2 |          3 | 0.3333333333333333 |         0.25 |
|    3 |          4 | 0.6666666666666666 |        0.375 |
|    3 |          5 | 0.6666666666666666 |        0.375 |
|    3 |          6 | 0.6666666666666666 |        0.375 |
|    4 |          7 | 0.8888888888888888 |         0.75 |
|    4 |          8 | 0.8888888888888888 |         0.75 |
|    5 |          9 |                  1 |            1 |
+------+------------+--------------------+--------------+
```

- `DENSE_RANK()` ***over_clause***

  Returns the rank of the current row within its partition, without gaps. Peers are considered ties and receive the same rank. This function assigns consecutive ranks to peer groups; the result is that groups of size greater than one do not produce noncontiguous rank numbers. For an example, see the `RANK()` function description.

This function should be used with `ORDER BY` to sort partition rows into the desired order. Without `ORDER BY`, all rows are peers.

over_clause is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

- `FIRST_VALUE(`***`expr`***`)` [***null_treatment***] ***over_clause***

  Returns the value of ***expr*** from the first row of the window frame.

  over_clause is as described in Section 12.21.2, “Window Function Concepts and Syntax”. ***null_treatment*** is as described in the section introduction.

  The following query demonstrates `FIRST_VALUE()`, `LAST_VALUE()`, and two instances of `NTH_VALUE()`:

```
mysql> SELECT
         time, subject, val,
         FIRST_VALUE(val)  OVER w AS 'first',
         LAST_VALUE(val)   OVER w AS 'last',
         NTH_VALUE(val, 2) OVER w AS 'second',
         NTH_VALUE(val, 4) OVER w AS 'fourth'
       FROM observations
       WINDOW w AS (PARTITION BY subject ORDER BY time
                    ROWS UNBOUNDED PRECEDING);
+----------+---------+------+-------+------+--------+--------+
| time     | subject | val  | first | last | second | fourth |
+----------+---------+------+-------+------+--------+--------+
| 07:00:00 | st113   |   10 |    10 |   10 |   NULL |   NULL |
| 07:15:00 | st113   |    9 |    10 |    9 |      9 |   NULL |
| 07:30:00 | st113   |   25 |    10 |   25 |      9 |   NULL |
| 07:45:00 | st113   |   20 |    10 |   20 |      9 |     20 |
| 07:00:00 | xh458   |    0 |     0 |    0 |   NULL |   NULL |
| 07:15:00 | xh458   |   10 |     0 |   10 |     10 |   NULL |
| 07:30:00 | xh458   |    5 |     0 |    5 |     10 |   NULL |
| 07:45:00 | xh458   |   30 |     0 |   30 |     10 |     30 |
| 08:00:00 | xh458   |   25 |     0 |   25 |     10 |     30 |
+----------+---------+------+-------+------+--------+--------+
```

  Each function uses the rows in the current frame, which, per the window definition shown, extends from the first partition row to the current row. For the `NTH_VALUE()` calls, the current frame does not always include the requested row; in such cases, the return value is `NULL`.

- `LAG(`***`expr`***` [, `***`N`***`[, `***`default`***`]])` [***null_treatment***] ***over_clause***

Returns the value of ***expr*** from the row that lags (precedes) the current row by ***N*** rows within its partition. If there is no such row, the return value is ***default***. For example, if ***N*** is 3, the return value is ***default*** for the first two rows. If ***N*** or ***default*** are missing, the defaults are 1 and `NULL`, respectively.

N must be a literal nonnegative integer. If ***N*** is 0, ***expr*** is evaluated for the current row.

Beginning with MySQL 8.0.22, ***N*** cannot be `NULL`. In addition, it must now be an integer in the range `1` to `2`^{63}, inclusive, in any of the following forms:

- an unsigned integer constant literal
- a positional parameter marker (`?`)
- a user-defined variable
- a local variable in a stored routine

over_clause is as described in Section 12.21.2, “Window Function Concepts and Syntax”.
null_treatment is as described in the section introduction.

`LAG()` (and the similar `LEAD()` function) are often used to compute differences between rows. The following query shows a set of time-ordered observations and, for each one, the `LAG()` and `LEAD()` values from the adjoining rows, as well as the differences between the current and adjoining rows:

```
mysql> SELECT
         t, val,
         LAG(val)        OVER w AS 'lag',
         LEAD(val)       OVER w AS 'lead',
         val - LAG(val)  OVER w AS 'lag diff',
         val - LEAD(val) OVER w AS 'lead diff'
       FROM series
       WINDOW w AS (ORDER BY t);
+----------+------+------+------+----------+-----------+
| t        | val  | lag  | lead | lag diff | lead diff |
+----------+------+------+------+----------+-----------+
| 12:00:00 |  100 | NULL |  125 |     NULL |       -25 |
| 13:00:00 |  125 |  100 |  132 |       25 |        -7 |
| 14:00:00 |  132 |  125 |  145 |        7 |       -13 |
| 15:00:00 |  145 |  132 |  140 |       13 |         5 |
| 16:00:00 |  140 |  145 |  150 |       -5 |       -10 |
| 17:00:00 |  150 |  140 |  200 |       10 |       -50 |
| 18:00:00 |  200 |  150 | NULL |       50 |      NULL |
+----------+------+------+------+----------+-----------+
```

In the example, the `LAG()` and `LEAD()` calls use the default ***N*** and ***default*** values of 1 and `NULL`, respectively.

The first row shows what happens when there is no previous row for `LAG()`: The function returns the ***default*** value (in this case, `NULL`). The last row shows the same thing when there is no next row for `LEAD()`.

`LAG()` and `LEAD()` also serve to compute sums rather than differences. Consider this data set, which contains the first few numbers of the Fibonacci series:

```
mysql> SELECT n FROM fib ORDER BY n;
+------+
| n    |
+------+
|    1 |
|    1 |
|    2 |
|    3 |
|    5 |
|    8 |
+------+
```

The following query shows the `LAG()` and `LEAD()` values for the rows adjacent to the current row. It also uses those functions to add to the current row value the values from the preceding and following rows. The effect is to generate the next number in the Fibonacci series, and the next number after that:

```
mysql> SELECT
         n,
         LAG(n, 1, 0)      OVER w AS 'lag',
         LEAD(n, 1, 0)     OVER w AS 'lead',
         n + LAG(n, 1, 0)  OVER w AS 'next_n',
         n + LEAD(n, 1, 0) OVER w AS 'next_next_n'
       FROM fib
       WINDOW w AS (ORDER BY n);
+------+------+------+--------+-------------+
| n    | lag  | lead | next_n | next_next_n |
+------+------+------+--------+-------------+
|    1 |    0 |    1 |      1 |           2 |
|    1 |    1 |    2 |      2 |           3 |
|    2 |    1 |    3 |      3 |           5 |
|    3 |    2 |    5 |      5 |           8 |
|    5 |    3 |    8 |      8 |          13 |
|    8 |    5 |    0 |     13 |           8 |
+------+------+------+--------+-------------+
```

One way to generate the initial set of Fibonacci numbers is to use a recursive common table expression. For an example, see Fibonacci Series Generation.

Beginning with MySQL 8.0.22, you cannot use a negative value for the rows argument of this function.

- `LAST_VALUE(`***`expr`***`)` [***`null_treatment`***] ***`over_clause`***

  Returns the value of ***`expr`*** from the last row of the window frame.

  `over_clause` is as described in Section 12.21.2, "Window Function Concepts and Syntax". ***`null_treatment`*** is as described in the section introduction.

  For an example, see the `FIRST_VALUE()` function description.

- `LEAD(`***`expr`***` [, `***`N`***`[, `***`default`***`]])` [***`null_treatment`***] ***`over_clause`***

  Returns the value of ***`expr`*** from the row that leads (follows) the current row by ***`N`*** rows within its partition. If there is no such row, the return value is ***`default`***. For example, if ***`N`*** is 3, the return value is ***`default`*** for the last two rows. If ***`N`*** or ***`default`*** are missing, the defaults are 1 and `NULL`, respectively.

  `N` must be a literal nonnegative integer. If ***`N`*** is 0, ***`expr`*** is evaluated for the current row.

  Beginning with MySQL 8.0.22, ***`N`*** cannot be `NULL`. In addition, it must now be an integer in the range `1` to `2`^{63}, inclusive, in any of the following forms:

  - an unsigned integer constant literal
  - a positional parameter marker (`?`)
  - a user-defined variable
  - a local variable in a stored routine

  `over_clause` is as described in Section 12.21.2, "Window Function Concepts and Syntax". ***`null_treatment`*** is as described in the section introduction.

  For an example, see the `LAG()` function description.

  In MySQL 8.0.22 and later, use of a negative value for the rows argument of this function is not permitted.

- `NTH_VALUE(`***`expr`***`, `***`N`***`)` [***`from_first_last`***] [***`null_treatment`***] ***`over_clause`***

  Returns the value of ***`expr`*** from the ***`N`***-th row of the window frame. If there is no such row, the return value is `NULL`.

  `N` must be a literal positive integer.

  `from_first_last` is part of the SQL standard, but the MySQL implementation permits only `FROM FIRST` (which is also the default). This means that calculations begin at the first row of the window. `FROM LAST` is parsed, but produces an error. To obtain the same effect as `FROM LAST` (begin calculations at the last row of the window), use `ORDER BY` to sort in reverse order.

  `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”. ***`null_treatment`*** is as described in the section introduction.

  For an example, see the `FIRST_VALUE()` function description.

  In MySQL 8.0.22 and later, you cannot use `NULL` for the row argument of this function.

- `NTILE(`***`N`***`)` ***`over_clause`***

  Divides a partition into ***`N`*** groups (buckets), assigns each row in the partition its bucket number, and returns the bucket number of the current row within its partition. For example, if ***`N`*** is 4, `NTILE()` divides rows into four buckets. If ***`N`*** is 100, `NTILE()` divides rows into 100 buckets.

  `N` must be a literal positive integer. Bucket number return values range from 1 to ***`N`***.

  Beginning with MySQL 8.0.22, ***`N`*** cannot be `NULL`. In addition, it must be an integer in the range `1` to 2^{63}, inclusive, in any of the following forms:

  - an unsigned integer constant literal
  - a positional parameter marker (`?`)
  - a user-defined variable
  - a local variable in a stored routine

  This function should be used with `ORDER BY` to sort partition rows into the desired order.

  `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

The following query shows, for the set of values in the `val` column, the percentile values resulting from dividing the rows into two or four groups. For reference, the query also displays row numbers using `ROW_NUMBER()`:

```
mysql> SELECT
         val,
         ROW_NUMBER() OVER w AS 'row_number',
         NTILE(2)     OVER w AS 'ntile2',
         NTILE(4)     OVER w AS 'ntile4'
       FROM numbers
       WINDOW w AS (ORDER BY val);
+------+------------+--------+--------+
| val  | row_number | ntile2 | ntile4 |
+------+------------+--------+--------+
|    1 |          1 |      1 |      1 |
|    1 |          2 |      1 |      1 |
|    2 |          3 |      1 |      1 |
|    3 |          4 |      1 |      2 |
|    3 |          5 |      1 |      2 |
|    3 |          6 |      2 |      3 |
|    4 |          7 |      2 |      3 |
|    4 |          8 |      2 |      4 |
|    5 |          9 |      2 |      4 |
+------+------------+--------+--------+
```

Beginning with MySQL 8.0.22, the construct `NTILE(NULL)` is no longer permitted.

- `PERCENT_RANK()` ***`over_clause`***

  Returns the percentage of partition values less than the value in the current row, excluding the highest value. Return values range from 0 to 1 and represent the row relative rank, calculated as the result of this formula, where ***`rank`*** is the row rank and ***`rows`*** is the number of partition rows:

  ```
  (rank - 1) / (rows - 1)
  ```

  This function should be used with `ORDER BY` to sort partition rows into the desired order. Without `ORDER BY`, all rows are peers.

  `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

  For an example, see the `CUME_DIST()` function description.

- `RANK()` ***`over_clause`***

  Returns the rank of the current row within its partition, with gaps. Peers are considered ties and receive the same rank. This function does not assign consecutive ranks to peer groups if groups of size greater than one exist; the result is noncontiguous rank numbers.

  This function should be used with `ORDER BY` to sort partition rows into the desired order. Without `ORDER BY`, all rows are peers.

  `over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”.

  The following query shows the difference between `RANK()`, which produces ranks with gaps, and `DENSE_RANK()`, which produces ranks without gaps. The query shows rank values for each member of a set of values in the `val` column, which contains some duplicates. `RANK()` assigns peers (the duplicates) the same rank value, and the next greater value has a rank higher by the number of peers minus one. `DENSE_RANK()` also assigns peers the same rank value, but the next higher value has a rank one greater. For reference, the query also displays row numbers using `ROW_NUMBER()`:

```
mysql> SELECT
         val,
         ROW_NUMBER() OVER w AS 'row_number',
         RANK()       OVER w AS 'rank',
         DENSE_RANK() OVER w AS 'dense_rank'
       FROM numbers
       WINDOW w AS (ORDER BY val);
+------+------------+------+------------+
| val  | row_number | rank | dense_rank |
+------+------------+------+------------+
|    1 |          1 |    1 |          1 |
|    1 |          2 |    1 |          1 |
|    2 |          3 |    3 |          2 |
|    3 |          4 |    4 |          3 |
|    3 |          5 |    4 |          3 |
|    3 |          6 |    4 |          3 |
|    4 |          7 |    7 |          4 |
|    4 |          8 |    7 |          4 |
|    5 |          9 |    9 |          5 |
+------+------------+------+------------+
```

- `ROW_NUMBER()` ***`over_clause`***

  Returns the number of the current row within its partition. Rows numbers range from 1 to the number of partition rows.

`ORDER BY` affects the order in which rows are numbered. Without `ORDER BY`, row numbering is nondeterministic.

`ROW_NUMBER()` assigns peers different row numbers. To assign peers the same value, use `RANK()` or `DENSE_RANK()`. For an example, see the `RANK()` function description.

`over_clause` is as described in Section 12.21.2, “Window Function Concepts and Syntax”.