# Window functions

Seminar 5

## Analytical (window) functions

Analytical (window) functions:

- Takes as an argument a column of the intermediate result of the calculation and return the same column.
- The place of their use can only be the ORDER BY and SELECT sections that perform the final processing of the logical intermediate result.
- Acts like aggregate functions, but do not reduce the level of detail.
- Aggregate data in portions, the number and size of which is regulated by a special syntactic construction.

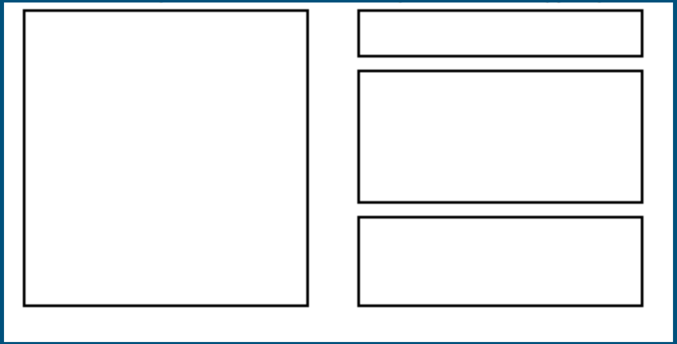
## Syntax

```
function_name(expression) OVER (
    [ <PARTITION BY clause> ]       -- окно
    [ <ORDER BY clause> ]           -- сортировка
    [ <ROWS or RANGE clause> ]      -- границы окна
) AS attr_name
```

In a normal query, the entire set of rows is processed as a single "whole piece", for which aggregates are considered.

And when using window functions, the query is divided into parts (windows) and its aggregates are already considered for each of the individual parts

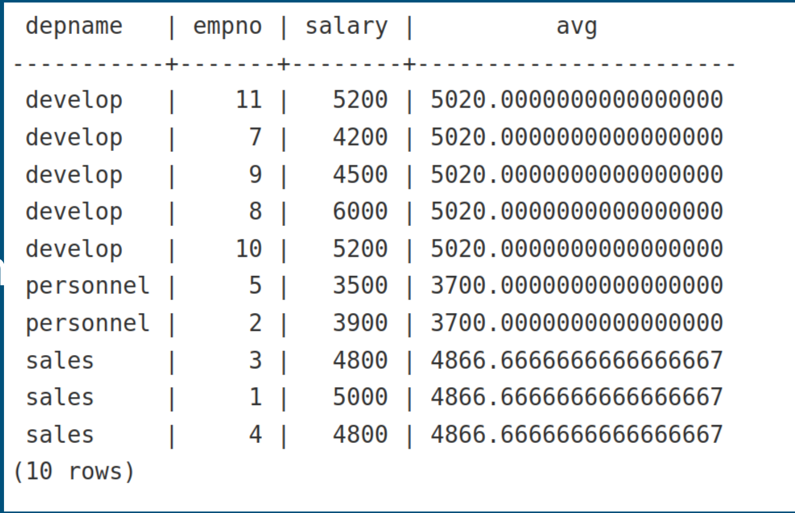
.



## Example

An example showing how to compare the salary of each employee with the average salary of his department:

```
SELECT
    depname,
    empno,
    salary,
    avg(salary) OVER (PARTITION BY depnan
FROM
    empsalary;
```

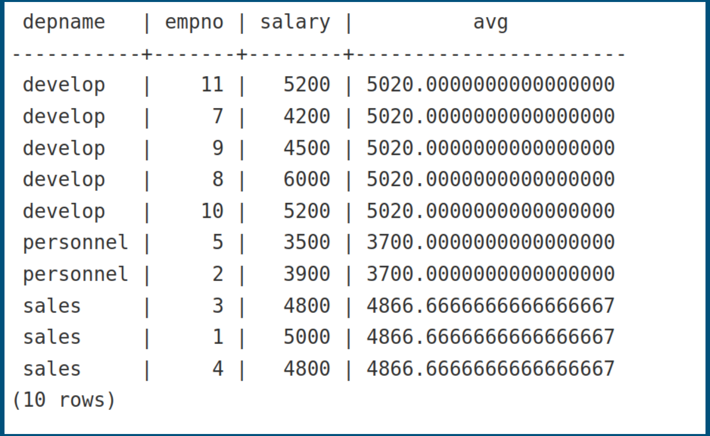


```
 depname   | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

## Example

The first three columns are extracted directly from the emp salary table, and there is a result row for each row of the table.

The fourth column turned out to be the average value calculated for all rows having the same depname value as the current row. (In fact, the average is calculated by the same ordinary, non-windowed avg function, but the OVER clause turns it into a windowed one, so that its action is limited to the window frames.)



```
 depname   | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

## OVER

- OVER defines the set of rows that the window function will use, including data sorting. (“window”)
- In the expression that specifies the window function, the OVER statement restricts the sets of rows with the same values in the field by which the division occurs.
- The OVER() statement itself is unlimited and contains all the rows from the result set.
- The OVER statement can be used multiple times in a single SELECT, each with its own division and sorting.

## Partitioning rules

Inside OVER, you must specify the table field on which the “window” will slide and the rule by which the rows will be partitioned:

- PARTITION BY: — responsible for partitioning criteria

Logically divides the set into groups according to criteria.

Analytical functions are applied to groups independently.

If you do not specify the partitioning construction, the entire set is considered one group.

| maker | price |
|---|---|
| Yamaha | 300 |
| Yamaha | 500 |
| Fender | 450 |
| Fender | 450 |

SELECT maker, price, **avg(price) OVER (PARTITION BY maker)** as avg FROM table;

| maker | price | avg |
|---|---|---|
| Yamaha | 300 | 400 |
| Yamaha | 500 | 400 |
| Fender | 450 | 450 |
| Fender | 450 | 450 |

## ORDER BY

- ORDER BY: — responsible for sorting

Sets the sorting criteria within each group.

Aggregate functions in the absence of the ORDER BY construct are calculated for all rows of the group, and the same value is given for each row, i.e. the function is used as a summary.

If an aggregate function is used with an ORDER BY construct, then it is calculated on the current row and all rows before it, i.e. the function is used as a windowed one. (the cumulative total is calculated)

| maker | price |
|---|---|
| Yamaha | 300 |
| Yamaha | 500 |
| Fender | 450 |
| Fender | 450 |

SELECT maker, price **avg(price) OVER (ORDER BY maker)** as avg FROM table;

| maker | price | avg |
|---|---|---|
| Yamaha | 300 | 400 |
| Yamaha | 500 | 400 |
| Fender | 450 | 425* |
| Fender | 450 | 425 |

## ROWS | RANGE

- ROWS | RANGE: — additional restrictions on the range of window rows (the presence of ORDER BY is required):

ROWS (by rows) — allows you to manually determine the boundaries of the window for which the value is calculated; can work with PRECEDING/FOLLOWING.

RANGE (based on the values from ORDER BY, a sub-window is formed)— close enough to the previous one, but still not the same (Alexander Faritovich's professional opinion: “I have no idea when it can be used”); But using ‘RANGE CURRENT ROW’ after ORDER BY allows you to get rid of the cumulative total; does not know how to work with PRECEDING/FOLLOWING.

By default, it considers from UNBOUNDED PRECEDING to CURRENT ROW. (UNBOUNDED PRECEDING/FOLLOWING — we consider up to the end/beginning of the window.)

We select the lines within the window, but if necessary, we can manually register the previous/subsequent lines so that it can go beyond the window.

| n |
|---|
| 1 |
| 1 |
| 2 |
| 3 |
| 4 |

```
SELECT n,
        sum(n) OVER (ORDER BY n
        ROWS BETWEEN CURRENT ROW
        AND 1 FOLLOWING*) AS cur_foll,
        sum(n) OVER (ORDER BY n RANGE
        CURRENT ROW**)
FROM table;
```

*The window is the current row and the next one.

**We count within the window with the same

by the value of p. (there is no cumulative total)

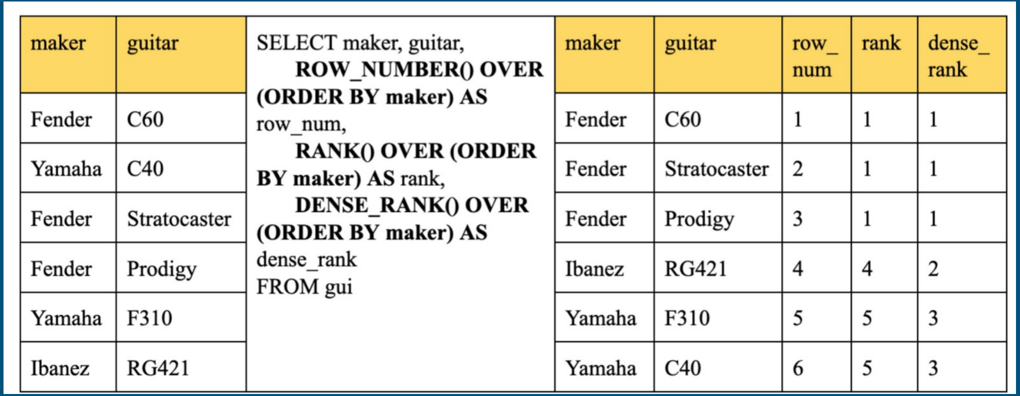
| n | cur_foll | cur_row |
|---|---|---|
| 1 | 2 | 2 |
| 1 | 3 | 2 |
| 2 | 5 | 2 |
| 3 | 7 | 3 |
| 4 | 4 | 4 |

## Classification of window functions

- Aggregating (sum, avg, min, max, count)
- Ranking (row_number, rank, dense_rank)
- Offsets (lag, lead, first_value, last_value); offset functions are used with field indication.

## Ranking functions

- row_number() – we number each row of the window sequentially with a step of 1.
- rank() – rank each row of the window with a gap in the numbering when the values are equal.
- dense_rank() – we rank each row of the window without breaks in the numbering when the values are equal.



| maker | guitar |
|---|---|
| Fender | C60 |
| Yamaha | C40 |
| Fender | Stratocaster |
| Fender | Prodigy |
| Yamaha | F310 |
| Ibanez | RG421 |

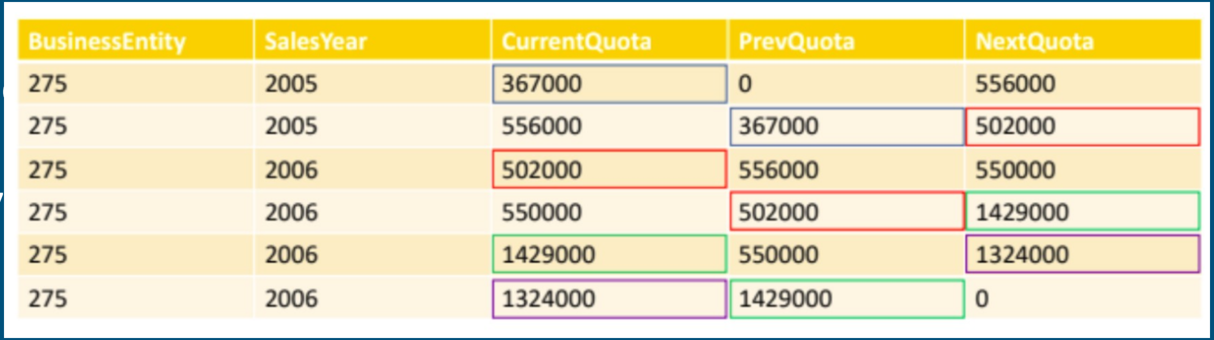
```
SELECT maker, guitar,
    ROW_NUMBER() OVER (ORDER BY maker) AS row_num,
    RANK() OVER (ORDER BY maker) AS rank,
    DENSE_RANK() OVER (ORDER BY maker) AS dense_rank
FROM gui
```

| maker | guitar | row_num | rank | dense_rank |
|---|---|---|---|---|
| Fender | C60 | 1 | 1 | 1 |
| Fender | Stratocaster | 2 | 1 | 1 |
| Fender | Prodigy | 3 | 1 | 1 |
| Ibanez | RG421 | 4 | 4 | 2 |
| Yamaha | F310 | 5 | 5 | 3 |
| Yamaha | C40 | 6 | 5 | 3 |

## Offset functions

- lag(attr, offset (offset), default_value(default value in case our string is the first)) – the previous value with a shift.
- lead(attr, offset, default_value) – the next value with a shift.
- first_value(attr) – the first value in the window from the first to the current line.
- last_value(attr) – the last value in the window from the first to the current line.

```
SELECT
    BusinessEntity,
    SalesYear,
    CurrentQuota,
    LAG(CurrentQuota, 1, 0) OVER (ORDER BY SalesYear) AS PrevQuota,
    LEAD(CurrentQuota, 1, 0) OVER (ORDER BY SalesYear) AS NextQuota
FROM
    SalesPersonQuotaHist
WHERE
    BusinessEntityID = 27
```



| BusinessEntity | SalesYear | CurrentQuota | PrevQuota | NextQuota |
|---|---|---|---|---|
| 275 | 2005 | 367000 | 0 | 556000 |
| 275 | 2005 | 556000 | 367000 | 502000 |
| 275 | 2006 | 502000 | 556000 | 550000 |
| 275 | 2006 | 550000 | 502000 | 1429000 |
| 275 | 2006 | 1429000 | 550000 | 1324000 |
| 275 | 2006 | 1324000 | 1429000 | 0 |

## Filtering

Filtering based on the results of calculating the window function

Window functions are allowed to be used in the request only in the SELECT list and the ORDER BY clause. In all other

offers, including GROUP BY, HAVING and WHERE, they are prohibited. This is explained by the fact that logically they are executed

after these sentences, as well as after non-window aggregate functions, and therefore the aggregate function can be called in

the arguments of the window, but not vice versa.

## Example

If you need to filter or group rows after calculating window functions, you can use a nested query. For example:

```
SELECT
    depname,
    empno,
    salary,
    enroll_date
FROM (
    SELECT
        depname,
        empno,
        salary,
        enroll_date,
        rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
    FROM
        empsalary
) AS ss
WHERE
    pos < 3;
```

## Named window function calls

When several window functions are calculated in a query for identically defined windows, of course, you can write

a separate OVER clause for each of them, but at the same time it will be duplicated, which will inevitably provoke errors.

Therefore, it is better to highlight the definition of a window in the WINDOW clause, and then refer to it in OVER. For example:

## Example

```
SELECT
    sum(salary) OVER w,
    avg(salary) OVER w
FROM
    empsalary
WINDOW
    w AS (PARTITION BY depname ORDER BY salary DESC);
```

## Useful links

SELECT

processing window functions

Window functions

Window functions