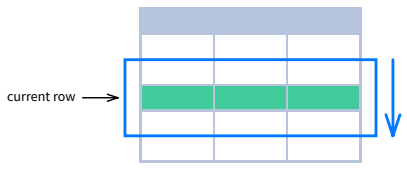
SQL Window Functions Cheat Sheet

LearnSQL.com

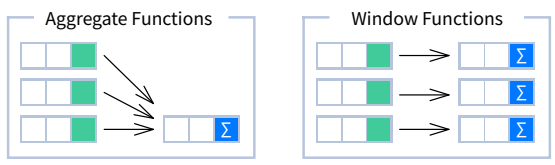
WINDOW FUNCTIONS

Window functions compute their result based on a sliding window frame, a set of rows that are somehow related to the current row.



AGGREGATE FUNCTIONS VS. WINDOW FUNCTIONS

Unlike aggregate functions, window functions do not collapse rows.



SYNTAX

```
SELECT city, month,
  SUM(sold) OVER (
    PARTITION BY city
    ORDER BY month
    RANGE UNBOUNDED PRECEDING) total
FROM sales;
```

```
SELECT <column_1>, <column_2>,
  <window_function> OVER (
    PARTITION BY <...>
    ORDER BY <...>
    <window_frame>) <window_column_alias>
FROM <table_name>;
```

NAMED WINDOW DEFINITION

```
SELECT country, city,
  RANK() OVER country_sold_avg
FROM sales
WHERE month BETWEEN 1 AND 6
GROUP BY country, city
HAVING sum(sold) > 10000
WINDOW country_sold_avg AS (
  PARTITION BY country
  ORDER BY avg(sold) DESC)
ORDER BY country, city;
```

```
SELECT <column_1>, <column_2>,
  <window_function>() OVER <window_name>
FROM <table_name>
WHERE <...>
GROUP BY <...>
HAVING <...>
WINDOW <window_name> AS (
  PARTITION BY <...>
  ORDER BY <...>
  <window_frame>)
ORDER BY <...>;
```

`PARTITION BY`, `ORDER BY`, and window frame definition are all optional.

LOGICAL ORDER OF OPERATIONS IN SQL

1. `FROM`, `JOIN`
2. `WHERE`
3. `GROUP BY`
4. aggregate functions
5. `HAVING`
6. **window functions**
7. `SELECT`
8. `DISTINCT`
9. `UNION`/`INTERSECT`/`EXCEPT`
10. `ORDER BY`
11. `OFFSET`
12. `LIMIT`/`FETCH`/`TOP`

You can use window functions in `SELECT` and `ORDER BY`. However, you can't put window functions anywhere in the `FROM`, `WHERE`, `GROUP BY`, or `HAVING` clauses.

PARTITION BY

divides rows into multiple groups, called **partitions**, to which the window function is applied.

month	city	sold
1	Rome	200
2	Paris	500
1	London	100
1	Paris	300
2	Rome	300
2	London	400
3	Rome	400

`PARTITION BY city`

month	city	sold	sum
1	Paris	300	800
2	Paris	500	800
1	Rome	200	900
2	Rome	300	900
3	Rome	400	900
1	London	100	500
2	London	400	500

Default Partition: With no `PARTITION BY` clause, the entire result set is the partition.

ORDER BY

`ORDER BY` specifies the order of rows in each partition to which the window function is applied.

sold	city	month
200	Rome	1
500	Paris	2
100	London	1
300	Paris	1
300	Rome	2
400	London	2
400	Rome	3

`PARTITION BY city ORDER BY month`

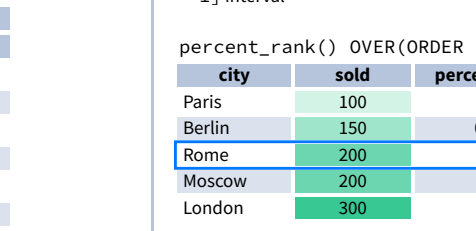
sold	city	month
300	Paris	1
500	Paris	2
200	Rome	1
300	Rome	2
400	Rome	3
100	London	1
400	London	2

Default `ORDER BY`: With no `ORDER BY` clause, the order of rows within each partition is arbitrary.

WINDOW FRAME

A **window frame** is a set of rows that are somehow related to the current row. The window frame is evaluated separately within each partition.

`<ROWS | RANGE | GROUPS> BETWEEN lower_bound AND upper_bound`



The bounds can be any of the five options:

- `UNBOUNDED PRECEDING`
- `n PRECEDING`
- `CURRENT ROW`
- `n FOLLOWING`
- `UNBOUNDED FOLLOWING`

The `lower_bound` must be BEFORE the `upper_bound`.

`ROWS` BETWEEN 1 PRECEDING AND 1 FOLLOWING

city	sold	month
Paris	300	1
Rome	200	1
Paris	500	2
Rome	100	4
Paris	200	4
Paris	300	5
Rome	200	5
London	200	5
London	100	6
Rome	300	6

1 row before the current row and 1 row after the current row

`RANGE` BETWEEN 1 PRECEDING AND 1 FOLLOWING

city	sold	month
Paris	300	1
Rome	200	1
Paris	500	2
Rome	100	4
Paris	200	4
Paris	300	5
Rome	200	5
London	200	5
London	100	6
Rome	300	6

values in the range between 3 and 5 `ORDER BY` must contain a single expression

`GROUPS` BETWEEN 1 PRECEDING AND 1 FOLLOWING

city	sold	month
Paris	300	1
Rome	200	1
Paris	500	2
Rome	100	4
Paris	200	4
Paris	300	5
Rome	200	5
London	200	5
London	100	6
Rome	300	6

1 group before the current row and 1 group after the current row regardless of the value

As of 2024, `GROUPS` is only supported in PostgreSQL 11 and up.

ABBREVIATIONS

ABBREVIATION	MEANING
`UNBOUNDED PRECEDING`	`BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`
`n PRECEDING`	`BETWEEN n PRECEDING AND CURRENT ROW`
`CURRENT ROW`	`BETWEEN CURRENT ROW AND CURRENT ROW`
`n FOLLOWING`	`BETWEEN CURRENT ROW AND n FOLLOWING`
`UNBOUNDED FOLLOWING`	`BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`

DEFAULT WINDOW FRAME

If `ORDER BY` is specified, then the frame is `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

Without `ORDER BY`, the frame specification is `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`.

LIST OF WINDOW FUNCTIONS

Aggregate Functions
- `avg()`
- `count()`
- `max()`
- `min()`
- `sum()`

Ranking Functions
- `row_number()`
- `rank()`
- `dense_rank()`

Distribution Functions
- `percent_rank()`
- `cume_dist()`

Analytic Functions
- `lead()`
- `lag()`
- `ntile()`
- `first_value()`
- `last_value()`
- `nth_value()`

AGGREGATE FUNCTIONS

- **`avg(`**`expr`**`)`** – average value for rows within the window frame
- **`count(`**`expr`**`)`** – count of values for rows within the window frame
- **`max(`**`expr`**`)`** – maximum value within the window frame
- **`min(`**`expr`**`)`** – minimum value within the window frame
- **`sum(`**`expr`**`)`** – sum of values within the window frame

ORDER BY and Window Frame: Aggregate functions do not require an `ORDER BY`. They accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

RANKING FUNCTIONS

- **`row_number()`** – unique number for each row within partition, with different numbers for tied values
- **`rank()`** – ranking within partition, with gaps and same ranking for tied values
- **`dense_rank()`** – ranking within partition, with no gaps and same ranking for tied values

city	price	row_number	rank	dense_rank
		over(order by price)		
Paris	7	1	1	1
Rome	7	2	1	1
London	8.5	3	3	2
Berlin	8.5	4	3	2
Moscow	9	5	5	3
Madrid	10	6	6	4
Oslo	10	7	6	4

ORDER BY and Window Frame: `rank()` and `dense_rank()` require `ORDER BY`, but `row_number()` does not require `ORDER BY`. Ranking functions do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

DISTRIBUTION FUNCTIONS

- **`percent_rank()`** – the percentile ranking number of a row—a value in `[0, 1]` interval: (rank-1) / (total number of rows - 1)
- **`cume_dist()`** – the cumulative distribution of a value within a group of values, i.e., the number of rows with values less than or equal to the current row's value divided by the total number of rows; a value in `(0, 1]` interval

`percent_rank() OVER(ORDER BY sold)`

city	sold	percent_rank
Paris	100	0
Berlin	150	0.25
Rome	200	0.5 ★
Moscow	200	0.5
London	300	1

★ without this row 50% of values are less than this row's value

`cume_dist() OVER(ORDER BY sold)`

city	sold	cume_dist
Paris	100	0.2
Berlin	150	0.4
Rome	200	0.8 ★
Moscow	200	0.8
London	300	1

★ 80% of values are less than or equal to this one

ORDER BY and Window Frame: Distribution functions require `ORDER BY`. They do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

ANALYTIC FUNCTIONS

- **`lead(`**`expr, offset, default`**`)`** – the value for the row *offset* rows after the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = `NULL`
- **`lag(`**`expr, offset, default`**`)`** – the value for the row *offset* rows before the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = `NULL`

`lead(sold) OVER(ORDER BY month)`

month	sold	lead
1	500	300
2	300	400
3	400	100
4	100	500
5	500	NULL

`lag(sold) OVER(ORDER BY month)`

month	sold	lag
1	500	NULL
2	300	500
3	400	300
4	100	400
5	500	100

`lead(sold, 2, 0) OVER(ORDER BY month)`

month	sold	lead
1	500	400
2	300	100
3	400	500
4	100	0
5	500	0

offset = 2

`lag(sold, 2, 0) OVER(ORDER BY month)`

month	sold	lag
1	500	0
2	300	0
3	400	500
4	100	300
5	500	400

offset = 2

- **`ntile(`**`n`**`)`** – divide rows within a partition as equally as possible into *n* groups, and assign each row its group number.

`ntile(3)`

city	sold	ntile
Rome	100	1
Paris	100	1
London	200	1
Moscow	200	2
Berlin	200	2
Madrid	300	2
Oslo	300	3
Dublin	300	3

ORDER BY and Window Frame: `ntile()`, `lead()`, and `lag()` require an `ORDER BY`. They do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

- **`first_value(`**`expr`**`)`** – the value for the first row within the window frame
- **`last_value(`**`expr`**`)`** – the value for the last row within the window frame

`first_value(sold) OVER (PARTITION BY city ORDER BY month)`

city	month	sold	first_value
Paris	1	500	500
Paris	2	300	500
Paris	3	400	500
Rome	2	200	200
Rome	3	300	200
Rome	4	500	200

`last_value(sold) OVER (PARTITION BY city ORDER BY month RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)`

city	month	sold	last_value
Paris	1	500	400
Paris	2	300	400
Paris	3	400	400
Rome	2	200	500
Rome	3	300	500
Rome	4	500	500

Note: You usually want to use `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` with `last_value()`. With the default window frame for `ORDER BY`, `RANGE UNBOUNDED PRECEDING`, `last_value()` returns the value for the current row.

- **`nth_value(`**`expr, n`**`)`** – the value for the *n*-th row within the window frame; *n* must be an integer

city	month	sold	first_value
Paris	1	500	300
Paris	2	300	300
Paris	3	400	300
Rome	2	200	300
Rome	3	300	300
Rome	4	500	300
Rome	5	300	300
London	1	100	NULL

ORDER BY and Window Frame: `first_value()`, `last_value()`, and `nth_value()` do not require an `ORDER BY`. They accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).