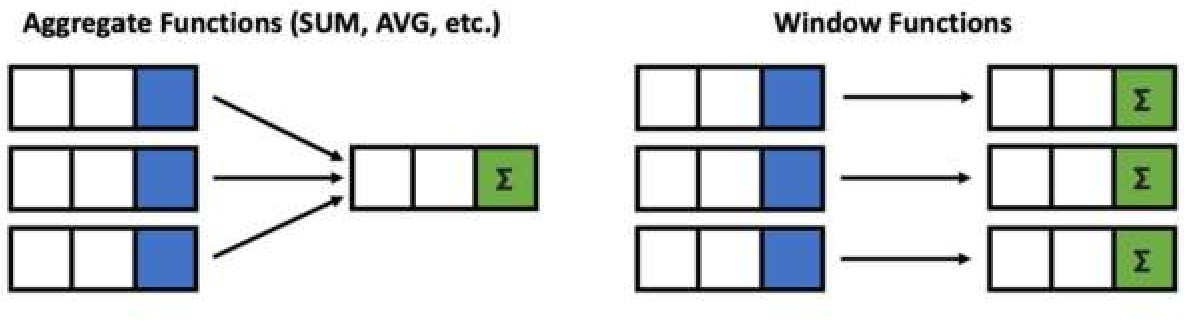
Daily BI Talks

Business Intelligence Chats and Tips for Data Professionals!



Advanced SQL: Windows Function Cheat Sheet

SQL **window functions** are one of the most powerful tools for performing calculations across a defined range of rows while maintaining access to individual row-level details. Unlike aggregate functions that collapse rows into a single value, **window functions allow you to perform calculations over a subset of rows (a window) while preserving the original row structure**.

This in-depth guide explores **what window functions are, how they work, different types, use cases, and practical examples**.

What Are SQL Window Functions?

A **window function** operates on a subset of rows (called a **window**) and returns a value for each row based on calculations across that subset. These functions use the **OVER()** clause, which defines how the rows in the window are partitioned and ordered.

Unlike GROUP BY aggregation, **window functions do not collapse rows**, meaning you can compute rankings, running totals, and moving averages while retaining individual row details.

Key Components of Window Functions

A SQL window function is composed of:

1. **Function Type –** The specific computation performed (e.g., `RANK()`, `SUM()`, `LAG()`).
2. **OVER() Clause –** Defines the window for calculation.
   - **PARTITION BY –** Splits rows into groups for separate calculations.
   - **ORDER BY –** Determines the order of rows within the window.
   - **ROWS/RANGE –** Defines the subset of rows to consider.

Basic Syntax

```
SELECT column_name,
       window_function() OVER(
           PARTITION BY column_name
           ORDER BY column_name
           ROWS BETWEEN n PRECEDING AND m FOLLOWING
       ) AS computed_value
FROM table_name;
```

Types of SQL Window Functions

SQL window functions can be categorized into four main types:

1. **Ranking Functions** (Assign ranks to rows)
2. **Aggregate Window Functions** (Perform cumulative calculations)
3. **Offset (Lag/Lead) Functions** (Access previous or next rows)
4. **Statistical Window Functions** (Provide percentiles, NTILE bucketing)

Let's explore each category with examples.

1. Ranking Functions

Ranking functions **assign unique or non-unique ranks to rows based on the specified order**.

1.1 RANK()

Assigns a rank to each row. Rows with **equal values receive the same rank**, and the next rank **skips numbers**.

Example: Rank Employees by Salary

```
SELECT employee_id, department, salary,
       RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank
FROM employees;
```

- **Same salaries get the same rank**.
- **Next rank is skipped** (e.g., ranks go 1, 2, 2, 4).

1.2 DENSE_RANK()

Works like RANK`()` but **does not skip rank values**.

```
SELECT employee_id, department, salary,
       DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS dense_rank
FROM employees;
```

- If two employees have the same salary, they get the same rank.
- The next rank **does not skip a number** (e.g., ranks go 1, 2, 2, 3).

1.3 ROW_NUMBER()

Assigns a **unique** number to each row, even for ties.

```
SELECT employee_id, department, salary,
       ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS
row_num
FROM employees;
```

- Every row gets a unique number, even for identical values.

2. Aggregate Window Functions

Aggregate functions return **cumulative calculations** across a window of rows.

2.1 SUM()

Computes a running total.

```
SELECT order_id, customer_id, order_date, total_amount,
       SUM(total_amount) OVER (PARTITION BY customer_id ORDER BY order_date)
AS running_total
FROM orders;
```

- Calculates a cumulative sum **per customer** over time.

2.2 AVG()

Computes a moving average.

```
SELECT product_id, sale_date, sales,
       AVG(sales) OVER (PARTITION BY product_id ORDER BY sale_date ROWS
BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving_avg
FROM sales;
```

- Averages sales over a 3-day window (`2 preceding + current row`).

2.3 COUNT()

Counts occurrences **without collapsing rows**.

```
SELECT customer_id, purchase_category,
       COUNT(*) OVER (PARTITION BY customer_id) AS purchase_count
FROM purchases;
```

- Returns the number of purchases per customer.

3. Offset (Lag/Lead) Functions

Offset functions help **access previous or next row values**.

3.1 LAG()

Retrieves the value from the **previous row**.

```
SELECT employee_id, department, salary,
       LAG(salary, 1) OVER (PARTITION BY department ORDER BY salary DESC) AS
previous_salary
FROM employees;
```

- Returns the salary of the **previous employee** in the department.

3.2 LEAD()

Retrieves the value from the **next row**.

```
SELECT employee_id, department, salary,
       LEAD(salary, 1) OVER (PARTITION BY department ORDER BY salary DESC) AS
next_salary
FROM employees;
```

- Returns the salary of the **next employee** in the department.

4. Statistical Window Functions

Used for percentiles, quantiles, and distribution analysis.

4.1 NTILE()

Divides rows into **equal-sized buckets**.

```
SELECT customer_id, total_spent,
       NTILE(4) OVER (ORDER BY total_spent DESC) AS quartile
FROM customers;
```

- Divides customers into **four quartiles** based on spending.

4.2 PERCENT_RANK()

Returns a percentile rank from **0 to 1**.

```
SELECT employee_id, salary,
       PERCENT_RANK() OVER (ORDER BY salary DESC) AS percent_rank
FROM employees;
```

- Returns the relative rank percentage.

Key Differences Between Window Functions and Aggregate Functions

FEATURE	WINDOW FUNCTIONS	AGGREGATE FUNCTIONS
Row Retention	Retains all rows	Collapses rows into groups
Scope	Works over a defined “window”	Works over entire dataset or group

FEATURE	WINDOW FUNCTIONS	AGGREGATE FUNCTIONS
Usage	Ranking, running totals, lag/lead	Sum, average, count per group

When to Use Window Functions

✅ **Ranked Reports:** Sorting employees, customers, or products based on sales, revenue, etc.
✅ **Running Totals:** Computing cumulative values like revenue growth over time.
✅ **Comparing Rows:** Accessing previous/next values with `LAG()` and `LEAD()`.
✅ **Statistical Distributions:** Dividing data into quantiles with `NTILE()`.

Final Thoughts

SQL window functions provide **flexibility, efficiency, and analytical power** by allowing you to perform row-wise calculations while keeping individual row details.

Mastering these functions **optimizes query performance, simplifies complex reporting, and enhances analytical capabilities** in SQL-based applications.

💡 **Try them out today in your database system!**

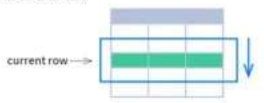
Lastly, here are the very very helpful windows function cheat sheet from learnsql.com you will find very helpful, happy sqling!

SQL Window Functions Cheat Sheet

LearnSQL • com

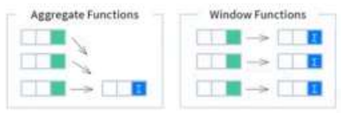
WINDOW FUNCTIONS

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.



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

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.

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

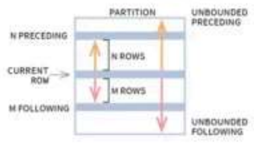
```
1.  FROM, JOIN                 7.  SELECT
2.  WHERE                      8.  DISTINCT
3.  GROUP BY                   9.  UNION/INTERSECT/EXCEPT
4.  aggregate functions        10. ORDER BY
5.  HAVING                     11. OFFSET
6.  window functions           12. LIMIT/FETCH/TOP
```

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.

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

current row

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

current row

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

current row

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

As of 2020, 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 AND 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.

SQL Window Functions Cheat Sheet

LearnSQL • com

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

order by month

lag(sold) OVER(ORDER BY month)

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

order by month

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

order by month
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

order by month
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	1
Paris	100		1
London	200		1
Moscow	200	2	2
Berlin	200		2
Madrid	300		2
Oslo	300	3	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	nth_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).