SQL Window Functions Cheat Sheet

With this SQL Window Functions cheat sheet, you'll have a handy reference guide to the various types of window functions in SQL.

Oct 23, 2022 · 10 min read

Richie Cotton
Webinar & podcast host, course and book author, spends all day chit-chatting about data

TOPICS

SQL

SQL, also known as Structured Query Language, is a powerful tool to search through large amounts of data and return specific information for analysis. Learning SQL is crucial for anyone aspiring to be a data analyst, **data engineer**, or data scientist, and helpful in many other fields such as web development or marketing.

In this cheat sheet, you'll find a handy collection of information about window functions in SQL including what they are and how to construct them.

💻 To easily run all the example code in this tutorial yourself, you can **create a DataLab workbook** for free that has the dataset, SQL, and the code samples pre-configured. Great for experimenting!

datacamp

SQL for Data Science

SQL Window Functions

Learn SQL online at www.DataCamp.com

> Example dataset

We will use a dataset on the sales of bicycles as a sample. This dataset includes:

The [product] table

The product table contains the types of bicycles sold, their model year, and list price.

product_id	product_name	model_year	list_price
1	Treak 820 - 2016	2016	379.99
2	Ritchey Timberwolf Frameset - 2016	2016	749.99
3	Surly Wednesday Frameset - 2016	2016	999.99
4	Trek Fuel EX 8 29 - 2016	2016	2899.99
5	Heller Shagamaw Frame - 2016	2016	1320.99

The [order] table

The order table contains the order_id and its date.

order_id	order_date
1	2016-01-01T00:00:00.000Z
2	2016-01-01T00:00:00.000Z
3	2016-01-02T00:00:00.000Z
4	2016-01-03T00:00:00.000Z
5	2016-01-03T00:00:00.000Z

The [order_items] table

The order_items table lists the orders of a bicycle store. For each order_id, there are several products sold (product_id). Each product_id has a discount value.

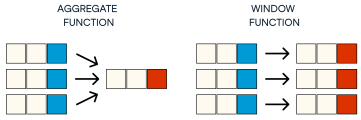
order_id	product_id	discount
1	20	0.2
1	8	0.07
1	10	0.05
1	16	0.05
1	4	0.2
2	20	0.07

What are Window Functions?

A window function makes a calculation across multiple rows that are related to the current row. For example, a window function allows you to calculate.

- Running totals (i.e. sum values from all the rows before the current row)
- 7-day moving averages (i.e. average values from 7 rows before the current row)
- Rankings

Similar to an aggregate function (GROUP BY), a window function performs the operation across multiple rows. Unlike an aggregate function, a window function does not group rows into one single row.



> Syntax

Windows can be defined in the SELECT section of the query.

```
SELECT
 window_function() OVER(
      PARTITION BY partition_expression
      ORDER BY order_expression
      window_frame_extent
 ) AS window_column_alias
 FROM table_name
```

To reuse the same window with several window functions, define a named window using the WINDOW keyword. This appears in the query after the HAVING section and before the ORDER BY section.

```
SELECT
 window_function() OVER(window_name)
 FROM table_name
 [HAVING ...]
 WINDOW window_name AS (
      PARTITION BY partition_expression
      ORDER BY order_expression
      window_frame_extent
 )
 [ORDER BY ...]
```

> Order by

ORDER BY is a subclause within the OVER clause. ORDER BY changes the basis on which the function assigns numbers to rows.

It is a must-have for window functions that assign sequences to rows, including RANK and ROW_NUMBER. For example, if we ORDER BY the expression `price` on an ascending order, then the lowest-priced item will have the lowest rank.

Let's compare the following two queries which differ only in the ORDER BY clause.

```
/* Rank price from HIGH->LOW */
SELECT
    product_name,
    list_price,
    RANK() OVER
        (ORDER BY list_price DESC) rank
FROM products
```

product_name	list_price	rank
Trek Domane SLR 9 Disc - 2018	11999.99	1
Trek Domane SLR 8 Disc - 2018	7499.99	2
Trek Domane SL Frameset - 2018	6499.99	3

```
/* Rank price from LOW->HIGH */
SELECT
    product_name,
    list_price,
    RANK() OVER
        (ORDER BY list_price ASC) rank
FROM products
```

product_name	list_price	rank
Strider Classic 12 Balance Bike - 2018	89.99	1
Sun Bicycles Lil Kitt'n - 2017	109.99	2
Trek Boy's Kickster - 2015/2017	149.99	3

> Partition by

We can use PARTITION BY together with OVER to specify the column over which the aggregation is performed.

Comparing PARTITION BY with GROUP BY, we find the following similarity and difference:

- Just like GROUP BY, the OVER subclause splits the rows into as many partitions as there are unique values in a column.
- However, while the result of a GROUP BY aggregates all rows, the result of a window function using PARTITION BY aggregates each partition independently. Without the PARTITION BY clause, the result set is one single partition.

For example, using GROUP BY, we can calculate the average price of bicycles per model year using the following query.

```
SELECT
 model_year,
 AVG(list_price) avg_price
FROM products
GROUP BY model_year
```

model_year	avg_price
2016	980.39922
2017	1259.931276
2018	1466.479641
2019	3583.333333

What if we want to compare each product's price with the average price of that year? To do that, we use the AVG() window function and PARTITION BY the model year, as such.

```
SELECT
 model_year,
 product_name,
 list_price,
 AVG(list_price) OVER
   (PARTITION BY model_year)
 avg_price
 FROM products
```

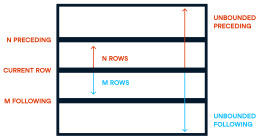
model_year	product_name	list_price	avg_price
2018	Electra Amsterdam Fashion 3i Ladies' - 2017/2018	899.99	1458.479641
2017	Electra Amsterdam Fashion 7i Ladies' - 2017	1899.99	1259.931276
2017	Electra Amsterdam Original 3i - 2016/2017	459.99	1259.931276

Notice how the avg_price of 2018 is exactly the same whether we use the PARTITION BY clause or the GROUP BY clause.

> Window frame extent

A window frame is the selected set of rows in the partition over which aggregation will occur. Put simply, they are a set of rows that are somehow related to the current row.

A window frame is defined by a lower bound and an upper bound relative to the current row. The lowest possible bound is the first row, which is known as UNBOUNDED PRECEDING. The highest possible bound is the last row, which is known as UNBOUNDED FOLLOWING. For example, if we only want to get 5 rows before the current row, then we will specify the range using 5 PRECEDING.



> Accompanying Material

You can use this https://bit.ly/3scZtOK to run any of the queries explained in this cheat sheet.

Have this cheat sheet at your fingertips

Download PDF

Example dataset

We will use a dataset on the sales of bicycles as a sample. This dataset includes:

The [product] table

The product table contains the types of bicycles sold, their model year, and list price.

The [order] table

The order table contains the `order_id` and its date.

The [order_items] table

The order_items table lists the orders of a bicycle store. For each `order_id`, there are several products sold (`product_id`). Each `product_id` has a discount value.

What are Window Functions?

A window function makes a calculation across multiple rows that are related to the current row. For example, a window function allows you to calculate:

- Running totals (i.e. sum values from all the rows before the current row)
- 7-day moving averages (i.e. average values from 7 rows before the current row)
- Rankings

Similar to an aggregate function (`GROUP BY`), a window function performs the operation across multiple rows. Unlike an aggregate function, a window function does not group rows into one single row.

Syntax

Windows can be defined in the `SELECT` section of the query.

```
SELECT
  window_function() OVER(
       PARTITION BY partition_expression
       ORDER BY order_expression
       window_frame_extent
  ) AS window_column_alias
  FROM table_name
```


To reuse the same window with several window functions, define a named window using the `WINDOW` keyword. This appears in the query after the `HAVING` section and before the `ORDER BY` section.

```
SELECT
 window_function() OVER(window_name)
 FROM table_name
 [HAVING ...]
 WINDOW window_name AS (
      PARTITION BY partition_expression
      ORDER BY order_expression
      window_frame_extent
 )
 [ORDER BY ...]
```

POWERED BY datalab

Order by

ORDER BY is a subclause within the OVER clause. ORDER BY changes the basis on which the function assigns numbers to rows.

It is a must-have for window functions that assign sequences to rows, including RANK and ROW_NUMBER . For example, if we ORDER BY the expression price on an ascending order, then the lowest-priced item will have the lowest rank.

Let's compare the following two queries which differ only in the ORDER BY clause

```
/* Rank price from LOW->HIGH */
SELECT
    product_name,
    list_price,
    RANK() OVER
       (ORDER BY list_price ASC) rank
FROM products

/* Rank price from HIGH->LOW */
SELECT
    product_name,
    list_price,
    RANK() OVER
       (ORDER BY list_price DESC) rank
FROM products
```

POWERED BY datalab

Partition by

We can use `PARTITION BY` together with `OVER` to specify the column over which the aggregation is performed.

Comparing `PARTITION BY` with `GROUP BY`, we find the following similarity and difference:

Just like `GROUP BY`, the `OVER` subclause splits the rows into as many partitions as there are unique values in a column.

However, while the result of a `GROUP BY` aggregates all rows, the result of a window function using `PARTITION BY` aggregates each partition independently. Without the `PARTITION BY` clause, the result set is one single partition.

For example, using `GROUP BY`, we can calculate the average price of bicycles per model year using the following query.

```
  SELECT
   model_year,
   AVG(list_price) avg_price
FROM products
GROUP BY model_year
```

POWERED BY datalab

What if we want to compare each product's price with the average price of that year? To do that, we use the `AVG()` window function and `PARTITION BY` the model year, as such.

```
SELECT
model_year,
product_name,
list_price,
AVG(list_price) OVER
  (PARTITION BY model_year)     avg_price
FROM products
```

POWERED BY datalab

Notice how the avg_price of 2018 is exactly the same whether we use the `PARTITION BY` clause or the `GROUP BY` clause.

Window frame extent

A window frame is the selected set of rows in the partition over which aggregation will occur. Put simply, they are a set of rows that are somehow related to the current row.

A window frame is defined by a lower bound and an upper bound relative to the current row. The lowest possible bound is the first row, which is known as `UNBOUNDED PRECEDING`. The highest possible bound is the last row, which is known as `UNBOUNDED FOLLOWING`. For example, if we only want to get 5 rows before the current row, then we will specify the range using `5 PRECEDING`.

Accompanying Material

You can use this **https://bit.ly/3scZtOK** to run any of the queries explained in this cheat sheet.

Ranking window functions

There are several window functions for assigning rankings to rows. Each of these functions requires an `ORDER BY` sub-clause within the `OVER` clause.

The following are the ranking window functions and their description:

Function Syntax	Function Description	Additional notes
ROW_NUMBER()	Assigns a sequential integer to each row within the partition of a result set.	Row numbers are not repeated within each partition.
RANK()	Assigns a rank number to each row in a partition.	• Tied values are given the same rank. • The next rankings are skipped.
PERCENT_RANK()	Assigns the rank number of each row in a partition as a percentage.	• Tied values are given the same rank. • Computed as the fraction of rows less than the current row, i.e., the rank of row divided by the largest rank in the partition.
NTILE(n_buckets)	Distributes the rows of a partition into a	• For example, if we perform the window function NTILE(5) on a

	specified number of buckets.	table with 100 rows, they will be in bucket 1, rows 21 to 40 in bucket 2, rows 41 to 60 in bucket 3, et cetera.
CUME_DIST()	The cumulative distribution: the percentage of rows less than or equal to the current row.	• It returns a value larger than 0 and at most 1. • Tied values are given the same cumulative distribution value.

We can use these functions to rank the product according to their prices.

```
   /* Rank all products by price */
SELECT
   product_name,
   list_price,
   ROW_NUMBER() OVER (ORDER BY list_price) AS row_num,
   DENSE_RANK() OVER (ORDER BY list_price) AS dense_rank,
   RANK() OVER (ORDER BY list_price) AS rank,
   PERCENT_RANK() OVER (ORDER BY list_price) AS pct_rank,
   NTILE(75) OVER (ORDER BY list_price) AS ntile,
   CUME_DIST() OVER (ORDER BY list_price) AS cume_dist
FROM products
```


Value window functions

FIRST_VALUE() and LAST_VALUE() retrieve the first and last value respectively from an ordered list of rows, where the order is defined by ORDER BY .

Value window function	Function
FIRST_VALUE(value_to_return) OVER (ORDER BY value_to_order_by)	Returns the first value in an ordered set of values

LAST_VALUE(value_to_return) OVER (ORDER BY value_to_order_by)	Returns the last value in an ordered set of values
NTH_VALUE(value_to_return, n) OVER (ORDER BY value_to_order_by)	Returns the nth value in an ordered set of values.

To compare the price of a particular bicycle model with the cheapest (or most expensive) alternative, we can use the FIRST_VALUE (or LAST_VALUE).

```
   /* Find the difference in price from the cheapest alternative */
SELECT
  product_name,
  list_price,
  FIRST_VALUE(list_price) OVER (
     ORDER BY list_price
     ROWS BETWEEN
         UNBOUNDED PRECEDING
         AND
         UNBOUNDED FOLLOWING
     ) AS cheapest_price,
FROM products
```

POWERED BY datalab

```
  /* Find the difference in price from the priciest alternative */
SELECT
   product_name,
   list_price,
   LAST_VALUE(list_price) OVER (
     ORDER BY list_price
     ROWS BETWEEN
        UNBOUNDED PRECEDING
        AND
        UNBOUNDED FOLLOWING
     ) AS highest_price
FROM products
```

POWERED BY datalab

Aggregate window functions

Aggregate functions available for `GROUP BY`, such as `COUNT()`, `MIN()`, `MAX()`, `SUM()`, and `AVG()` are also available as window functions.

Function Syntax	Function Description
`COUNT(expression) OVER (PARTITION BY partition_column)`	Count the number of rows that have a non-null expression in the partition.
`MIN(expression) OVER (PARTITION BY partition_column)`	Find the minimum of the expression in the partition.
`MAX(expression) OVER (PARTITION BY partition_column)`	Find the maximum of the expression in the partition.
`AVG(expression) OVER (PARTITION BY partition_column)`	Find the mean (average) of the expression in the partition.

Suppose we want to find the average, maximum and minimum discount for each product, we can achieve it as such.

```
SELECT
   order_id,
   product_id,
   discount,
   AVG(discount) OVER (PARTITION BY product_id) AS avg_discount,
   MIN(discount) OVER (PARTITION BY product_id) AS min_discount,
   MAX(discount) OVER (PARTITION BY product_id) AS max_discount
FROM order_items
```


LEAD, LAG

The `LEAD` and `LAG` locate a row relative to the current row.

Function Syntax	Function Description

LEAD(expression [,offset[,default_value]]) OVER(ORDER BY columns)	Accesses the value stored in a row after the current row.
LAG(expression [,offset[,default_value]]) OVER(ORDER BY columns)	Accesses the value stored in a row before the current row.

Both `LEAD` and `LAG` take three arguments:

- `Expression` : the name of the column from which the value is retrieved
- `Offset` : the number of rows to skip. Defaults to 1.
- `Default_value` : the value to be returned if the value retrieved is null. Defaults to `NULL` .

With `LAG` and `LEAD` , you must specify `ORDER BY` in the `OVER` clause.

`LEAD` and `LAG` are most commonly used to find the value of a previous row or the next row. For example, they are useful for calculating the year-on-year increase of business metrics like revenue.

Here is an example of using lag to compare this year's sales to last year's.

```
   /* Find the number of orders in a year */
WITH yearly_orders AS (
   SELECT
    year(order_date) AS year,
    COUNT(DISTINCT order_id) AS num_orders
    FROM sales.orders
    GROUP BY year(order_date)
)

/* Compare this year's sales to last year's */
SELECT
   *,
   LAG(num_orders) OVER (ORDER BY year) last_year_order,
   LAG(num_orders) OVER (ORDER BY year) - num_orders diff_from_last_year
FROM yearly_orders
```

POWERED BY datalab

Similarly, we can make a comparison of each year's order with the next year's.

```
   /* Find the number of orders in a year */
WITH yearly_orders AS (
   SELECT
    year(order_date) AS year,
    COUNT(DISTINCT order_id) AS num_orders
    FROM sales.orders
    GROUP BY year(order_date)
)

   LEAD(num_orders) OVER (ORDER BY year) next_year_order,
   LEAD(num_orders) OVER (ORDER BY year) - num_orders diff_from_next_year
FROM yearly_orders
```

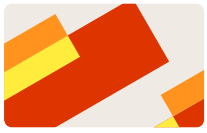
EN

POWERED BY datalab

TOPICS

SQL

Related

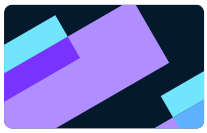


CHEAT-SHEET

SQL Basics Cheat Sheet

CHEAT-SHEET

SQL Joins Cheat Sheet



CHEAT-SHEET

MySQL Basics Cheat Sheet

See More →

Azure Courses

Google Sheets Courses

AI Courses

Data Analysis Courses

Data Visualization Courses

Machine Learning Courses

Data Engineering Courses

Probability & Statistics Courses

DATALAB

Get Started

Pricing

Security

Documentation

CERTIFICATION

Certifications

Data Scientist

Data Analyst

Data Engineer

SQL Associate

Power BI Data Analyst

Tableau Certified Data Analyst

Azure Fundamentals

AI Fundamentals

.