

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	Trek 820 - 2016	2016	379.99
2	Ritchey Timberwolf Frameset - 2016	2016	749.99
3	Sunl 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	2016-01-01T00:00:00Z
2	2016-01-01T00:00:00Z
3	2016-01-02T00:00:00Z
4	2016-01-03T00:00:00Z
5	2016-01-03T00:00:00Z

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.

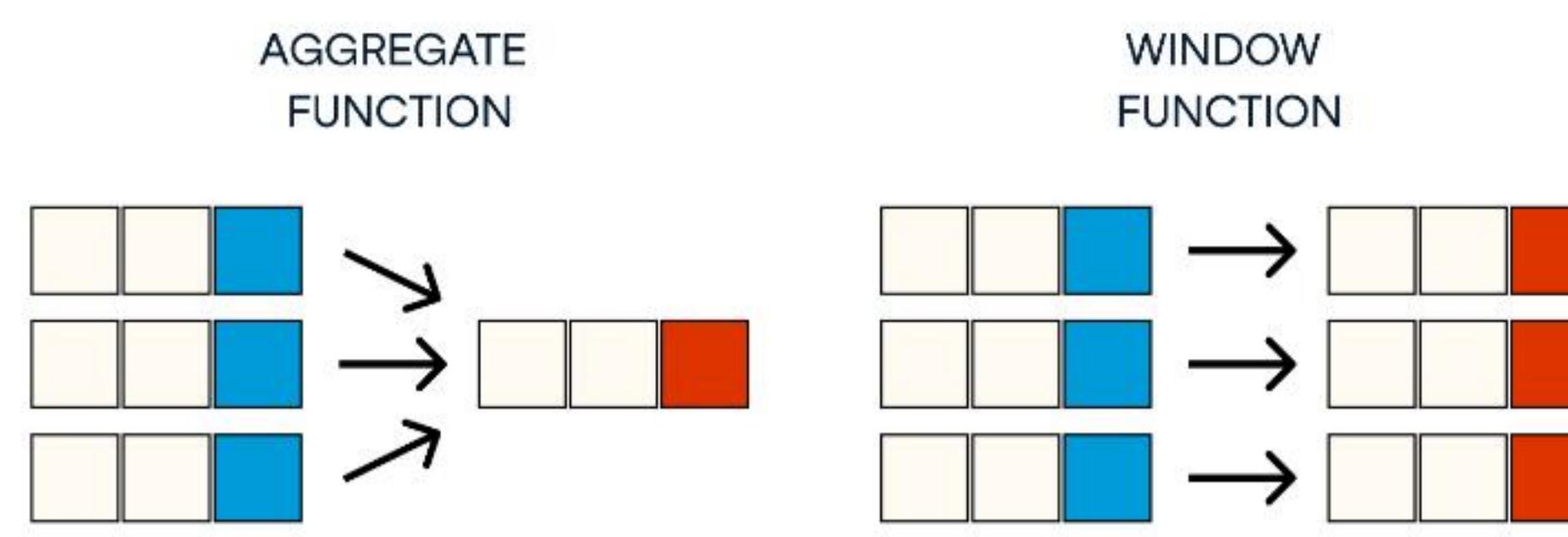
order_id	product_id	discount
1	20	0.2
1	8	0.07
1	10	0.05
1	16	0.05
1	4	0.2
2	20	0.07

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
/* 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	product_name	list_price	rank
Trek Domane SLR 9 Disc - 2018	11999.99	1	Strider Classic 12 Balance Bike - 2018	89.99	1
Trek Domane SLR 8 Disc - 2018	7499.99	28	Sun Bicycles LIL KITT'n - 2017	109.99	2
Trek Domane SL Frameset - 2018	6499.99	3	Trek Boy's Kickster - 2015/2017	149.99	11999.99

> 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	980.29923
2017	1279.931176
2018	1686.479441
2019	2983.323333

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
```

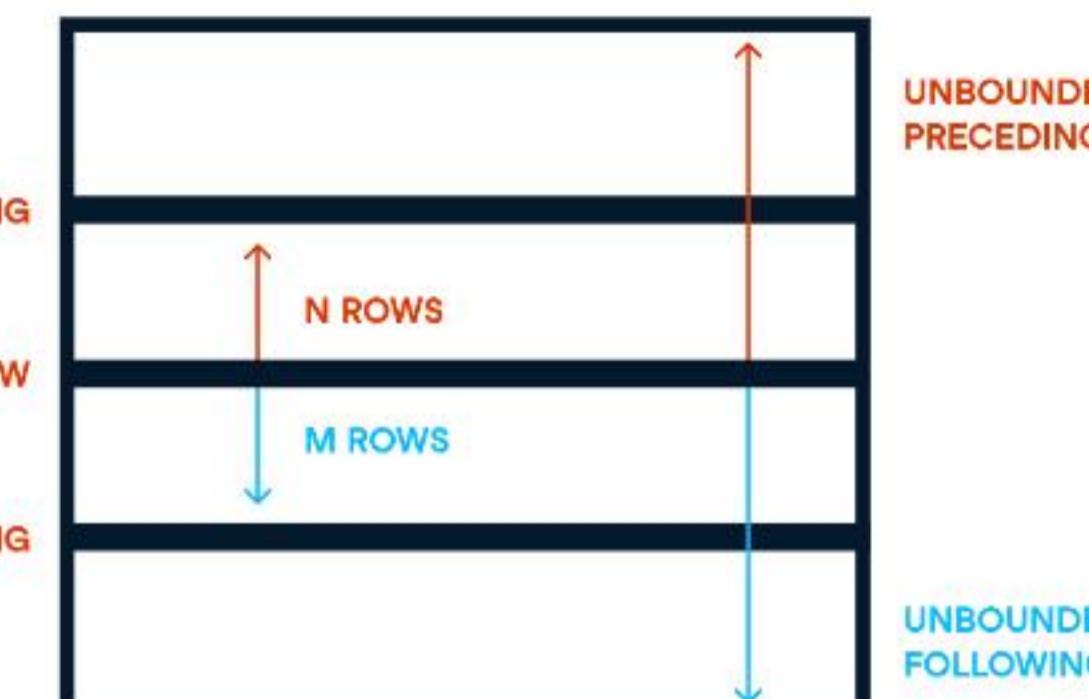
model_year	product_name	list_price	avg_price
2018	Trek Amsterdam Fashion 31 Ladies'	899.99	1686.479441
2017	Electra Amsterdam Fashion 71 Ladies'	1099.99	1279.931176
2017	Electra Amsterdam Original 31	659.99	1279.931176
2017	2015/2017	109.99	1279.931176

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.

SQL for Data Science

SQL Window Functions

Learn SQL online at www.DataCamp.com

> 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 specified number of buckets.	• For example, if we perform the window function NTILE(5) on a table with 100 rows, they will be in bucket 1, rows 21 to 40 in bucket 2, rows 41 to 60 in bucket 3, etcetera.
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
```

product_name	list_price	row_num	dense_rank	rank	pct_rank	ntile	cume_dist
Strider Classic 12 Balance Bike - 2018	89.99	1	1	1	0	1	0.0031152648
Sun Bicycles LIL KITT'n - 2017	109.99	2	2	2	0.003125	1	0.0062305296
Trek Boy's Kickster - 2015/2017	149.99	3	3	3	0.00625	1	0.0124610592
Trek Girl's Kickster - 2017	149.99	4	3	3	0.00625	1	0.0124610592
Trek Kickster - 2018	159.99	5	4	5	0.0125	1	0.015767324
Trek Precaliber 12 Boys - 2017	189.99	6	5	6	0.01625	2	0.0218068536
Trek Precaliber 12 Girls - 2017	189.99	7	5	6	0.01625	2	0.0218068536

> 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
  /* 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
```

product_name	list_price	cheapest_price	highest_price
Strider Classic 12 Balance Bike - 2018	89.99	89.99	11999.99
Sun Bicycles LIL KITT'n - 2017	109.99	109.99	11999.99
Trek Boy's Kickster - 2015/2017	149.99	149.99	149.99
Trek			