SQL for Data Science

SQL Basics Cheat Sheet

Learn SQL online at www.DataCamp.com

What is SQL?

SQL stands for “structured query language”. It is a language used to query, analyze, and manipulate data from databases. Today, SQL is one of the most widely used tools in data.

> The different dialects of SQL

Although SQL languages all share a basic structure, some of the specific commands and styles can differ slightly. Popular dialects include MySQL, SQLite, SQL Server, Oracle SQL, and more. PostgreSQL is a good place to start —since it’s close to standard SQL syntax and is easily adapted to other dialects.

> Sample Data

Throughout this cheat sheet, we’ll use the columns listed in this sample table of `airbnb_listings`

airbnb_listings				
id	city	country	number_of_rooms	year_listed
1	Paris	France	5	2018
2	Tokyo	Japan	2	2017
3	New York	USA	2	2022

> Querying tables

1. Get all the columns from a table

```
SELECT *
FROM airbnb_listings;
```

2. Return the `city` column from the table

```
SELECT city
FROM airbnb_listings;
```

3. Get the `city` and `year_listed` columns from the table

```
SELECT city, year_listed
FROM airbnb_listings;
```

4. Get the listing `id, city,` ordered by the `number_of_rooms` in ascending order

```
SELECT id, city
FROM airbnb_listings
ORDER BY number_of_rooms ASC;
```

5. Get the listing `id, city,` ordered by the `number_of_rooms` in descending order

```
SELECT id, city
FROM airbnb_listings
ORDER BY number_of_rooms DESC;
```

6. Get the first 5 rows from the `airbnb_listings` table

```
SELECT *
FROM airbnb_listings
LIMIT 5;
```

7. Get a unique list of cities where there are listings

```
SELECT DISTINCT city
FROM airbnb_lisitings;
```

> Filtering Data

Filtering on numeric columns

1. Get all the listings where `number_of_rooms` is more or equal to 3

```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms >= 3;
```

2. Get all the listings where `number_of_rooms` is more than 3

```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms > 3;
```

3. Get all the listings where `number_of_rooms` is exactly equal to 3

```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms = 3;
```

4. Get all the listings where `number_of_rooms` is lower or equal to 3

```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms <= 3;
```

5. Get all the listings where `number_of_rooms` is lower than 3

```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms < 3;
```

6. Get all the listings with 3 to 6 rooms

```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms BETWEEN 3 AND 6;
```

Filtering on text columns

7. Get all the listings that are based in ‘Paris’

```
SELECT *
FROM airbnb_listings
WHERE city = ‘Paris’;
```

8. Get the listings based in the ‘USA’ and in ‘France’

```
SELECT *
FROM airbnb_listings
WHERE country IN (‘USA’, ‘France’);
```

9. Get all the listings where the city starts with ‘j’ and where the city does not end in ‘t’

```
SELECT *
FROM airbnb_listings
WHERE city LIKE ‘j%’ AND city NOT LIKE ‘%t’;
```

Filtering on multiple columns

10. Get all the listings in `Paris` where `number_of_rooms` is bigger than 3

```
SELECT *
FROM airbnb_listings
WHERE city = ‘Paris’ AND number_of_rooms > 3;
```

11. Get all the listings in `Paris` OR the ones that were listed after 2012

```
SELECT *
FROM airbnb_listings
WHERE city = ‘Paris’ OR year_listed > 2012;
```

Filtering on missing data

12. Return the listings where `number_of_rooms` is missing

```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms IS NULL;
```

13. Return the listings where `number_of_rooms` is not missing

```
SELECT *
FROM airbnb_listings
WHERE number_of_rooms IS NOT NULL;
```

> Aggregating Data

Simple aggregations

1. Get the total number of rooms available across all listings

```
SELECT SUM(number_of_rooms)
FROM airbnb_listings;
```

2. Get the average number of rooms per listing across all listings

```
SELECT AVG(number_of_rooms)
FROM airbnb_listings;
```

3. Get the listing with the highest number of rooms across all listings

```
SELECT MAX(number_of_rooms)
FROM airbnb_listings;
```

4. Get the listing with the lowest number of rooms across all listings

```
SELECT MIN(number_of_rooms)
FROM airbnb_listings;
```

Grouping, filtering, and sorting

5. Get the total number of rooms for each country

```
SELECT country, SUM(number_of_rooms)
FROM airbnb_listings
GROUP BY country;
```

6. Get the average number of rooms for each country

```
SELECT country, AVG(number_of_rooms)
FROM airbnb_listings
GROUP BY country;
```

7. Get the listing with the maximum number of rooms per country

```
SELECT country, MAX(number_of_rooms)
FROM airbnb_listings
GROUP BY country;
```

8. Get the listing with the lowest amount of rooms per country

```
SELECT country, MIN(number_of_rooms)
FROM airbnb_listings
GROUP BY country;
```

9. For each country, get the average number of rooms per listing, sorted by ascending order

```
SELECT country, AVG(number_of_rooms) AS avg_rooms
FROM airbnb_listings
GROUP BY country
ORDER BY avg_rooms ASC;
```

10. For Japan and the USA, get the average number of rooms per listing in each country

```
SELECT country, AVG(number_of_rooms)
FROM airbnb_listings
WHERE country IN (‘USA’,‘Japan’);
GROUP BY country;
```

11. Get the number of cities per country, where there are listings

```
SELECT country, COUNT(city) AS number_of_cities
FROM airbnb_listings
GROUP BY country;
```

12. Get all the years where there were more than 100 listings per year

```
SELECT year_listed
FROM airbnb_listings
GROUP BY year_listed
HAVING COUNT(id) > 100;
```

Joining Data in SQL

Cheat Sheet

Learn SQL online at www.DataCamp.com

> Definitions used throughout this cheat sheet

Primary key:
A primary key is a field in a table that uniquely identifies each record in the table. In relational databases, primary keys can be used as fields to join tables on.

Foreign key:
A foreign key is a field in a table which references the primary key of another table. In a relational database, one way to join two tables is by connecting the foreign key from one table to the primary key of another.

One-to-one relationship:
Database relationships describe the relationships between records in different tables. When a one-to-one relationship exists between two tables, a given record in one table is uniquely related to exactly one record in the other table.

One-to-many relationship:
In a one-to-many relationship, a record in one table can be related to one or more records in a second table. However, a given record in the second table will only be related to one record in the first table.

Many-to-many relationship:
In a many-to-many relationship, records in a given table 'A' can be related to one or more records in another table 'B', and records in table B can also be related to many records in table A.

> Sample Data

Artist Table

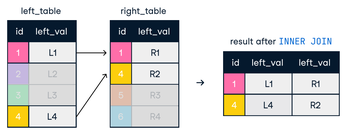
artist_id	name
1	AC/DC
2	Aerosmith
3	Alanis Morissette

Album Table

album_id	title	artist_id
1	For those who rock	1
2	Dream on	2
3	Restless and wild	2
4	Let there be rock	1
5	Rumours	6

INNER JOIN

An inner join between two tables will return only records where a joining field, such as a key, finds a match in both tables.



INNER JOIN join ON one field

```
SELECT *
FROM artist AS art
INNER JOIN album AS alb
ON art.artist_id = alb.artist_id;
```

Result after INNER JOIN:

album_id	name	artist_id
1	AC/DC	1
1	AC/DC	4
2	Aerosmith	2
2	Aerosmith	3

INNER JOIN with USING

```
SELECT *
FROM artist AS art
INNER JOIN album AS alb
USING (artist_id);
```

SELF JOIN

Self-joins are used to compare values in a table to other values of the same table by joining different parts of a table together.

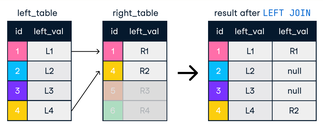
```
SELECT
  art1.artist_id,
  art1.title AS art1_title,
  art2.title AS art2_title
FROM artist as art1
INNER JOIN artist as art2
ON art1.artist_id = art2.album_id;
```

Result after Self join:

artist_id	art1_title	art2_title
1	For those who rock	For those who rock
2	Dream on	Dream on
2	Restless and wild	Dream on
1	Let there be rock	For those who rock

LEFT JOIN

A left join keeps all of the original records in the left table and returns missing values for any columns from the right table where the joining field did not find a match.



LEFT JOIN on one field

```
SELECT *
FROM artist AS art
LEFT JOIN album AS alb
ON art.artist_id = alb.album_id;
```

Result after LEFT JOIN:

artist_id	name	album_id	title	name
1	AC/DC	1	For those who rock	1
1	AC/DC	4	Let there be rock	1
2	Aerosmith	2	Dream on	2
2	Aerosmith	3	Restless and wild	2
3	Alanis Morissette	null	null	null

RIGHT JOIN

A right join keeps all of the original records in the right table and returns missing values for any columns from the left table where the joining field did not find a match. Right joins are far less common than left joins, because right joins can always be re-written as left joins.

RIGHT JOIN on one field

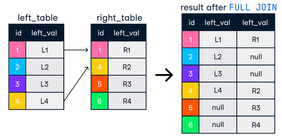
```
SELECT*
FROM artist as art
RIGHT JOIN album AS alb
ON art.artist_id = alb.album_id;
```

Result after RIGHT JOIN:

artist_id	name	album_id	title	name
1	AC/DC	1	For those who rock	1
1	Aerosmith	2	Dream on	2
2	Aerosmith	3	Restless and wild	2
2	AC/DC	4	Let there be rock	1
3	null	5	Rumours	6

FULL JOIN

A full join combines a left join and right join. A full join will return all records from a table, irrespective of whether there is a match on the joining field in the other table, returning null values accordingly.



FULL JOIN on one field

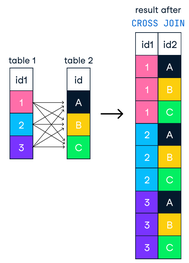
```
SELECT *
FROM artist as art
FULL OUTER JOIN album AS alb
ON art.artist_id = alb.album_id;
```

Result after FULL JOIN:

artist_id	name	album_id	title	name
1	AC/DC	1	For those who rock	1
1	AC/DC	4	Let there be rock	1
2	Aerosmith	2	Balls to the wall	2
2	Aerosmith	3	Restless and wild	2
3	Alanis Morissette	null	null	null
null	null	5	Rumours	6

CROSS JOIN

`CROSS JOIN` creates all possible combinations of two tables. `CROSS JOIN` does not require a field to join `ON`.




```
SELECT name, title
FROM artist
CROSS JOIN album;
```

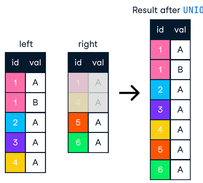
Result after CROSS JOIN:

name	title
AC/DC	For those who rock
AC/DC	Dream on
AC/DC	Restless and wild
AC/DC	Let there be rock
AC/DC	Rumours
Aerosmith	For those who rock
Aerosmith	Dream on
Aerosmith	Restless and wild
Aerosmith	Let there be rock
Aerosmith	Rumours
Alanis Morissette	For those who rock
Alanis Morissette	Dream on
Alanis Morissette	Restless and wild
Alanis Morissette	Let there be rock
Alanis Morissette	Rumours

Set Theory Operators in SQL

UNION

The `UNION` operator is used to vertically combine the results of two `SELECT` statements. For `UNION` to work without errors, all `SELECT` statements must have the same number of columns and corresponding columns must have the same data type. `UNION` does not return duplicates.



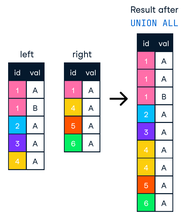

```
SELECT artist_id
FROM artist
UNION
SELECT artist_id
FROM album;
```

Result after UNION:

artist_id
1
2
3
6

UNION ALL

The `UNION ALL` operator works just like `UNION`, but it returns duplicate values. The same restrictions of `UNION` hold true for `UNION ALL`



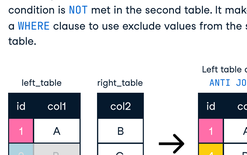

```
SELECT artist_id
FROM artist
UNION ALL
SELECT artist_id
FROM album;
```

Result after UNION ALL:

artist_id
1
2
3
1
2
2
1
6

INTERSECT

The `INTERSECT` operator returns only identical rows from two tables.




```
SELECT artist_id
FROM artist
INTERSECT
SELECT artist_id
FROM album;
```

Result after INTERSECT:

artist_id
1
2

EXCEPT

The `EXCEPT` operator returns only those rows from the left table that are not present in the right table.

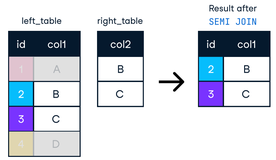

```
SELECT artist_id
FROM artist
EXCEPT
SELECT artist_id
FROM album;
```

Result after EXCEPT:

artist_id
1
2
3

SEMI JOIN

A semi join chooses records in the first table where a condition is met in the second table. A semi join makes use of a `WHERE` clause to use the second table as a filter for the first.




```
SELECT *
FROM album
WHERE artist_id IN
(SELECT artist_id
FROM artist);
```

Result after Semi join:

album_id	title	artist_id
1	For those who rock	1
2	Dream on	2
3	Restless and wild	2

ANTI JOIN

The anti join chooses records in the first table where a condition is `NOT` met in the second table. It makes use of a `WHERE` clause to use exclude values from the second table.


```
SELECT *
FROM album
WHERE artist_id NOT IN
(SELECT artist_id
FROM artist);
```

Result after Anti join:

album_id	title	artist_id
5	Rumours	6

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.

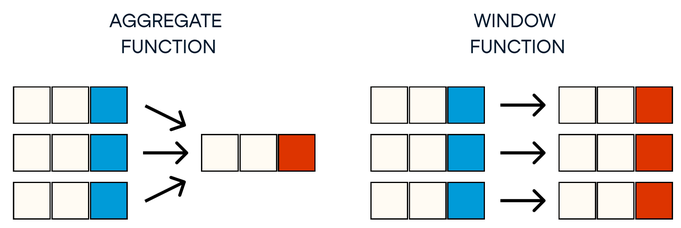
order_id	product_id	discount
1	20	0.2
1	8	0.07
1	10	0.05
1	16	0.05
1	4	0.2
2	20	0.07

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 LOW->HIGH */
SELECT
    product_name,
    list_price,
    RANK() OVER
       (ORDER BY list_price DESC) 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

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

> 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	1658.470441
2019	2583.323333

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
```

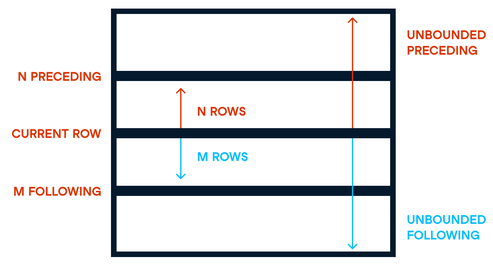
model_year	product_name	list_price	avg_price
2018	Electra Amsterdam Fashion 3i Ladies' - 2017/2018	899.99	1658.470441
2017	Electra Amsterdam Fashion 7i Ladies' - 2017	1099.99	1279.931176
2017	Electra Amsterdam Original 3i - 2015/2017	659.99	1279.931176

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.

datacamp

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, 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
```

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	0.015576324
Trek Precaliber 12 Boys - 2017	189.99	6	5	6	0.015625	2	0.0218068536
Trek Precaliber 12 Girls - 2017	189.99	7	5	6	0.015625	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
```

product_name	list_price	cheapest_price	diff
Strider Classic 12 Balance Bike - 2018	89.99	89.99	0
Sun Bicycles Lil Kitt'n - 2017	109.99	89.99	20
Trek Boy's Kickster - 2015/2017	149.99	89.99	60
Trek Girl's Kickster - 2017	149.99	89.99	60

```
   /* 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	highest_price	diff
Strider Classic 12 Balance Bike - 2018	89.99	11999.99	11910
Sun Bicycles Lil Kitt'n - 2017	109.99	11999.99	11890
Trek Boy's Kickster - 2015/2017	149.99	11999.99	11850

> 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
```

order_id	product_id	discount	avg_discount	min_discount	max_discount
2	16	0.05	0.113191	0.05	0.2
2	20	0.07	0.11253	0.05	0.2
3	20	0.05	0.11253	0.05	0.2
3	3	0.05	0.105581	0.05	0.2

> 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
```

year	num_orders	last_year_order	diff_from_last_year
2016	635	null	null
2017	688	635	-53
2018	292	688	396

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)
)

/* Compare the number of years compared to next year */
SELECT *,
   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
```

year	num_orders	next_year_order	diff_from_next_year
2016	635	688	53
2017	688	292	-396
2018	292	null	null