14th Sep 2023 | 20 minutes read

11 SQL Window Functions Exercises with Solutions

Martyna Sławińska

Window Functions Sql Practice

Emoji Day Promo 😉👌 - 9 hours only!
Get **70+ online SQL courses** and save up to $450
9h : 22m : 37s

In this article, we present 11 practice exercises involving SQL window functions, along with solutions and detailed explanations.

SQL window functions are a powerful feature that lets us extract meaningful insights from our data easily, yet few SQL courses offer SQL window functions exercises. This makes practicing window functions quite difficult. In this article, we'll give you 11 practice exercises that feature window functions.

All exercises shown in this article come from our interactive courses Window Functions and Window Functions Practice Set. The Window Functions course is an in-depth tutorial with over 200 exercises. We recommend this course to learn or review SQL window functions knowledge. Window Functions Practice Set is a practice course for those who know SQL window functions and are looking for more practice.

Window Functions in a Nutshell

SQL window functions are tools that help analyze data in different ways. From computing running totals and moving averages, comparing data within specific subsets, and identifying top performers, to achieving intricate ranking and partitioning, these functions empower us to extract deeper insights from our data – and accomplish complex tasks easily.

SQL window functions offer a versatile toolkit for enhanced data analysis. This toolkit includes:

- ROW_NUMBER(), RANK(), DENSE_RANK(), and `NTILE()` for ranking data.
- `SUM()` , `AVG()` , `COUNT()` , `MAX()` , and `MIN()` for aggregations.
- LEAD() and LAG() for comparing data across rows.
- `FIRST_VALUE()` and `LAST_VALUE()` for extracting boundary values.

Furthermore, the OVER() clause enables precise data partitioning and ordering within these functions, enabling users to perform complex calculations on defined subsets of data.

What's your SQL level ?
Check now

Mastering SQL window functions is becoming increasingly crucial for data professionals, analysts, and engineers. It not only empowers them to efficiently solve complex analytical challenges, it also provides a deeper understanding of data. Practicing SQL window functions will improve your ability to create advanced queries and help you discover new insights from data. This is a vital skill in today's data-focused world.

Before tackling the exercises, you may want to take a look at our window functions cheat sheet, which will remind you of the list of window functions and their syntax.

SQL Window Functions Practice Exercises: Online Movie Store

Before we start the exercises, let's check out the dataset we'll be using.

Dataset

The following exercises use the online movie store database, which contains six tables.

- The `customer` table stores information on all registered customers. The columns are `id` , `first_name` , `last_name` , `join_date` , and `country` .
- The `movie` table contains records of all movies available in the store. The columns are `id` , `title` , `release_year` , `genre` , and `editor_ranking` .
- The `review` table stores customer ratings of the movies. The columns are `id` , `rating` , `customer_id` (references the `customer` table), and `movie _id` (references the `movie` table).
- The `single_rental` table stores information about movies that were rented for a certain period of time by customers. The columns are `id` , `rental_date` , `rental_period` , `platform` , `customer_id` (references the `customer` table), `movie _id` (references the `movie` table), `payment_date` , and `payment_amount` .
- The `subscription` table stores records for all customers who subscribed to the store. The columns are `id` , `length` (in days), `start_date` , `platform` , `payment_date` , `payment_amount` , and `customer_id` (references the `customer` table).
- The `giftcard` table contains information about purchased gift cards. The columns are `id` , `amount_worth` , `customer_id` (references the `customer`

table), `payment_date` , and `payment_amount` .

Now that we are familiar with the dataset, let’s proceed to the SQL practice exercises.

Exercise 1: Rank Rentals by Price

Exercise:

For each single rental, show the `rental_date` , the title of the movie rented, its genre, the payment amount, and the rank of the rental in terms of the price paid (the most expensive rental should have rank = 1). The ranking should be created separately for each movie genre. Allow the same rank for multiple rows and allow gaps in numbering.

Solution Hide solution

Code

```
SELECT
  rental_date,
  title,
  genre,
  payment_amount,
  RANK() OVER(PARTITION BY genre ORDER BY payment_amount DESC)
FROM movie
JOIN single_rental
  ON single_rental.movie_id = movie.id;
```

Solution explanation:

The instruction tells us to show certain information about single rentals and movies. Thus, we join the `single_rental` table with the `movie` table on their common column (i.e. the `movie_id` column).

Next, we need to rank all rentals in terms of the price paid per rental. To do so, we use `RANK()` . Then, in the `OVER()` clause, we order the data by the `payment_amount` column in descending order, so that the most expensive rental has the rank of 1.

As the ranking should be created separately for each movie genre, in the `OVER()` clause, we partition the data by the genre column.

Why did we choose `RANK()` instead of `DENSE_RANK()` or `ROW_NUMBER()`? The instruction says that the same rank for multiple rows is allowed; hence, we reduce the options to `RANK()` and `DENSE_RANK()`. The `ROW_NUMBER()` function assigns consecutive numbers as ranks to successive rows; it doesn't allow multiple rows with the same rank.

Gaps in row numbering are allowed, so we need the `RANK()` function. `DENSE_RANK()` doesn't skip any number in a sequence, even if multiple rows have the same rank. The following table presents these ranking functions and how they work given a list of data values:

VALUE	ROW_NUMBER()	RANK()	DENSE_RANK()
Apple	1	1	1
Apple	2	1	1
Apple	3	1	1
Carrot	4	4	2
Banana	5	5	3
Banana	6	5	3
Peach	7	7	4
Tomato	8	8	5

Check out this article to learn more about different ranking functions.

Exercise 2: Find 2nd Giftcard-Purchasing Customer

Exercise:

Show the first and last name of the customer who bought the second most-recent gift card, along with the date when the payment took place. Assume that a unique rank is assigned for each gift card purchase.

Solution Hide solution

Code

```
WITH ranking AS (
  SELECT
    first_name,
    last_name,
    payment_date,
    ROW_NUMBER() OVER(ORDER BY payment_date DESC) AS rank
  FROM customer
  JOIN giftcard
    ON customer.id = giftcard.customer_id
)

SELECT
  first_name,
  last_name,
  payment_date
FROM ranking
WHERE rank = 2;
```

Solution explanation:

We are going to show information about customers and their gift card purchases, so we need to join the `customer` table with the `giftcard` table on their common column, (`customer_id`).

The instruction says to find the customer who bought the second most-recent gift card. To do that, let's first rank the gift card purchases using the `ROW_NUMBER()` function; we assume that a unique rank is assigned for each gift card purchase.

The inner `SELECT` statement selects customer information and the dates of their gift card purchases. Then, we rank the rows using the `ROW_NUMBER()`

function to mark the second most-recent gift card purchase (i.e. the rank value of 2).

This inner `SELECT` statement is a Common Table Expression (CTE). It is wrapped inside the `WITH` clause and is named `ranking`. We select relevant data from this CTE and provide a condition in the `WHERE` clause to output only the row with rank equal 2.

Why do we need to define a CTE and then query it? Because we cannot use the rank column in the `WHERE` clause of the inner `SELECT`. The reason is the order of execution, which is: `FROM`, `JOINs`, `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, `DISTINCT`, `ORDER BY`, and `LIMIT`. So the rank column is not yet defined at the time when the `WHERE` clause of the inner `SELECT` would be executed.

Exercise 3: Calculate Running Total for Payments

Exercise:

For each single rental, show the `id`, `rental_date`, `payment_amount` and the running total of `payment_amounts` of all rentals from the oldest one (in terms of `rental_date`) to the current row.

Solution

Hide solution

Code

```
SELECT
  id,
  rental_date,
  payment_amount,
  SUM(payment_amount) OVER(
    ORDER BY rental_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM single_rental;
```

Solution explanation:

First, we select information about each single rental from the `single_rental` table.

Next, we are going to find the running total of payment amounts of all rentals using the `SUM()` function (which takes the `payment_amount` column as its argument) with the `OVER()` clause. Here is an article explaining the details about running total and how to compute it in SQL.

The instruction says to find the running total from the oldest rental date until the current row date. So, in the `OVER()` clause, we need to order the data by the `rental_date` column and then define `ROWS` to be counted in the running total, from the oldest date (`BETWEEN UNBOUNDED PRECEDING`) until the current date (`AND CURRENT ROW`).

SQL Window Functions Practice Exercises: Health Clinic

Dataset

The following exercises use a health clinic database that contains two tables.

- The `doctor` table stores information about doctors. The columns are `id` , `first_name` , `last_name` , and `age` .
- The `procedure` table contains information about procedures performed by doctors on patients. The columns are `id` , `procedure_date` , `doctor_id` (references the `doctor` table), `patient_id` , `category` , `name` , `price` , and `score` .

Now that we are familiar with the dataset, let's proceed to the SQL practice exercises.

Exercise 4: Calculate Moving Average for Scores

Exercise:

For each procedure, show the following information: `procedure_date` , `doctor_id` , `category` , `name` , `score` and the average score from the procedures in the same category which are included in the following window frame: the two previous rows, the current row, and the three following rows in terms of the procedure date.

Solution Hide solution

Code

```
SELECT
  procedure_date,
  doctor_id,
  category,
  name,
  score,
  AVG(score) OVER(
    PARTITION BY category
    ORDER BY procedure_date
    ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING)
FROM procedure;
```

Solution explanation:

We are going to show information for each procedure by selecting from the `procedure` table.

Then we'll find the average score from the procedures in the same category. To do that, we use the `AVG()` function with the score column as its argument. This is followed by the `OVER()` clause, where we partition the dataset into categories.

Additionally, we must consider only rows included in the following window frame: the two previous rows, the current row, and the three following rows in terms of the procedure date. We define this data frame within the `OVER()` clause. First, we order the dataset by the `procedure_date` column so the procedures are listed chronologically. And then we define the rows to be considered for calculating the average score value: two previous rows (`BETWEEN 2 PRECEDING`) and three following rows (`AND 3 FOLLOWING`), including the current row.

This is known as a moving average. You can learn more in What a Moving Average Is and How to Compute It in SQL.

Exercise 5: Find the Difference Between Procedure Prices

Exercise:

For each procedure, show the following information: `id` , `procedure_date` , `name` , `price` , `price` of the previous procedure (in terms of the `id`) and the difference between these two values. Name the last two columns `previous_price` and `difference` .

Solution

Hide solution

Code

```
SELECT
  id,
  procedure_date,
  name,
  price,
  LAG(price) OVER(ORDER BY id) AS previous_price,
  price - LAG(price) OVER(ORDER BY id) AS difference
```

```
FROM procedure;
```

Solution explanation:

Again, we start by selecting information about each procedure from the `procedure` table.

The instruction says to show the price of the previous procedure. To do that, we use the `LAG()` function, which returns the previous row value for its argument (here, for the `price` column). To ensure that we choose the previous procedure price in terms of the `id`, we order the dataset by the `id` column in the `OVER()` clause. We alias it as `previous_price`.

Now that we have the price value and the previous price value, we can select the difference between these two values. We simply subtract the `LAG()` function from the `price` column and alias it as `difference`.

Check out How to Calculate the Difference Between Two Rows in SQL to learn more.

Exercise 6: Find the Difference Between the Current and Best Prices

Exercise:

For each procedure, show the:

- `procedure_date`
- `name`
- `price`
- `category`
- `score`
- Price of the best procedure (in terms of the score) from the same category (column `best_procedure`).

- Difference between this `price` and the `best_procedure` (column `difference`).

Solution Hide solution

Code

```
SELECT
  procedure_date,
  name,
  price,
  category,
  score,
  FIRST_VALUE(price) OVER(PARTITION BY category ORDER BY score DESC)
     AS best_procedure,
  price - FIRST_VALUE(price) OVER(PARTITION BY category
     ORDER BY score DESC) AS difference
FROM procedure;
```

Solution explanation:

We start by selecting information about each procedure from the **procedure** table.

The next step is to find the price of the best procedure. We use the `FIRST_VALUE()` function, which returns the first value in an ordered partition of a result set. To get the price of the best procedure from the same category, we must partition the dataset by the `category` column. And to get the price of the best procedure in terms of score, we must order the dataset by the score column in descending order. We alias this expression as `best_procedure` .

Lastly, we find the difference between `price` and `best_procedure` by subtracting the `FIRST_VALUE()` function from the price column.

Exercise 7: Find the Best Doctor per Procedure

Exercise:

Find out which doctor is the best at each procedure. For each procedure, select the procedure name and the first and last name of all doctors who got high scores (higher than or equal to the average score for this procedure). Rank the doctors per procedure in terms of the number of times they performed this procedure. Then, show the best doctors for each procedure, i.e. those having a rank of 1.

Solution

Hide solution

Code

```
WITH cte AS (
  SELECT
    name,
    first_name,
    last_name,
    COUNT(*) c,
    RANK() OVER(PARTITION BY name ORDER BY count(*) DESC) AS rank
  FROM procedure p
  JOIN doctor d
    ON p.doctor_id = d.id
  WHERE score >= (SELECT avg(score)
                  FROM procedure pl
                  WHERE pl.name = p.name)
  GROUP BY name, first_name, last_name
)

SELECT
  name,
  first_name,
  last_name
FROM cte
WHERE rank = 1;
```

Solution explanation:

First, we select the procedure name and information about doctors, so we join the `procedure` table with the `doctor` table on their common column (`doctor_id`).

We want to select all doctors who got high scores (higher than or equal to the average score for this procedure). To do that, we define the `WHERE` clause condition for the score column. The `score` column must store a value equal to or greater than the average score for the current row's procedure.

Let's rank the doctors per procedure. We'll use the `RANK()` function with the `OVER()` clause, where we partition the dataset by the procedure name. Additionally, we must rank in terms of the number of times the doctor performed this procedure. To get the number of times the doctor performed this procedure, we must `COUNT(*)` while grouping by the procedure name and the first and last name of the doctor (that is, we are grouping by all columns listed in the `SELECT` statement).

All we've done until now is define a Common Table Expression (CTE), which is the inner `SELECT` statement enclosed by the `WITH` clause and named `cte` .

Want to learn about window functions? Click here for a great interactive experience!

Now we select the relevant columns from this CTE. To get the best doctors for each procedure (those having a rank of 1), we define the `WHERE` clause with the condition for the `rank` column.

Why do we need to define a CTE and then query it? Because we cannot use the `rank` column in the `WHERE` clause of the inner `SELECT` . The reason is the order of execution, which is: `FROM` , `JOINs` , `WHERE` , `GROUP BY` , `HAVING` , `SELECT` , `DISTINCT` , `ORDER BY` , and `LIMIT` . The `rank` column has not been defined when the `WHERE` clause is executed.

SQL Window Functions Practice Exercises: Athletic Championships

Dataset

The following exercises use the athletic championships database that contains eight tables.

- The `competition` table stores information about competitions. The columns are `id` , `name` , `start_date` , `end_date` , `year` , and `location` .
- The **discipline** table stores information for all running disciplines (from the short-distance runs (e.g. the 100 meter) to the long-distance runs (e.g. the marathon)). The columns are `id` , `name` , `is_men` , and `distance` .
- The `event` table stores information about the competition and discipline for each event. The columns are `id` , `competition_id` (references the `competition` table), and `discipline_id` (reference the `discipline` table).
- The `round` table stores the rounds of each event. The columns are `id` , `event_id` (references the `event` table), `round_name` , `round_number` , and `is_final` .
- The `race` table stores data for each race of each round. The columns are `id` , `round_id` (references the `round` table), `round_name` (same as in the `round` table), `race_number` , `race_date` , `is_final` (same as in the `round` table), and `wind` .
- The `athlete` table stores information about athletes participating in the competition. The columns are `id` , `first_name` , `last_name` , `nationality_id` (references the `nationality` table), and `birth_date` .
- The `nationality` table stores information about athlete's countries of origin. The columns are `id` , `country_name` , and `country_abbr` .
- The `result` table stores information for all participants of a particular event. The columns are `race_id` (references the `race` table), `athlete_id` (references the `athlete` table), `result` , `place` , `is_dsq` , `is_dns` , and `is_dnf` .

Now that we are familiar with the dataset, let's proceed to the SQL practice exercises.

Exercise 8: Calculate the Difference Between Daily Wind Speed Averages

Exercise:

For each date in which there was a race, display the `race_date`, the average wind on this date rounded to three decimal points, and the difference between the average wind speed on this date and the average wind speed on the date before, also rounded to three decimal points. The columns should be named `race_date`, `avg_wind`, and `avg_wind_delta`.

Solution:

Hide solution

Code

```
SELECT
  race_date,
  ROUND(AVG(wind), 3) AS avg_wind,
  ROUND(AVG(wind) - LAG(AVG(wind)) OVER(ORDER BY race_date), 3)
     AS avg_wind_delta
FROM race
GROUP BY race_date;
```

Solution explanation:

We are going to display race information for each race date, so we select data from the `race` table.

To find the average wind speed on this date rounded to three decimal points, we use the `AVG()` function with the `wind` column as its argument. Then, we enclose it within the `ROUND()` function and round it to three decimal places. Note that we must group by the `race_date` column, since we use the `AVG()` aggregate function.

We can get the average wind on the date before by using the `LAG()` function with the `AVG(wind)` value as its argument. The `OVER()` clause defines that we order the entire dataset by the `race_date` column to have the data rows listed chronologically.

Since we want to see the difference between the average wind speed on this date and the average wind speed on the date before, we subtract `LAG(AVG(wind))` from `AVG(wind)`. And to round it to three decimal places, we use the `ROUND()` function again.

Exercise 9: Compare the Best and Previous Results

Exercise:

For each woman who ran in the final round of the women's marathon in Rio, display the following information:

- The place they achieved in the race.
- Their first name.
- Their last name.
- `comparison_to_best` – The difference between their time and the best time in this final.
- `comparison_to_previous` – The difference between their time and the result for the athlete who got the next-highest

Sort the rows by the `place` column.

Solution: Hide solution

Code

```
SELECT
  place,
  first_name,
  last_name,
  result - FIRST_VALUE(result) OVER (ORDER BY result)
     AS comparison_to_best,
  result - LAG(result) OVER(ORDER BY result)
     AS comparison_to_previous
FROM competition
JOIN event
  ON competition.id = event.competition_id
JOIN discipline
  ON discipline.id = event.discipline_id
JOIN round
  ON event.id = round.event_id
JOIN race
  ON round.id = race.round_id
JOIN result
  ON result.race_id = race.id
JOIN athlete
  ON athlete.id = result.athlete_id
WHERE competition.name = 'Rio de Janeiro Olympic Games'
  AND discipline.name = 'Women''s Marathon'
  AND round.is_final IS TRUE
ORDER BY place;
```

Solution explanation:

We are going to use information about competitions, disciplines, rounds, athletes, and results. Therefore, we must join all these tables on their common columns, as mentioned in the dataset introduction.

The instruction says to display information for each woman who ran in the final round of the women's marathon in Rio. We cover it in the `WHERE` clause that contains the following conditions:

- The competition name must be `Rio de Janeiro Olympic Games` .
- The discipline name must be `Women's Marathon` .
- The round must be the final round.

Next we select the place column from the `result` table and the `first_name` and `last_name` columns from the `athlete` table.

To find the difference between their time and the best time in this final, we use the `FIRST_VALUE()` function with the `result` column as its argument. This is followed by the `OVER()` clause, which orders the dataset by the `result` column. Then we subtract this `FIRST_VALUE()` function from the current row `result` . We alias it as `comparison_to_best` .

To find the difference between their time and the result for the athlete who got the next-better place, we use the `LAG()` function with the `result` column as its argument to get the previous result. Once again, this is followed by the `OVER()` clause to order the dataset by the `result` column (to ensure we get the next better result). Then we subtract this `LAG()` function from the current row `result`. We alias it as `comparison_to_previous`.

Finally, we sort the rows by the place column using the `ORDER BY` clause.

SQL Window Functions Practice Exercises: Website Statistics

Dataset

The following exercises use the website statistics database that contains two tables.

- The `website` table stores information about websites. The columns are `id`, `name`, `budget`, and `opened`.
- The `statistics` table stores statistics for each The columns are website_id (references the `website` table), `day`, `users`, `impressions`, `clicks`, and `revenue`.

Now that we are familiar with the dataset, let's proceed to the SQL practice exercises.

Exercise 10: Look Ahead with the *LEAD()* Function

Exercise:

Take the statistics for the website with id = 2 between 1 and 14 May 2016 and show the day, the number of users, and the number of users 7 days later.

Note that the last 7 rows don't have a value in the last column. This is because no rows '7 days from now' can be found for them. For these cases, show -1 instead of `NULL` if no `LEAD()` value is found.

Solution: Hide solution

Code

```
SELECT
  day,
  users,
  LEAD(users, 7, -1) OVER(ORDER BY day)
FROM statistics
WHERE website_id = 2
  AND day BETWEEN '2016-05-01' AND '2016-05-14';
```

Solution explanation:

We are going to show the day, the number of users, and the number of users 7 days later. The first two values come from the `statistics` table – these are the day and users columns. The last value must be calculated using the `LEAD()` function.

We want to see the users column value after seven days; therefore, we pass the `users` column as the first argument and the value of 7 as the second argument to the `LEAD()` function. And to ensure that we show -1 instead of NULL if no `LEAD()` value is found, we pass the third argument as `-1` .

The `LEAD()` function is followed by the `OVER()` clause. This clause contains the condition to order the dataset by the day column, as the statistics should be ordered chronologically.

To show the statistics for the website with id = 2 between 1 and 14 May 2016, we need to define the relevant conditions in the `WHERE` clause.

Exercise 11: Look Back with the *LAG()* Function

Exercise:

Show the statistics for the website with `id = 3` that include day, revenue, and the revenue 3 days before. Show -1.00 for rows with no revenue value 3 days before.

Solution: Hide solution

Code

```
SELECT
  day,
  revenue,
  LAG(revenue, 3, -1.00) OVER(ORDER BY day)
FROM statistics
WHERE website_id = 3;
```

Solution explanation:

We are going to show the day, revenue, and the revenue 3 days before. The first two values come from the `statistics` table – these are the day and revenue columns. The last value must be calculated using the `LAG()` function.

We want to see the revenue column value from three days before the current row; therefore, we pass the revenue column as the first argument and the value of 3 as the second argument to the `LAG()` function. And to ensure that we show -1.00 for rows with no revenue value 3 days before, we pass the third argument as -1.00.

The `LAG()` function is followed by the `OVER()` clause. This contains the condition to order the dataset by the `day` column, as the statistics should be ordered chronologically.

To show the statistics for the website with `id = 3`, we need to define a condition in the `WHERE` clause.

More SQL Window Functions Practice

The SQL window functions practice exercises presented in this article provide a comprehensive platform for honing your SQL and data analysis skills one query at a time. These exercises come from our courses; to find additional practice exercises, visit the courses linked below.

1. Window Functions
2. Window Functions Practice Set

LearnSQL.com lets you learn SQL by writing SQL code on your own. You build your SQL skills gradually. Each new concept is reinforced by an interactive exercise. By actually writing SQL code, you build your confidence.

If you aim to learn about or refresh your knowledge of window functions, we suggest you start with the Window Functions course, which offers a thorough exploration of this topic. For those seeking to hone their skills in window functions, explore our Window Functions Practice Set. It has 100 exercises structured into three distinct parts, each utilizing a different dataset.

Sign up now and get started for free. Good luck!

Tags: Window Functions Sql Practice

You may also like

22 Exercises to Practice SQL Commands

Take your SQL knowledge from beginner to professional with these real-world practice SQL examples.

Read more

How to Practice SQL Window Functions Online

Here are several efficient ways of practicing SQL Window Functions online.

Read more

Subscribe to our newsletter

Join our monthly newsletter to be notified about the latest posts.

Email address

Subscribe

Quick links

Courses	Blog
Pricing	Cookbook
For Students	LearnPython.com
Affiliate Program	Vertabelo.com

Assistance

Need assistance? Drop us a line at **contact@learnsql.com**

Write to us

Follow us

Copyright ©2016-2024 **Vertabelo SA** All rights reserved

Terms of service
Privacy policy
Imprint