SQL Window Functions [With Example Queries + Cheat Sheet]

Last updated: November 22, 2023

Geekflare articles are written by humans for humans.

John Walter
Contributor

Rashmi Sharma
Editor

Have you been using SQL Window functions? Do you need some quick example queries that you can always look through while working with relational databases or a cheat sheet that you can incorporate into your developer workflow?

Window functions are calculation functions that reduce the complexity of SQL queries, increasing their efficiency. They are derived from two components: a window, a set of rows, and functions, which entail predefined SQL code blocks for performing data operations. By using SQL Window functions, you can perform advanced analytics without writing complex queries.

When analyzing or creating data reports, you can use SQL window functions to aggregate or compare data within specified windows of rows. These functions let you avoid self-joins or subqueries, aggregate data without collapsing results into a single value for an entire table, and compare values for rows using calculations such as totals, percentages, or even rankings.

SQL window functions work in three simple steps. First, you define a window. This is because Windows functions operate on a set of rows defined using the **OVER()** clause. Next, the **PARTITION BY** clause divides the result set into partitions to which the function is applied. And lastly, **ORDER** by clause to determine the order of rows in each partition.

> **NOTE:** Windowing in SQL isn't to be confused with **Aggregation.** Aggregation involves traditional functions, for instance, (SUM and AVG), which will operate on multiple rows, collapsing them into a single result. To distinguish them, remember that Window functions work on a window (specified set of rows), maintaining individual rows in their output.

Let's get into a hands-on approach to working with SQL window functions. For every category, you'll have a table summary for quick reference.

Syntax of SQL Window Functions

The general syntax for window functions comprises three components: the function itself, the `OVER` clause, and an optional `PARTITION BY`. Here's a breakdown of each.

#1. Function

As for the function, it indicates the computation operation you're targeting at the row. It could be standard aggregate functions (e.g., `SUM`, `AVG`, and `COUNT`). Or an analytic function (e.g., `ROW_NUMBER`, `RANK`, `LEAD`, and `LAG`).

Here's an example using the `SUM` function.

```
  column2,
  column3,
  SUM(column3) OVER () AS total_weight
FROM
  your_table;
```

#2. OVER Clause

Using the ***OVER*** clause, you define the rows over which the function will operate. There are two constituents: ***PARTITION BY*** and ***ORDER BY***. The first is optional and divides results into sections to which the window function is applied.

For the latter, it specifies the order of rows in each partition. When not specified, the function treats the window as an unordered set. Here's an example.

```
SELECT
  column1,
  column2,
  column3,
  SUM(column3) OVER (PARTITION BY column1 ORDER BY column2) AS
running_total_weight
FROM
  your_table;
```

#3. PARTITION BY Clause

This clause, optional as it is, divides the result into partitions to which the window function is applied. Its superpower can be realized when performing calculations independently with each partition. Here's an illustration.

```
SELECT
  column1,
  column2,
  column3,
  AVG(column3) OVER (PARTITION BY column1) AS avg_weight_per_group
```

For the above sample, the ***AVG*** function calculates the average of ***column3*** independently for each distinct value in ***column1*** . Simply put, for every group of items in ***column1*** , you'll have ***column3*** with the average weight for that specific item.

The general syntax for a window SQL function is:

```
SELECT
  column1,
  column2,
  column3,
  SUM(column3) OVER (PARTITION BY column1 ORDER BY column2) AS
window_function_result
FROM
  your_table;
```

However, the above syntax may vary slightly based on the choice of database, but the overall structure remains consistent.

Aggregate Window Functions in SQL

Aggregate window functions in SQL reuse prevailing aggregate functions, such as ***COUNT()*** or ***SUM()*** , altering how the Aggregation is defined and the results format. This means they perform calculations on a window related to the current row within the result set. You can use them to obtain aggregate values based on a specified window/frame. Here's a brief description of each.

#1. MIN()

This function returns the minimum value of a specified expression over a frame.

```
MIN(column) OVER (PARTITION BY partition_expression ORDER BY
sort_expression
                 ROWS BETWEEN start AND end)
```

rows around that row.

```
SELECT
column1,
column2,
  MIN(column2) OVER (ORDER BY date_column ROWS BETWEEN 2 PRECEDING AND
CURRENT ROW) AS moving_min
FROM
  your_table;
```

So, ***column1*** and ***column2*** are the columns you are selecting, ***MIN(column2)*** being the aggregate function calculating the minimum value and OVER(ORDER BY some_column ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) defining the window for calculation. It specifies the window, which includes the current row and two preceding it, based on the order specifier some_column. This, in turn, dictates that each row in the result set ***moving_min*** contains the minimum value of ***column2*** . Note that I did not include the ***PARTITION BY*** clause.

#2. MAX()

Our function here returns the maximum value of a specified expression over a window. Here's the syntax.

```
MAX(column) OVER (PARTITION BY partition_expression ORDER BY
sort_expression
                ROWS BETWEEN start AND end)
```

Here's an example query that calculates the running maximum of a column over the entire result set, ordered by the ***date_column*** .

```
SELECT
  column,
  MAX(column) OVER (ORDER BY date_column ROWS UNBOUNDED PRECEDING) AS
running_max
FROM
  your_table;
```

syntax.

```
AVG(column) OVER (PARTITION BY partition_expression ORDER BY
sort_expression
                  ROWS BETWEEN start AND end)
```

For an example query, consider the example below where the query calculates the average over a window that includes the current row, the one preceding it, and the one after it ordered by the ***date_column*** .

```
SELECT
  column,
  AVG(column) OVER (ORDER BY date_column ROWS BETWEEN 1 PRECEDING AND
1 FOLLOWING) AS moving_avg
FROM
  your_table;
```

#4. SUM()

It is used when you want to return the sum of a specified expression over a frame. Below is the syntax.

```
SUM(column) OVER (PARTITION BY partition_expression ORDER BY
sort_expression
                  ROWS BETWEEN start AND end)
```

For a sample query, consider the cumulative sum below. In this case, the query calculates the sum of a column over the entire result set, ordered by the ***date_column*** .

```
SELECT
  column,
  SUM(column) OVER (ORDER BY date_column ROWS UNBOUNDED PRECEDING) AS
cumulative_sum
FROM
  your_table;
```

```
COUNT(column) OVER (PARTITION BY partition_expression ORDER BY
sort_expression
                    ROWS BETWEEN start AND end)
```

For an example use case, consider the case below where the query calculates the running count of rows over the entire result set, as ordered by the ***`date_column`*** .

```
SELECT
  column,
  COUNT(column) OVER (ORDER BY date_column ROWS UNBOUNDED PRECEDING)
AS running_count
FROM
  your_table
```

Here's a quick cheat sheet for aggregate window functions.

Function	Purpose	Usage Example
MIN()	Returns the minimum value of a specified expression over a frame.	`MIN(column) OVER (ORDER BY date_column ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)` – Calculates moving minimum for each row.
MAX()	Returns the maximum value of a specified expression over a window.	`MAX(column) OVER (ORDER BY date_column ROWS UNBOUNDED PRECEDING)` – Calculates running maximum over the entire result set
AVG()	AVG() Returns the average value of a specified expression over a frame.	`AVG(column) OVER (ORDER BY date_column ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)` – Calculates moving average over a window.
SUM()	Returns the sum of a specified expression over a frame.	`SUM(column) OVER (ORDER BY date_column ROWS UNBOUNDED PRECEDING)` – Calculates cumulative sum over the entire result set.

COUNT()	Returns the number of rows in a window or the total row count.	`COUNT(column) OVER (ORDER BY date_column ROWS UNBOUNDED PRECEDING)` – Calculates the running count of rows over the entire result set.

Value Window Functions in SQL

In SQL, value window functions are used when allocating row values from other rows. Unlike aggregate functions – which return a single value for each group – they'll return a value for each row as per a specific window or frame of rows.

This operation model, in turn, allows you to access data from other rows relative to the current window, facilitating powerful analytical and reporting capabilities.

Here's a brief overview of each of the value window functions.

#1. LEAD()

The ***LEAD()*** SQL window function works opposite to the ***LAG()*** and is used to return values of succeeding rows. Simplified, it shifts values one row up. The syntax for calling it is similar to ***LAG()***. Here's how to write it.

```
LEAD(column, offset, default_value) OVER (PARTITION BY
partition_expression ORDER BY sort_expression)
```

For your example query, see the case below, which retrieves the next value of a column based on the ordering criterion, `*date_column*`, with a default 0 value if there's no next value.

```
SELECT
  column,
  LEAD(column, 1, 0) OVER (ORDER BY date_column) AS lead_value
FROM
  your_table;
```

#2. LAG()

such cases, the window function will include the ***ORDER BY*** in the ***OVER*** clause since the order aspect needs to be considered. The syntax is as follows:

```
LAG(column, offset, default_value) OVER (PARTITION BY
partition_expression ORDER BY sort_expression)
```

An example query can be written as shown below. The query extracts the previous value of a column ordered by the ***date_column*** . When there's no previous value, a default value of 0 occurs.

```
SELECT
  column,
  LAG(column, 1, 0) OVER (ORDER BY date_column) AS lag_value
FROM
  your_table;
```

#3. FIRST_VALUE()

You can use this function to retrieve the value of a specified expression for the first row within a window. The syntax:

```
FIRST_VALUE(column) OVER (PARTITION BY partition_expression ORDER BY
sort_expression)
```

Example:

```
SELECT
  column,
  FIRST_VALUE(column) OVER (PARTITION BY category_column ORDER BY
date_column) AS first_value
FROM
  your_table;
```

The query retrieves the last value of a column within each partition, as ordered by `date_column` .

syntax:

```
LAST_VALUE(column) OVER (PARTITION BY partition_expression ORDER BY
sort_expression)
```

Example:

```
SELECT
  column,
  LAST_VALUE(column) OVER (PARTITION BY category_column ORDER BY
date_column) AS last_value
FROM
  your_table;
```

By now, it should be a little obvious...

#5. N_TH VALUE()

As the last group member, this function can retrieve the value at a specified position within a window frame. You'll also specify the expression you want to retrieve the nth value. The expression could be a column, mathematical expression, or any valid SQL expression representing the data you're interested in. For our examples, we've used a column. It's been the same for ***`LAST_VALUE()`*** and ***`FIRST_VALUE()`*** .

The syntax:

```
NTH_VALUE(column, n) OVER (PARTITION BY partition_expression ORDER BY
sort_expression)
```

Sample:

```
SELECT
  column,
  NTH_VALUE(column, 3) OVER (PARTITION BY category_column ORDER BY
date_column) AS third_value
```

Function	Purpose	Usage Example
LEAD()	Returns values of succeeding rows within a window.	**`LEAD(column, 1, 0) OVER (ORDER BY date_column) AS lead_value`** – Retrieves the next value based on the ordering criterion, with a default value if there's no next value.
LAG()	Returns values of preceding rows within a window.	**`LEAD(column, 1, 0) OVER (ORDER BY date_column) AS lead_value`** – Retrieves the next value based on the ordering criterion, with a default value if there's no next value.
FIRST_VALUE()	Retrieves the value of a specified expression for the first row within a window.	**`FIRST_VALUE(column) OVER (PARTITION BY category_column ORDER BY date_column) AS first_value`** – Gets the first value within each category based on date order.
LAST_VALUE()	Retrieves the last value of a specified expression within a defined frame.	**`LAST_VALUE(column) OVER (PARTITION BY category_column ORDER BY date_column) AS last_value`** – Gets the last value within each category based on date ordering.
NTH_VALUE()	Retrieves the value at a specified position within a window frame.	**`NTH_VALUE(column, 3) OVER (PARTITION BY category_column ORDER BY date_column) AS third_value`** – Gets the third value within each category based on date ordering.

Ranking Window Function in SQL

Ranking window functions in SQL are helpful when assigning numbers (ranks or positions) to rows within a result set based on a specific ordering criterion. They are useful when analyzing and ordering data and identifying relative positions for rows.

If window functions are unavailable, you'd have to write multiple nested queries, which are inefficient. Keynote in ranking window functions: the ***`ORDER BY`*** clause must always be

#1. RANK()

The ***RANK( )*** function is the most common and assigns ranking values based on your specification order. Its syntax...

```
RANK( ) OVER (PARTITION BY partition_expression ORDER BY
sort_expression
```

To put that into perspective, look at the query below. It assigns a rank to each employee based on the sales performance; the highest sales get the lowest rank.

```
SELECT
  employee_id,
  sales,
  RANK( ) OVER (ORDER BY sales DESC) AS sales_rank
FROM
  sales_table;
```

#2. DENSE RANK()

While it operates similarly to the `*RANK()*`, it has a critical distinguishing feature: if there are ties in the data set, they get the same ranking value. It does not skip any numbers, assigning succeeding values to the consequent row. Write it as...

```
DENSE_RANK( ) OVER (PARTITION BY partition_expression ORDER BY
sort_expression)
```

For your test case, consider the query below. It allocates a dense rank to each student based on ***test_score*** , leaving no gaps even in the presence of tie scores.

```
SELECT
  student_id,
  test_score,
  DENSE_RANK( ) OVER (ORDER BY test_score DESC) AS score_rank
FROM
  scores_table
```

It's the simplest of all. Following the order defined by the `ORDER BY` clause in the `OVER` subsection gives numbers to all rows, starting from 1. The syntax is as...

```
ROW_NUMBER( ) OVER (PARTITION BY partition_expression ORDER BY
sort_expression)
```

In the example query below, we assign unique row numbers to each product item based on ***sales*** beginning with 1 from the highest sales.

```
SELECT
  product_id,
  category,
  ROW_NUMBER( ) OVER (PARTITION BY category ORDER BY sales DESC) AS
row_num
FROM
  products_table;
```

#4. PERCENT_RANK()

This function utilizes the ***RANK*** function to define the final ranking. It aims to outline a relative ranking for rows in the result set, expressing it as a percentage, such that the result is between 0 and 1. The 0 value is allocated to the first row, and 1 takes the last. The syntax is...

```
PERCENT_RANK( ) OVER (PARTITION BY partition_expression ORDER BY
sort_expression)
```

Here's a query where we assign a rank to each employee based on their sales performance.

```
SELECT
  customer_id,
  revenue,
  PERCENT_RANK( ) OVER (ORDER BY revenue DESC) AS revenue_percent_rank
FROM
  revenue_table;
```

(buckets) instead of numbering rows. The number of buckets is passed as an argument to the ***N-TILE*** function. For example, ***N-TILE(10)*** divide the dataset into 10 buckets. Syntax is...

```
NTILE(number_of_buckets) OVER (PARTITION BY partition_expression ORDER
BY sort_expression)
```

The example below divides employees into quartiles, as described in the function.

```
SELECT
  employee_id,
  salary,
  NTILE(4) OVER (ORDER BY salary) AS salary_quartile
FROM
  employee_salary_table;
```

#6. CUME_DIST

As the last member, it calculates the cumulative distribution of a value within a sorted data set. It assigns a value between 0 and 1 to represent the relative positions of rows. 0 At the start and 1 at the end. Syntax:

```
CUME_DIST( ) OVER (PARTITION BY partition_expression ORDER BY
sort_expression)
```

For the example below, the query evaluates the distribution of customers based on age, with a percentage of customers – either less than or equal to the current row.

```
SELECT
  customer_id,
  age,
  CUME_DIST( ) OVER (ORDER BY age) AS age_cume_dist
FROM
  customer_data;
```

A quick reference sheet for ranking window functions is available below.

RANK()	Assigns ranking values based on specified order.	**`RANK() OVER (ORDER BY sales DESC) AS sales_rank`** – Assigns a rank to each employee based on sales performance.
DENSE_RANK()	Similar to RANK(), but handles ties without gaps.	**`DENSE_RANK() OVER (ORDER BY test_score DESC) AS score_rank`** – Allocates a dense rank to students based on test scores.
ROW_NUMBER()	Assigns unique row numbers following ORDER BY clause.	**`ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) AS row_num`** – Assigns row numbers to products based on sales within each category.
PERCENT_RANK()	Defines relative ranking as a percentage (0 to 1).	**`PERCENT_RANK() OVER (ORDER BY revenue DESC) AS revenue_percent_rank`** – Assigns a percentage rank to customers based on revenue.
NTILE(number_of_buckets)	Divides rows into specified buckets	**`NTILE(4) OVER (ORDER BY salary) AS salary_quartile`** – Divides employees into quartiles based on salary.
CUME_DIST()	Calculates cumulative distribution (0 to 1) of values.	**`CUME_DIST() OVER (ORDER BY age) AS age_cume_dist`** – Evaluates the cumulative distribution of customers based on age, represented as a percentage.

Conclusion

In this article, we introduced SQL Window functions, a fine technique to walk through your SQL queries, simplifying your data operations. We have looked at the available models for Window functions, showcased their syntax, previewed their examples, explained their use, and concluded with an example cheat sheet you can adopt in your data workflow.

This means working with the right one, whether on SQL or PostgreSQL.

You can now check our full SQL cheat sheet that my colleague and I use throughout our daily developer endeavors.

John Walter
Contributor in

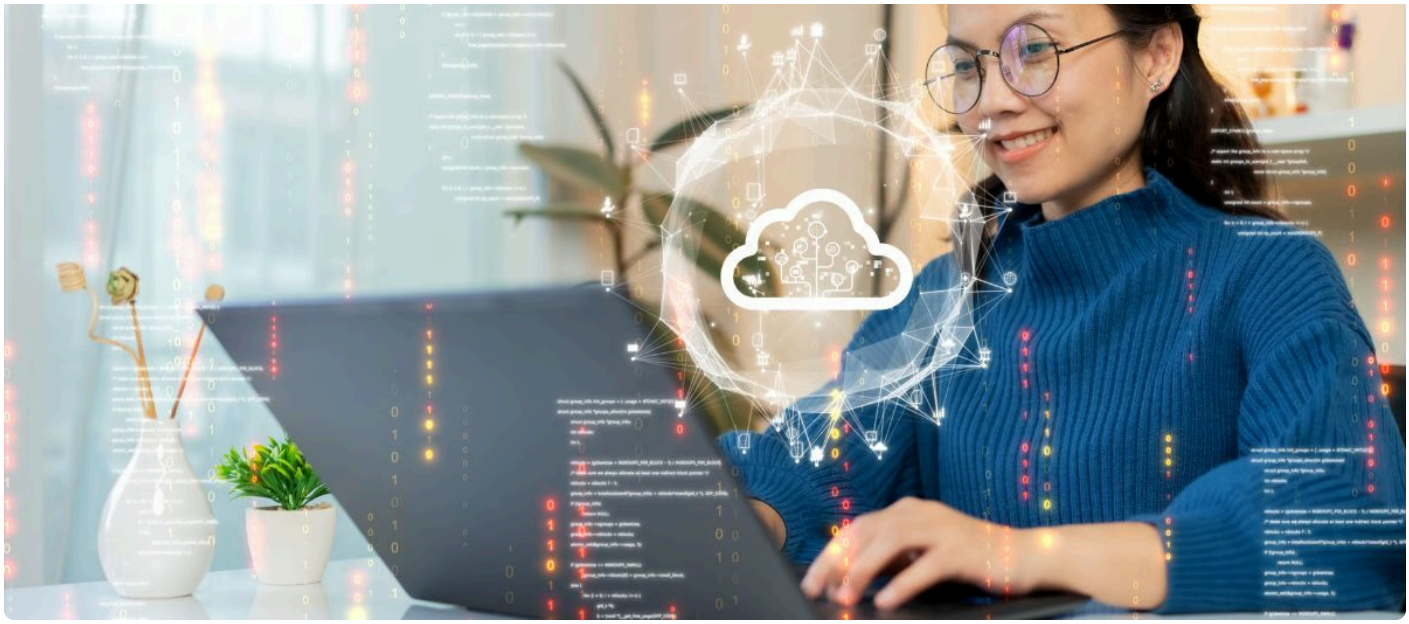
John Walter is an electrical and electronics engineer with a deep passion for software development and blockchain technology. He enjoys learning about new technologies and educating the online community about them. He is also a classical organist.

Rashmi Sharma
Editor in

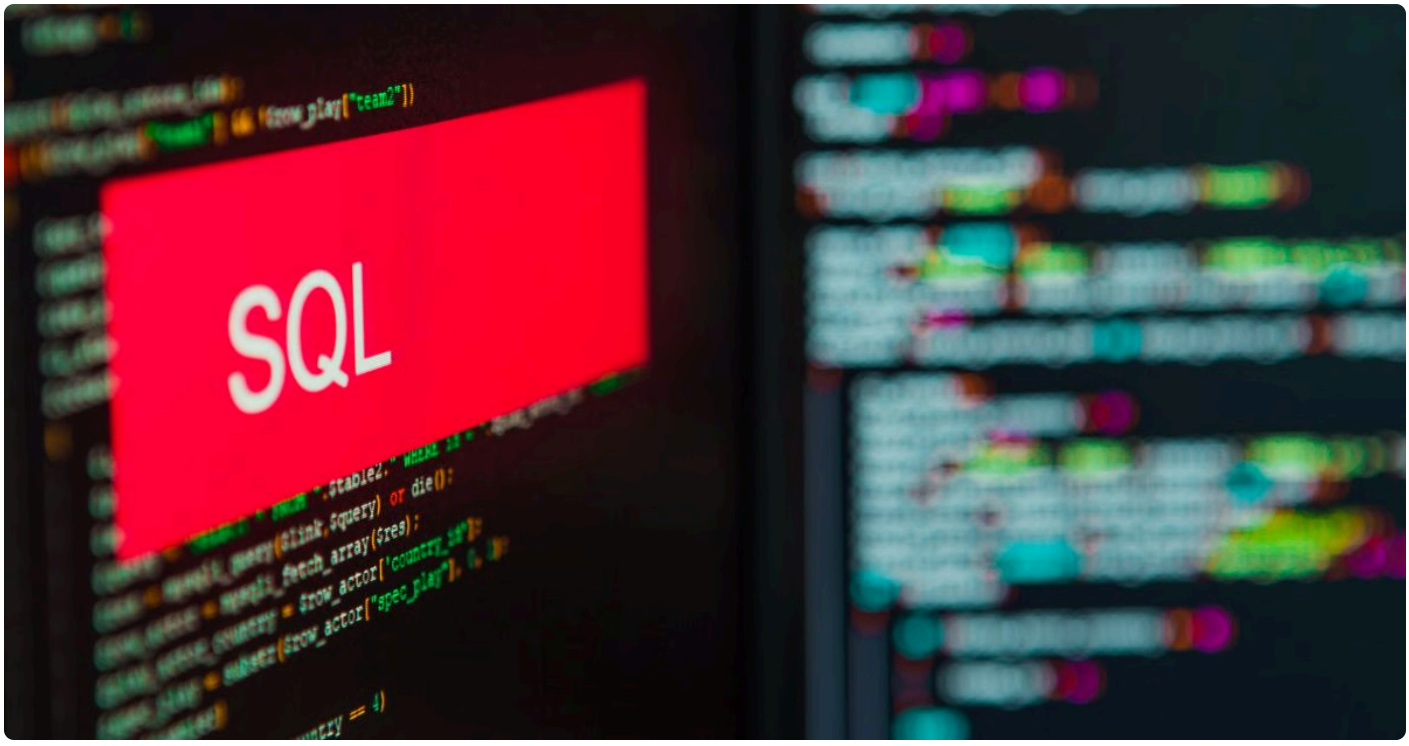
Rashmi Sharma is an editor at Geekflare. She is passionate about researching business resources and has an interest in data analysis.

More on Database

Data Normalization: All You Need to Know

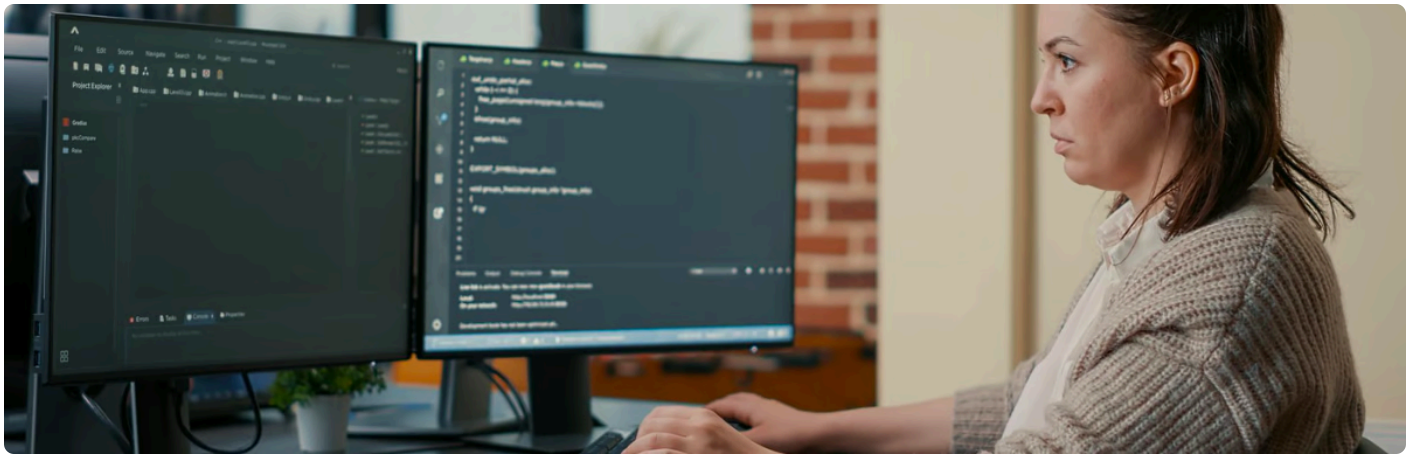


8 Best Database Migration Tools for Smooth Data Transfer



80+ Frequently Asked SQL Interview Questions and Answers [2024]

Snowflake vs. Redshift: How to Choose the Right Data Warehouse



Firebase vs. MongoDB: Which Database to Choose?

Understanding COALESCE () Function in SQL

GEEKFLARE

About Us

Advertise

Contact

Careers

LEGAL

Terms

Privacy Policy

Cookie Policy

GENERAL

Editorial Policies

Sitemap

Find and compare business software insights to increase efficiency, streamline operations, enhance collaboration, reduce costs, and grow your business.

English