**An Intro to SQL Window Functions**

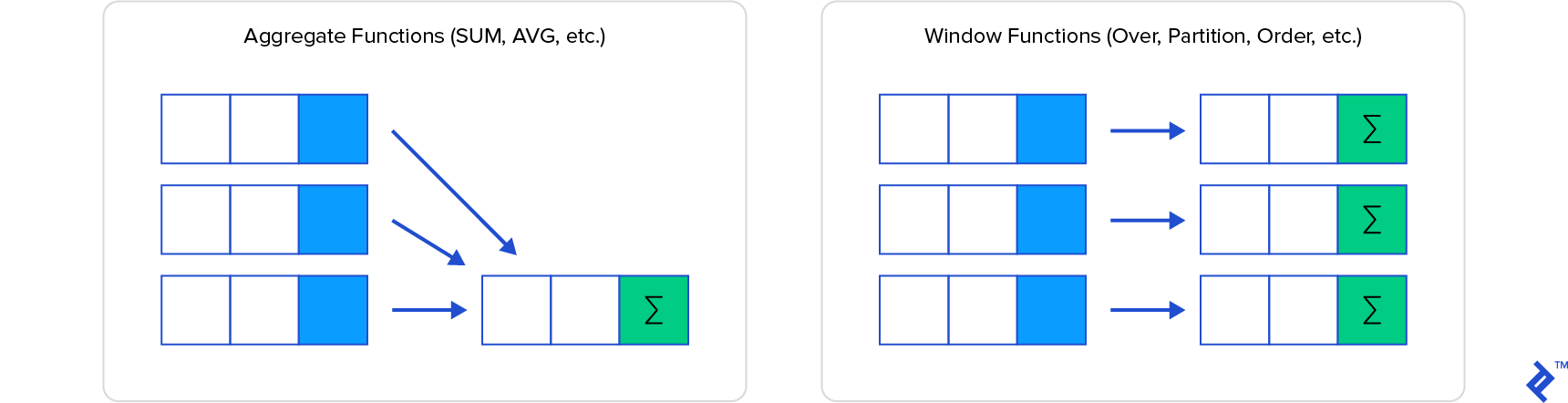
[Neal Barnett](https://www.toptal.com/resume/neal-barnett)

FROM: <https://www.toptal.com/sql/intro-to-sql-windows-functions>

**The very powerful feature you love to hate (but need to know)**

SQL window functions provide some extremely powerful and useful features. But for many, since they are so foreign to standard SQL, they are difficult to learn and understand, have strange syntax—and are very often avoided.

Window functions can be simply explained as calculation functions similar to aggregating, but where normal aggregating via the GROUP BY clause combines then hides the individual rows being aggregated, window functions have access to individual rows and can add some of the attributes from those rows into the result set.



In this SQL window functions tutorial, I’m going to get you started with window functions, explain the benefits and when you’d use them, and give you real examples to help with the concepts.

**A Window into Your Data**

One of the most used and important features in SQL is the ability to aggregate or group rows of data in particular ways. In some cases, however, grouping can become extremely complex, depending on what is required.

Have you ever wanted to loop through the results of your query to get a ranking, a top x list, or similar? Have you had any analytics projects where you wanted to prepare your data just right for a visualization tool, but found it nearly impossible or so complex that it wasn’t worth it?

Window functions can make things easier. After you get the result of your query—i.e., after the WHERE clause and any standard aggregation, window functions will act on the remaining rows (the *window* of data) and get you what you want.

Some of the window functions we’re going to look at include:

* OVER
* COUNT()
* SUM()
* ROW\_NUMBER()
* RANK()
* DENSE\_RANK()
* LEAD()
* LAG()

**Over Easy**

The OVER clause is what specifies a window function and must always be included in the statement. The default in an OVER clause is the entire rowset. As an example, let’s look at an employee table in a company database and show the total number of employees on each row, along with each employee’s info, including when they started with the company.

**SELECT** **COUNT**(\*) **OVER**() **As** NumEmployees, firstname, lastname, date\_started

**FROM** Employee

**ORDER** **BY** date\_started;

|  |  |  |  |
| --- | --- | --- | --- |
| **NumEmployees** | **firstname** | **lastname** | **date\_started** |
| 3 | John | Smith | 2019-01-01 00:00:00.000 |
| 3 | Sally | Jones | 2019-02-15 00:00:00.000 |
| 3 | Sam | Gordon | 2019-02-18 00:00:00.000 |

The above, like many window functions, can also be written in a more familiar non-windowed way—which, in this simple example, isn’t too bad:

**SELECT**

(**SELECT** **COUNT**(\*) **FROM** Employee) **as** NumEmployees, firstname, lastname, date\_started

**FROM** Employee

**ORDER** **BY** date\_started;

But now, let’s say we wish to show the number of employees who started in the same month as the employee in the row. We will need to narrow or restrict the count to just that month for each row. How is that done? We use the window PARTITION clause, like so:

**SELECT** **COUNT**(\*) **OVER** (**PARTITION** **BY** **MONTH**(date\_started),**YEAR**(date\_started))

**As** NumPerMonth,

**DATENAME**(**month**,date\_started)+' '+**DATENAME**(**year**,date\_started) **As** TheMonth,

firstname, lastname

**FROM** Employee

**ORDER** **BY** date\_started;

|  |  |  |  |
| --- | --- | --- | --- |
| NumPerMonth | TheMonth | Firstname | Lastname |
| 1 | January 2019 | John | Smith |
| 2 | February 2019 | Sally | Jones |
| 2 | February 2019 | Sam | Gordon |

Partitions allow you to filter the window into sections by a certain value or values. Each section is often called the window *frame*.

To take it further, let’s say we not only wanted to find out how many employees started in the same month, but we want to show in which order they started that month. For that, we can use the familiar ORDER BY clause. However, within a window function, ORDER BY acts a bit differently than it does at the end of a query.

**SELECT** **COUNT**(\*) **OVER** (**PARTITION** **BY** **MONTH**(date\_started), **YEAR**(date\_started)

**ORDER** **BY** date\_started) **As** NumThisMonth,

**DATENAME**(**month**,date\_started)+' '+**DATENAME**(**year**,date\_started) **As** TheMonth,

firstname, lastname, date\_started

**FROM** Employee

**ORDER** **BY** date\_started;

|  |  |  |  |
| --- | --- | --- | --- |
| NumThisMonth | TheMonth | Firstname | lastname |
| 1 | January 2019 | John | Smith |
| 1 | February 2019 | Sally | Jones |
| 2 | February 2019 | Sam | Gordon |

In this case, ORDER BY modifies the window so that it goes from the start of the partition (in this case the month and year of when the employee started) to the current row. Thus, the count restarts at each partition.

**Rank It**

Window functions can be very useful for ranking purposes. Previously we saw that using the COUNT aggregation function enabled us to see in what order Employees joined the company. We could also have used window ranking functions, such as ROW\_NUMBER(), RANK(), and DENSE\_RANK().

The differences can be seen after we add a new employee the following month, and remove the partition:

**SELECT**

ROW\_NUMBER() **OVER** (**ORDER** **BY** **YEAR**(date\_started),**MONTH**(date\_started))

**As** StartingRank,

**RANK**() **OVER** (**ORDER** **BY** **YEAR**(date\_started),**MONTH**(date\_started)) **As** EmployeeRank,

**DENSE\_RANK**() **OVER** (**ORDER** **BY** **YEAR**(date\_started),**MONTH**(date\_started)) **As** DenseRank,

**DATENAME**(**month**,date\_started)+' '+**DATENAME**(**year**,date\_started) **As** TheMonth,

firstname, lastname, date\_started

**FROM** Employee

**ORDER** **BY** date\_started;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| StartingRank | EmployeeRank | DenseRank | TheMonth | firstname | lastname | date\_started |
| 1 | 1 | 1 | January 2019 | John | Smith | 2019-01-01 |
| 2 | 2 | 2 | February 2019 | Sally | Jones | 2019-02-15 |
| 3 | 2 | 2 | February 2019 | Sam | Gordon | 2019-02-18 |
| 4 | 4 | 3 | March 2019 | Julie | Sanchez | 2019-03-19 |

You can see the differences. ROW\_NUMBER() gives a sequential count within a given partition (but with the absence of a partition, it goes through all rows). RANK() gives the rank of each row based on the ORDER BY clause. It shows ties, and then skips the next ranking. DENSE\_RANK also shows ties, but then continues with the next consecutive value as if there were no tie.

Other ranking functions include:

* CUME\_DIST – Calculates the relative rank of the current row within a partition
* NTILE – Divides the rows for each window partition as equally as possible
* PERCENT\_RANK – Percent rank of the current row

Notice also in this example that you can have multiple Window functions in a single query—and both the partition and order can be different in each!

**Rows and Ranges and Frames, Oh My**

To further define or limit your window frame within the OVER() clause, you can use ROWS and RANGE. With the ROWS clause, you can specify the rows included in your partition as those previous to or after the current row.

**SELECT** OrderYear, OrderMonth, TotalDue,

**SUM**(TotalDue) **OVER**(**ORDER** **BY** OrderYear, OrderMonth **ROWS** **BETWEEN**

**UNBOUNDED** **PRECEDING** **AND** 1 **PRECEDING**) **AS** 'LaggingRunningTotal'

**FROM** sales\_products;

In this example, the window frame goes from the first row to the current row minus 1, and the window size continues to increase for each row.

Range works a bit different and we may get a different result.

**SELECT** OrderYear, OrderMonth, TotalDue,

**SUM**(TotalDue) **OVER**(**ORDER** **BY** OrderYear, OrderMonth **RANGE** **BETWEEN**

**UNBOUNDED** **PRECEDING** **AND** 1 **PRECEDING**) **AS** 'LaggingRunningTotal'

**FROM** sales\_products;

Range will include those rows in the window frame which have the same ORDER BY values as the current row. Thus, it’s possible that you can get duplicates with RANGE if the ORDER BY is not unique.

Some describe ROWS as a physical operator while RANGE is a logical operator. In any case, the default values for ROWS and RANGE are always UNBOUNDED PRECEDING AND CURRENT ROW.

**What Else?**

Most standard aggregate functions work with Window functions. We’ve seen COUNT in the examples already. Others include SUM, AVG, MIN, MAX, etc.

With window functions, you can also access both previous records and subsequent records using LAG and LEAD, and FIRST\_VALUE and LAST\_VALUE. For example, let’s say you want to show on each row a sales figure for the current month, and the difference between last month’s sale figure. You might do something like this:

**SELECT** **id**, OrderMonth, OrderYear, product, sales,

sales - LAG(sales,1) **OVER** (**PARTITION** **BY** product **ORDER** **BY** OrderYear, OrderMonth) **As** sales\_change

**FROM** sales\_products

**WHERE** sale\_year = 2019;

**Basically, SQL Window Functions Are Very Powerful**

While this is a quick introduction to SQL window functions, hopefully it will spark your interest to see all that they can do. We learned that window functions perform calculations similar to how aggregation functions do, but with the added benefit that they have access to data within the individual rows, which makes them quite powerful. They always contain the OVER clause, and may contain PARTITION BY, ORDER BY, and a host of aggregating (SUM, COUNT, etc.) and other positional functions (LEAD, LAG). We also learned about window frames and how they encapsulate sections of data.

Note that different flavors of SQL may implement window functions differently, and some may not have implemented all window functions or clauses. Make sure to check the documentation for the platform you’re using.

If, as a [SQL developer](https://www.toptal.com/sql-developer), you’re interested in tuning your SQL database performance check out [*SQL Database Performance Tuning for Developers*](https://www.toptal.com/sql-server/sql-database-tuning-for-developers).

Happy windowing!

For more information on specific implementations, see:

* PostgreSQL’s [Window Functions](https://www.postgresql.org/docs/11/functions-window.html) documentation for a PostgreSQL implementation.
* [*SELECT - OVER Clause (Transact-SQL)*](https://docs.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql?view=sql-server-2017) docs by Microsoft.
* [*Window Functions in SQL Server*](https://www.red-gate.com/simple-talk/sql/learn-sql-server/window-functions-in-sql-server/) for a great overview on SQL Server implementations, and its [part 2](https://www.red-gate.com/simple-talk/sql/learn-sql-server/window-functions-in-sql-server-part-2-the-frame/).

**Understanding the basics**

**What does a window function in SQL do?**

A window function performs calculations over a set of rows, and uses information within the individual rows when required.

**How is a window aggregate function different from a "group by" aggregate function?**

With "group by," you can only aggregate columns not in the "group by" clause. Window functions allow you to gather both aggregate and non-aggregate values at once.

**Can you use more than one window function in a single SELECT statement?**

Yes, and this is a great advantage, since the window "frames" in each can be based on different filters.

**Can I access previous data with window functions?**

Yes, you can access both previous and future rows, using the LAG and LEAD functions.

**Can I generate running totals with window functions?**

Yes, you can add the ORDER BY clause to produce running