Suffyan Asad Follow
Jan 15 · 8 min read · Listen



# Window Functions in SQL: A Comprehensive Guide for Beginners

## Introduction

This is an introductory article covering the concept of "Window" over data in a relational database, the operations that can be performed within the window bounds, and situations where window functions can be helpful, with examples. The objective is to introduce the concepts of window functions and their usage visually using diagrams for better understanding. The article is aimed at the users of relational databases and SQL who are not familiar with the concept.

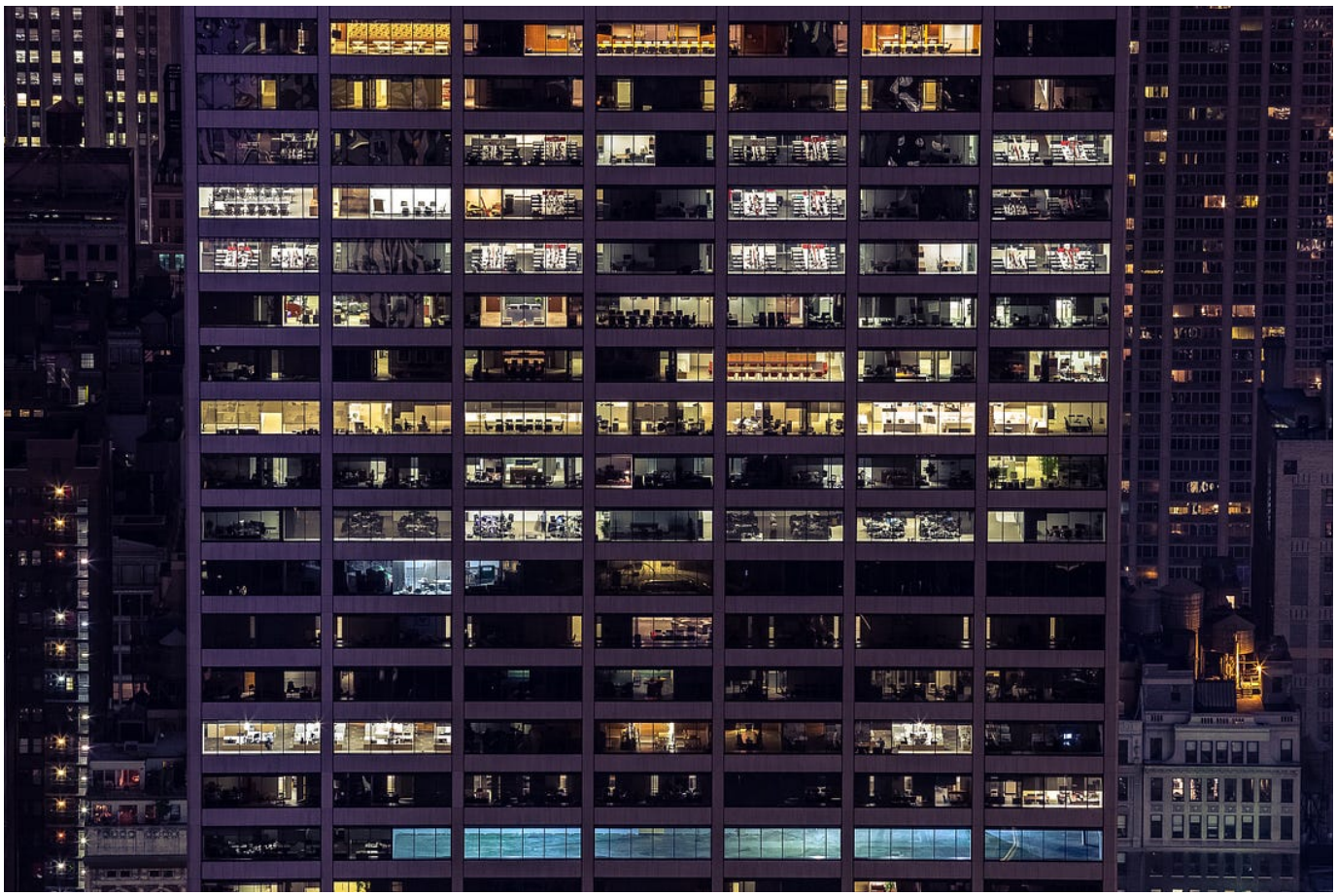


Photo by Vladimir Kudinov on Unsplash

### Windows and Window Functions

A Window is a set (or group) of rows that are related to each-other based on some conditions. And window functions are functions that run on each row, and operate on each window that a row belongs to. The official

PostgreSQL documentation defines window functions as:

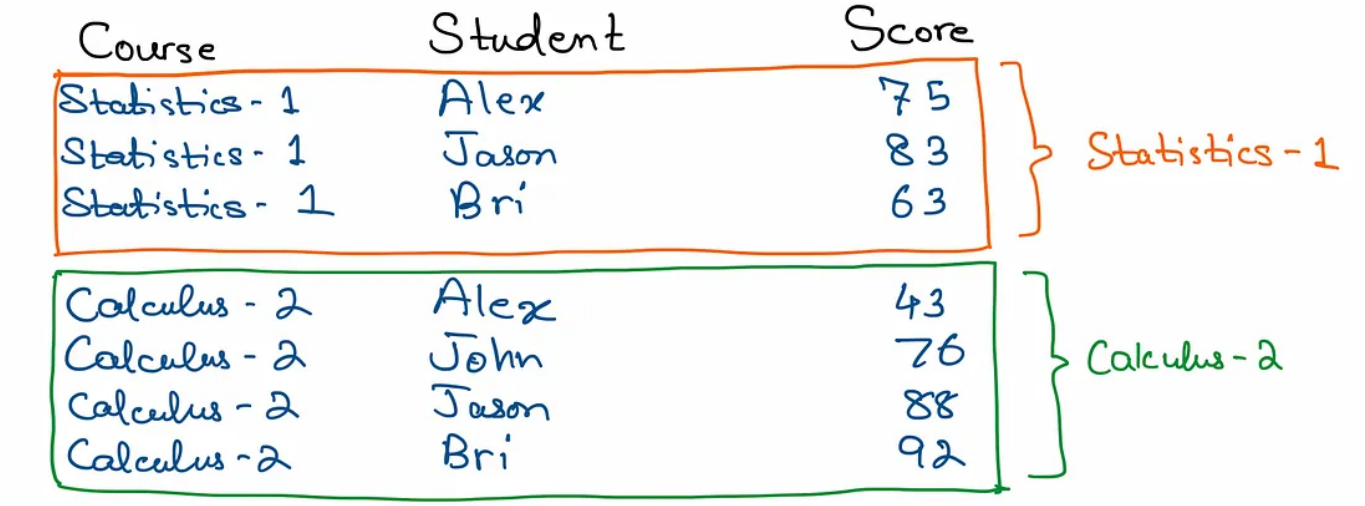
> A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. However, window functions do not cause rows to become grouped into a single output row like non-window aggregate calls would. Instead, the rows retain their separate identities.

First, lets look into defining "windows" over data.

### Defining windows

A "window" is a group of rows that are related to each other.

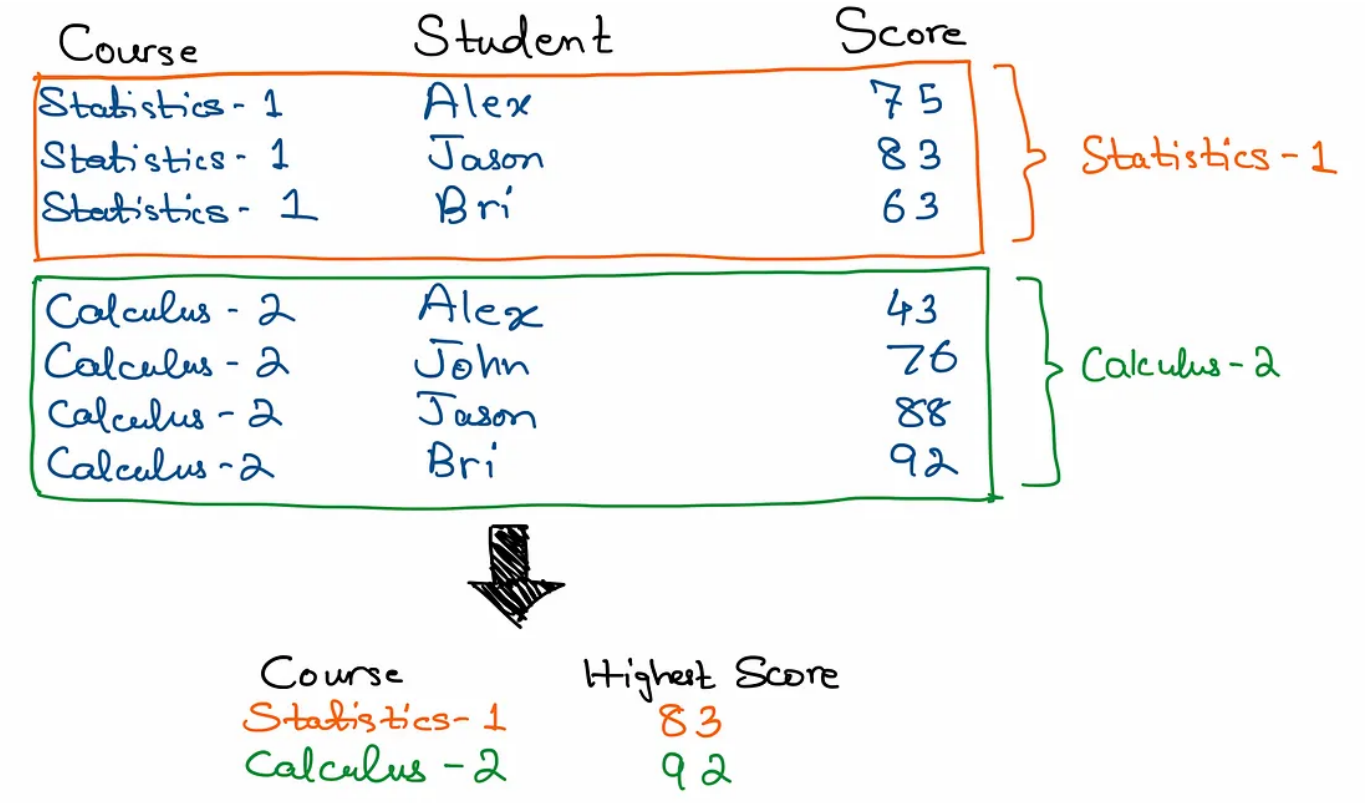
For example, consider data containing Courses, Students, and their Score in the final exam of each course:




Data containing exam scores

And some aggregate function that needs to be performed over a group i.e. highest score in each course. The requirement is to write a query that returns the highest score in each course. To add some complexity, the result should also contain the difference of each student's score from the highest.

In order to achieve this, the each course can be turned into a group using the `GROUP BY` operator, and then the `MAX` aggregate function can be used to calculate the maximum score in each course:



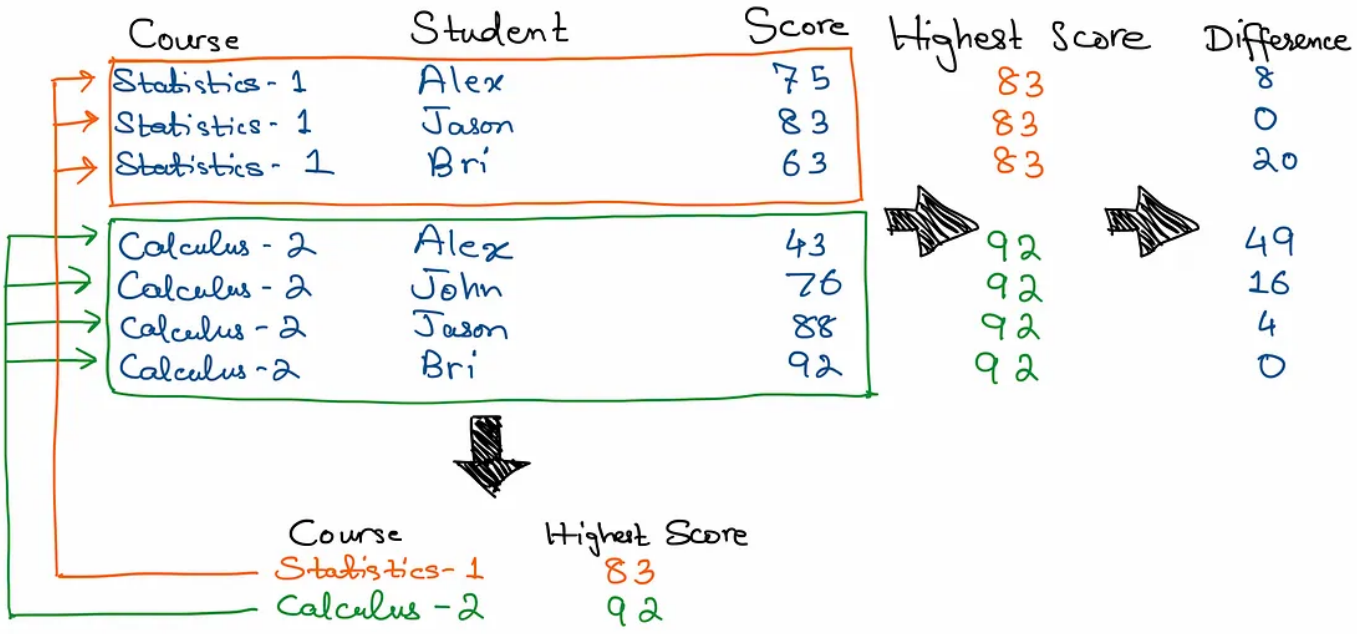

Grouping by Course to get highest score in each course

```sql
SELECT Course, MAX(Score) AS Highest_Score
FROM exam_scores
GROUP BY Course
```

window_1.sql hosted with ❤ by GitHub view raw

Query with Group By

Next, we need to join the highest score with the original exam_scores table. This will attach the highest score with each row matched by the join operation. Then, the difference can be calculated by subtraction:



Steps to create the required result using group by -> join -> subtraction

The SQL query is:

```sql
SELECT e.Course, e.Student, e.Score, h.Highest_Score, h.Highest_Score - e.Score AS Difference
FROM exam_scores e
JOIN (
        SELECT Course, MAX(Score) AS Highest_Score
        FROM exam_scores
        GROUP BY Course
) h
ON e.Course = h.Course
```

**window_2.sql** hosted with ❤ by **GitHub** **view raw**

Final query

The above result can also be obtained by using the Window functions, because it is another way to define groups. The difference is that instead of collapsing the data to one-row-per-group like the group-by clause, window functions attach their result with each row. Because, as mentioned earlier:

> However, window functions do not cause rows to become grouped into a single output row like non-window aggregate calls would. Instead, the rows retain their separate identities.

Link: https://www.postgresql.org/docs/current/tutorial-window.html

Because each course is a group, the window bounds are defined by using the `Course` column:

```sql
SELECT Course, Student, Score, Highest_Score, (Highest_Score - Score) AS diff
FROM (
        SELECT Course, Student, Score, MAX(Score) OVER (PARTITION BY Course) AS Highest_Score
        FROM windowtutorial.exam_score
) t;
```

**window_3.sql** hosted with ❤ by **GitHub** **view raw**

Difference from highest score in each course query implementation using Window functions

And the result of the query is:

Ctrl+click to open SQL console

| | | 123 Score | 123 Highest_Score | 123 diff |
|---|---|---|---|---|
| Calculus 2 | Alex | 43 | 92 | 49 |
| Calculus 2 | John | 76 | 92 | 16 |
| Calculus 2 | Jason | 89 | 92 | 3 |
| Calculus 2 | Bri | 92 | 92 | 0 |
| Statistics 1 | Alex | 75 | 83 | 8 |
| Statistics 1 | Jason | 83 | 83 | 0 |
| Statistics 1 | Bri | 63 | 83 | 20 |

Query result

**Note:** The final select clause that computes the difference wraps the inner query that applies the window function. This is because the column Highest_Score is not yet defined inner select clause and it cannot be used in the same clause. If the inner query is executed, we get the following result:

```
SELECT Course, Student, Score, MAX(Score) OVER (PARTITION BY Course) AS Highest_Score
FROM windowtutorial.exam_score
```

window_4.sql hosted with ❤ by GitHub view raw

Inner query

| | | | 123 Score | 123 Highest_Score |
|---|---|---|---|---|
| 1 | Calculus 2 | Alex | 43 | 92 |
| 2 | Calculus 2 | John | 76 | 92 |
| 3 | Calculus 2 | Jason | 89 | 92 |
| 4 | Calculus 2 | Bri | 92 | 92 |
| 5 | Statistics 1 | Alex | 75 | 83 |
| 6 | Statistics 1 | Jason | 83 | 83 |
| 7 | Statistics 1 | Bri | 63 | 83 |

Result of inner query: highest score column added

**Window clause syntax**

Lets inspect the window clause. In the aforementioned query, the window definition is the following part of the select statement:

```
(PARTITION BY Course)
```

window_5.sql hosted with ❤ by GitHub view raw

partition by clause

The boundary of each window is all the rows containing the same unique value of the course column. And the `MAX` aggregate function is calculating the maximum value for each window. If more than one column is to be used in the partition clause, it can be separated by a comma.

Following is the full syntax of the window definition clause in PostgreSQL:

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

window_6.sql hosted with ❤ by GitHub view raw

Window definition syntax in PostgreSQL

**Note:** The above clause is for PostgreSQL, but generally, the main constituents of a window definition are almost the same in other popular RDBMS systems. The differences, if present, are generally not very significant, and the concepts are also generic.

Following are the constructs of the window:

- Existing window name
- `PARTITION BY` clause is used to define the grouping columns
- `ORDER BY` clause is used to define the ordering column(s). This clause is required in some cases that we will see later.
- Frame clause

In the previous example, only the `PARTITION BY` clause was used to define the window.

#### Defining and re-using window in the `window` clause

In the previous example, the window definition was given after the `OVER` clause in parentheses immediately after the `MAX` window function. This way of defining windows will require repeating the definition if more than one window functions are used in a query. For example, if average, maximum and minimum are needed, the previous query will look like:

```sql
SELECT  Course, Student, Score,
                MAX(Score) OVER (PARTITION BY Course) AS Highest_Score,
                MIN(Score) OVER (PARTITION BY Course) AS Lowest_Score,
                AVG(Score) OVER (PARTITION BY Course) AS Avg_Score
FROM windowtutorial.exam_score;
```

window_7.sql hosted with ❤ by GitHub view raw

Window definition repeated with each window function in the select clause

| | course | student | score | highest_score | lowest_score | avg_score |
|---|---|---|---|---|---|---|
| 1 | Calculus 2 | Alex | 43 | 92 | 43 | 74.75 |
| 2 | Calculus 2 | John | 76 | 92 | 43 | 74.75 |
| 3 | Calculus 2 | Bri | 92 | 92 | 43 | 74.75 |
| 4 | Calculus 2 | Jason | 88 | 92 | 43 | 74.75 |
| 5 | Statistics 1 | Jason | 83 | 83 | 63 | 73.6666666667 |
| 6 | Statistics 1 | Bri | 63 | 83 | 63 | 73.6666666667 |
| 7 | Statistics 1 | Alex | 75 | 83 | 63 | 73.6666666667 |

Query result

Alternatively, the window can be defined once, and re-used in multiple aggregate functions:

```sql
SELECT  Course, Student, Score,
                MAX(Score) OVER w_course AS Highest_Score,
                MIN(Score) OVER w_course AS Lowest_Score,
                AVG(Score) OVER w_course AS Avg_Score
FROM windowtutorial.exam_score
WINDOW w_course AS (PARTITION BY Course);
```

window_8.sql hosted with ❤ by GitHub view raw

Defining window once example

Here, the window `w_course` has been defined once in the `WINDOW` clause of the SQL query, and referred to by its

name in the select statement. Similarly, more than once windows can be defined, and used. For example, the following query defined two windows:

```sql
SELECT  Course, Student, Score,
                MAX(Score) OVER w_course AS Highest_Score,
                MIN(Score) OVER w_course AS Lowest_Score,
                AVG(Score) OVER w_course AS Avg_Score,
                COUNT(course) OVER w_student as Courses
FROM windowtutorial.exam_score
WINDOW  w_course AS (PARTITION BY Course),
        w_student AS (PARTITION BY Student);
```

window_9.sql hosted with ❤ by GitHub view raw

Defining multiple windows in the WINDOW clause of the query

And it gives the following result:

Ctrl+click to open SQL console

| | | 123 score | 123 highest_score | 123 lowest_score | 123 avg_score | 123 courses |
|---|---|---|---|---|---|---|
| Calculus 2 | Bri | 92 | 92 | 43 | 74.75 | 2 |
| Calculus 2 | Alex | 43 | 92 | 43 | 74.75 | 2 |
| Calculus 2 | Jason | 88 | 92 | 43 | 74.75 | 2 |
| Calculus 2 | John | 76 | 92 | 43 | 74.75 | 1 |
| Statistics 1 | Alex | 75 | 83 | 63 | 73.6666666667 | 2 |
| Statistics 1 | Bri | 63 | 83 | 63 | 73.6666666667 | 2 |
| Statistics 1 | Jason | 83 | 83 | 63 | 73.6666666667 | 2 |

Query result

## Ranks, Running totals, ordered windows

Windows can be defined in a way that they are different for every row. One example is a window for each row ordered by time to compute a running aggregate, such as a running total. In this case, the window will have to be ordered by a timestamp (or date column). If no bounds are specified, the window may span the entire table.

### Ordered window

Consider the same exam scores in courses table used in the previous examples. If a query needs to return the rank of scores in each course, it will not be possible without ordering the data within each window, because the database will not know what order to consider when running functions such as `RANK` or `ROW_NUMBER`.

The following query uses two windows, one per course, and one overall, to calculate per course, and overall ranks of test scores:

```sql
SELECT  Course, Student, Score,
        RANK() OVER w_course AS course_rank,
        RANK() OVER w_overall AS overall_rank
FROM windowtutorial.exam_score
WINDOW  w_course AS (PARTITION BY Course ORDER BY score DESC),
        w_overall AS (ORDER BY score DESC)
ORDER BY course, course_rank;
```

window_10.sql hosted with ❤ by GitHub view raw

Window function Rank example

And it produces the following result:

| | course | student | score | course_rank | overall_rank |
|---|---|---|---|---|---|
| 1 | Calculus 2 | Bri | 92 | 1 | 1 |
| 2 | Calculus 2 | Jason | 88 | 2 | 2 |
| 3 | Calculus 2 | John | 76 | 3 | 4 |
| 4 | Calculus 2 | Alex | 43 | 4 | 7 |
| 5 | Statistics 1 | Jason | 83 | 1 | 3 |
| 6 | Statistics 1 | Alex | 75 | 2 | 5 |
| 7 | Statistics 1 | Bri | 63 | 3 | 6 |

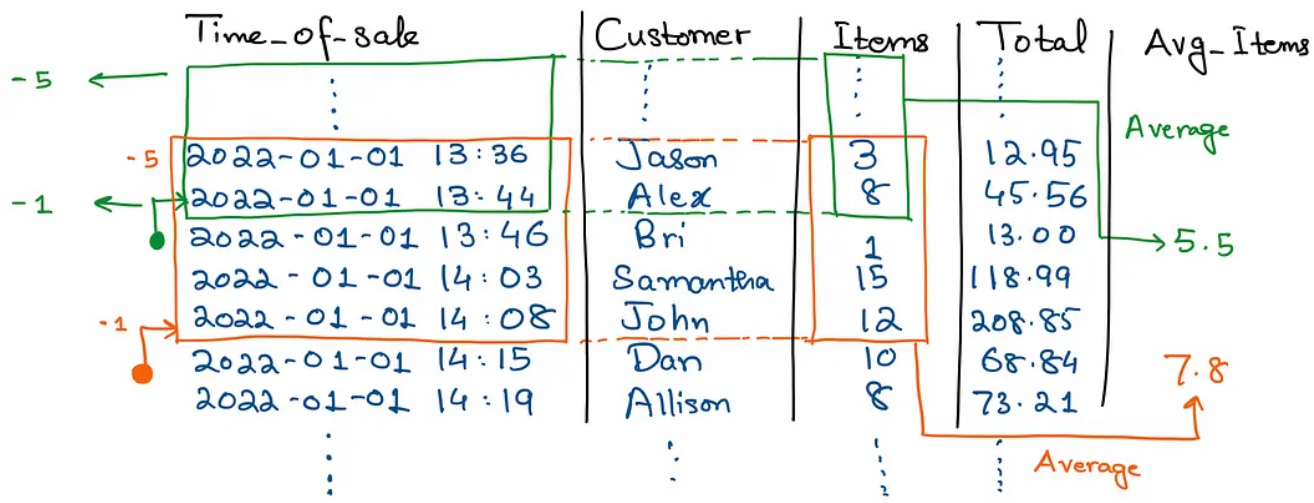
Query result

In this example, the window without the `PARTITION BY` clause will include all the data in the query. The `ORDER BY` clause defines the order in which the `RANK` operator should get applied. Without the ordering, the function may not work as intended for some database systems, or return an error for other systems.

#### Sliding windows — window for rows relative to a given row

Consider an example of a table recording sales at a shop by customers. The data contains number of items in each sale, and the total. The requirement is to create a column with average number of items sold in last 5 sales before any given sale.

To achieve this, a window needs to be defined, which is called a sliding window. A sliding window "slides" with the data i.e. it covers rows relative to each row:




Sliding windows

In the diagram, the sliding window containing previous 5 rows of any given row has been depicted. The window is sliding with each row as it has been defined as relative to each row, ordered by the `Time_of_sale` column. The avergae number of items sold in last 5 sales are being calculated in the last column using the sliding window.

Sliding windows and are defined in the Frame clause of the window definition. For PostgreSQL, the frame clause is defined as:

```
{ RANGE | ROWS | GROUPS } frame_start [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

window_11.sql hosted with ❤ by GitHub view raw

and `frame start` and `frame end` clauses can have the values:

```
UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING
```

window_12.sql hosted with ❤ by GitHub view raw

Using these, the sliding window over the data can be defined as:

```
SELECT  time_of_sale, customer, items, total,
        AVG(items) OVER (ORDER BY time_of_sale ROWS BETWEEN 5 PRECEDING AND 1 PRECEDING) AS avg
FROM windowtutorial.sales;
```

window_13.sql hosted with ❤ by GitHub view raw

And the query gives the following result:

| | time_of_sale | customer | items | total | avg |
|---|---|---|---|---|---|
| 1 | 2022-01-01 13:26:00.000 | Jason | 3 | 12.95 | [NULL] |
| 2 | 2022-01-01 13:44:00.000 | Alex | 8 | 45.56 | 3 |
| 3 | 2022-01-01 13:46:00.000 | Bri | 1 | 13 | 5.5 |
| 4 | 2022-01-01 14:03:00.000 | Samantha | 15 | 118.99 | 4 |
| 5 | 2022-01-01 14:08:00.000 | John | 12 | 208.85 | 6.75 |
| 6 | 2022-01-01 14:15:00.000 | Dan | 10 | 68.84 | 7.8 |
| 7 | 2022-01-01 14:19:00.000 | Allison | 8 | 73.21 | 9.2 |

Here, the window has been defined to encompass 5 rows to 1 row before each row, as given by the `ROWS BETWEEN 5 PRECEDING AND 1 PRECEDING` clause. Remember the following points:

- Not defining the ending bound ends the window at the current row e.g. `ROWS BETWEEN 5 PRECEDING` will create a window including the current row.

- If a window needs to include rows after the current row, `FOLLOWING` can be used in the frame definition, e.g. `ROWS BETWEEN 5 PRECEDING AND 1 FOLLOWING`. And if current row needs to be excluded, the exclusion can be defined as `ROWS BETWEEN 5 PRECEDING AND 1 FOLLOWING EXCLUDE CURRENT ROW`.
- Unbounded windows at one (or both) ends can be defined by using the `UNBLOUNDED PRECEDING` and `UNBOUNDED FOLLOWING` clauses as needed.
- The result in the previous example has `null` in the first row because there are no rows in the sliding window relative to the first row. The window ends at the row before each row, and the one with the null is the first row with no previous rows.

#### Sliding window definition with Range

Instead of `ROWS`, there is an alternative clause called `RANGE`, which can be used to define the bounds as range (added to or subtracted from) the order by column. For example, if the requirement is to create a column containing average number of items sold in last 10 minutes compared to each row, the following window will be needed:

```
1  SELECT  time_of_sale, customer, items, total,
2          AVG(items) OVER (ORDER BY time_of_sale RANGE BETWEEN '10 MINUTES' PRECEDING AND '1 SECOND' PRECEDING) AS avg
3  FROM windowtutorial.sales;
```

**window_14.sql** hosted with ❤ by **GitHub** **view raw**

And this query gives the required result:

| | time_of_sale | customer | items | total | avg |
|---|---|---|---|---|---|
| 1 | 2022-01-01 13:26:00 | Jason | 3 | 12.95 | [NULL] |
| 2 | 2022-01-01 13:44:00 | Alex | 8 | 45.56 | [NULL] |
| 3 | 2022-01-01 13:46:00 | Bri | 1 | 13 | 8 |
| 4 | 2022-01-01 14:03:00 | Samantha | 15 | 118.99 | [NULL] |
| 5 | 2022-01-01 14:08:00 | John | 12 | 208.85 | 15 |
| 6 | 2022-01-01 14:15:00 | Dan | 10 | 68.84 | 12 |
| 7 | 2022-01-01 14:19:00 | Allison | 8 | 73.21 | 10 |

Please note that here, some of the differences between database systems start to matter. The above query will run correctly in PostgreSQL, whereas it will have to be modified as following for MySQL:

```
1  SELECT  time_of_sale, customer, items, total,
2          AVG(items) OVER (ORDER BY time_of_sale RANGE between INTERVAL 10 MINUTE PRECEDING and INTERVAL 1 SECOND PRECEDING
3  FROM windowtutorial.sales;
```

**window_15.sql** hosted with ❤ by **GitHub** **view raw**

## Conclusion

- Windows can be defined in SQL Queries that contain one or more rows, and may or may not include the

current row. Window functions do not collapse the query result to one row per group/window like `GROUP BY` clause. This property allows avoiding the join with a grouped form of the table in many cases.

- Sliding windows that 'slide' with the data are defined in terms of rows compared to a given row: `ROWS BETWEEN .. PRECEDING AND .. PRECEDING/FOLLOWING`. Alternatively, they can be defined in terms of range of values of the `ORDER BY` clause, for example: `RANGE BETWEEN .. PRECEDING AND .. PRECEDING/FOLLOWING`.
- Aggregate functions such as `MAX`, `AVG`, `MIN`, `SUM` etc. can be used to aggregate data in windows.
- Window functions such as `RANK`, `DENSE_RANK`, `ROW_NUMBER`, `LAG`, `LEAD` etc. can be used on data in windows, and they might require specifying the order using the `ORDER BY` clause in window definition.

## References

#### PostgreSQL Documentation about Windows and Window functions

- https://www.postgresql.org/docs/current/tutorial-window.html
- https://www.postgresql.org/docs/current/functions-window.html
- https://www.postgresql.org/docs/current/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS

#### MySQL Documentation about Windows and Window functions

- https://dev.mysql.com/doc/refman/8.0/en/window-functions-frames.html

## Appendix — Creating test data

The queries were tested on PostgreSQL and MySQL databases. Following commands can be used to spin up databases instances in Docker containers, and test data can be created by running the script at the end.

#### Running PostgreSQL Docker image