

# Advanced SQL - window frames

BY MICHAŁ KONARSKI    SEPTEMBER 18, 2019

This article is a part of my series of articles discussing advanced SQL concepts that are supported by popular databases for quite some time, but are not very well known by database users. My idea is to explain them in simple terms, with examples.

What you're reading is a continuation of [my post](#) published almost two years ago describing one of the most powerful features of modern SQL - **window functions**. They allow to perform calculations across a set of related rows, without actually grouping these rows together. In this article, I'm going to focus on an important aspect of window functions that make them even more flexible and useful - **window frames**.

Window frames have been a part of the SQL standard for some time now. All popular database systems [support them to some extent](#), but none of them has all features implemented. PostgreSQL is currently [the leader in this field](#). Its latest version 11 introduced most of the window frames related features described by the standard. Therefore I'll be using it throughout this article.

## Window functions - a quick recap

Let's start with a quick reminder about what window functions are. Let's say that we're working with the following table:

films		
PK	<u>id</u>	integer
	release_year	integer
	rating	numeric

*Table schema*

id	release_year	rating
1	2015	8.00
2	2015	8.50
3	2015	9.00
4	2016	8.20
5	2016	8.40
6	2017	7.00

*Input rows*

Now we have the following task to solve:

**For each film find an average rating of all films in its release year.**

id	release_year	rating	year_avg
1	2015	8.00	8.50
2	2015	8.50	8.50
3	2015	9.00	8.50
4	2016	8.20	8.30
5	2016	8.40	8.30
6	2017	7.00	7.00

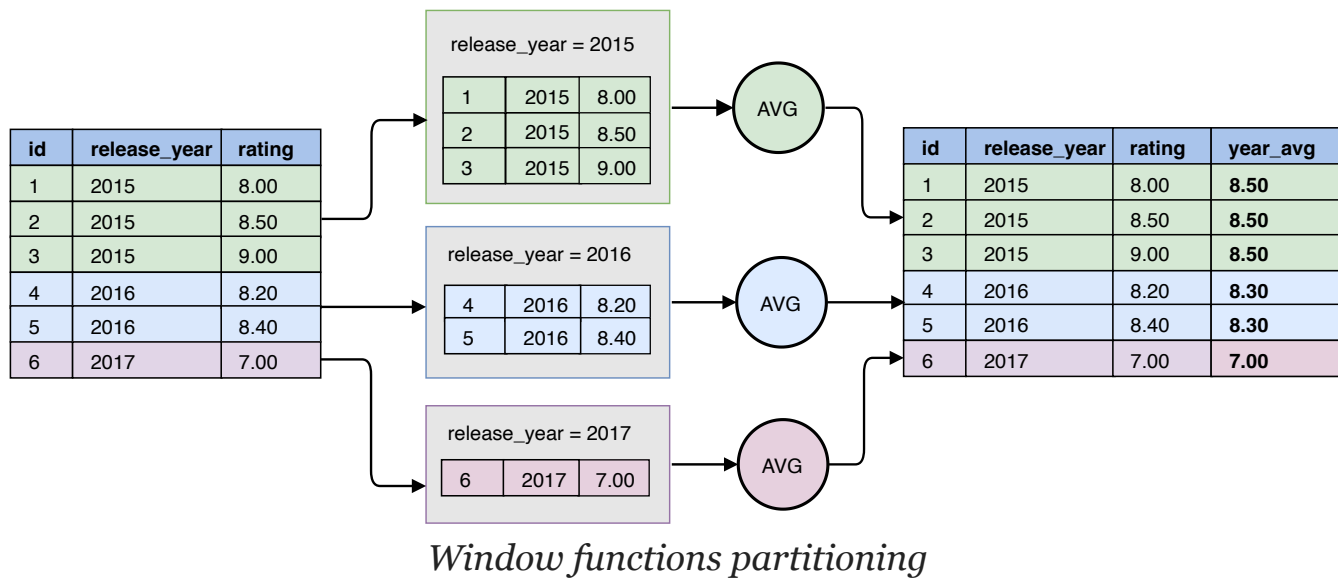
To solve this problem we have to aggregate the input rows. We need to take rows belonging to the same year, then group them and compute an average for every group. This can be easily done with `GROUP BY` statement, but in the result set we'd get just one row for every `release_year`. The output set should contain an additional column, but the same number of rows as the input set. This is a job for a window function:

### SELECT

```
f.id, f.release_year, f.rating,
AVG(rating) OVER (PARTITION BY release_year) AS year_avg
FROM films f ORDER BY release_year, rating;
```

The above code (`PARTITION BY release_year`) instructs the database engine to divide the input rows into disjoint sets called partitions, using the `release_year` column. Each partition receives only the rows that have the same value of `release_year`. Then, by using an aggregate function `AVG(rating)`, we tell the database engine how to

calculate the final result for each partition. This diagram below presents the whole operation:



Now, let's complicate our initial problem a little bit:

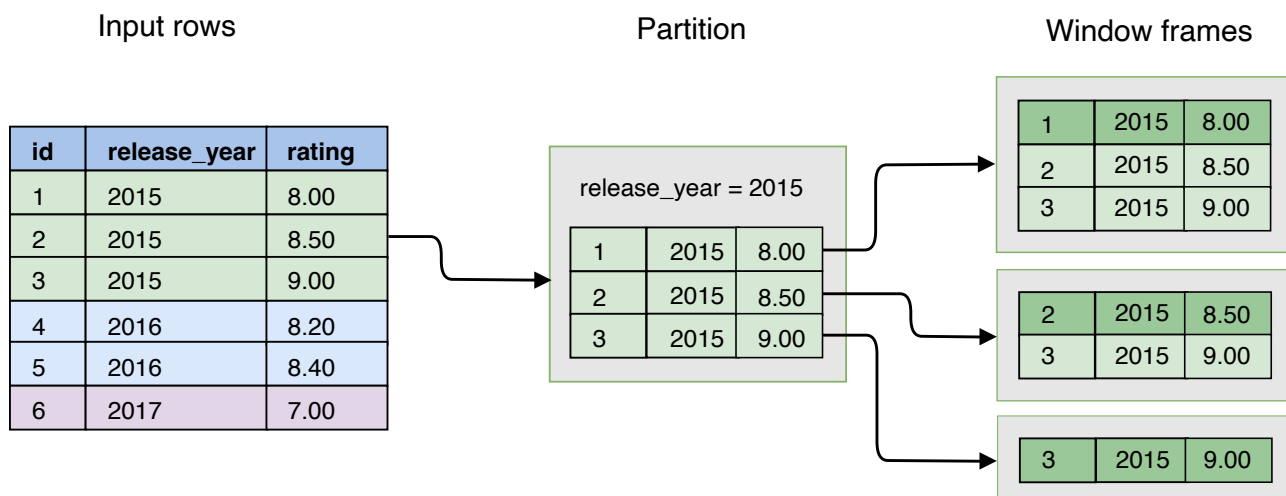
**For each film find an average rating of all strictly better films in its release year.**

id	release_year	rating	avg_of_better
1	2015	8.00	8.75
2	2015	8.50	9.00
3	2015	9.00	[NULL]
4	2016	8.20	8.40
5	2016	8.40	[NULL]
6	2017	7.00	[NULL]

It's clear that now we also need to divide the rows into `release_year` partitions. But the calculation of average needs to be done only on a subset of a partition. The subset needs to be different for every row - we need to consider rows that have a greater value in the `rating` column only. Partitions are exactly the same for every row they contain, so they will not help us achieve this effect. We need something more powerful.

## Window frames

Window frames are a feature that allows us to divide partitions into smaller subsets. What's even more important, these subsets can differ from a row to row. This is something that can't be achieved with partitioning only. For example, we can have window frames that contain all the rows with the same or greater value in a given column:



*Mechanism of creating window frames*

SQL gives us many ways to specify which rows should be included in window frames. In the next paragraphs I will describe all these ways in detail.

## Syntax

A general (and much simplified) format of a window function call is:

```
function_name OVER (PARTITION BY ... ORDER BY ... frame_clause)
```

`frame_clause` is the part that defines window frames. It looks as follows:

```
mode BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

This syntax can be divided into three sections:

- **mode** sets the way a database engine treats input rows. There are three possible values: `ROWS`, `GROUPS` and `RANGE`.
- **frame\_start** and **frame\_end** define where a window frame starts and where it ends.
- **frame\_exclusion** can be used to specify parts of a window frame that have to be excluded from the calculations.

It's important to remember is that window frames are constructed for every single input row separately, so their content can differ from row to row. Therefore it's essential to consider a window frame with regard to the row that that frame is built for. We'll call it **the current row**.

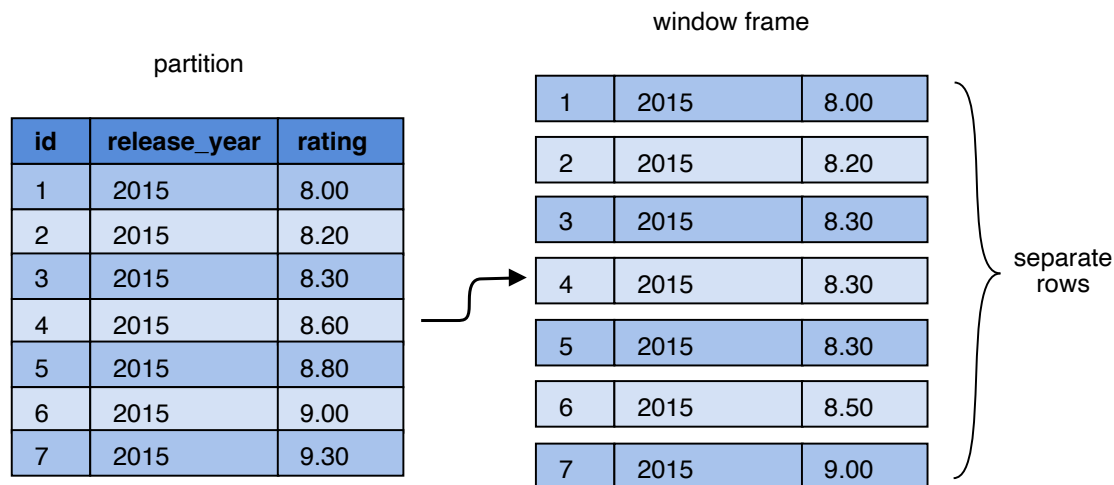
What is also usually crucial is to specify the order in which rows appear in a window frame. In most cases the exact position of the current row compared to other rows will have a direct impact on the content of a frame. Therefore it's always safe to assume that if you want to use window framing you need to have the rows sorted consistently. It can be done by adding an `ORDER BY` clause to a window function call.

All the following examples have the rows sorted by the `rating` column in ascending order. For the sake of simplicity, I also slightly modified the input set - now it contains films released in a single year only.

## Window frame modes

### Rows mode

The `ROWS` mode is the simplest one. It instructs the database to treat each input row separately, as individual entities:



### *ROWS mode*

In the `ROWS` mode ***frame\_start*** and ***frame\_end*** allow us to specify which rows the window frame starts and ends with. They accept the following values:

- `UNBOUNDED PRECEDING` - (possible only in ***frame\_start***) start with the first row of the partition
- `offset PRECEDING` - start/end with a given number of rows before the current row
- `CURRENT ROW` - start/end with the current row
- `offset FOLLOWING` - start/end with a given number of rows after the current row
- `UNBOUNDED FOLLOWING` - (possible only as a ***frame\_end***) end with the last row of the partition

Let's take a look at some examples. Remember that it's crucial to know which row is the current row, because for different rows the window frame can look differently. All the figures below present how the frame looks like for a single, chosen input row.

Let's start with a do-nothing option. It simply selects all rows from the beginning of the partition to the end:

### **ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING**

current row →	1	2015	7.00
	2	2015	7.50
	3	2015	8.00
	4	2015	8.30
	5	2015	8.50
	6	2015	8.80
	7	2015	9.00

Now we'll do something more interesting. In the example below we start with the beginning of the partition, but end with the current row. This is where the order of rows begins to matter:

#### ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

current row →	1	2015	7.00
	2	2015	7.50
	3	2015	8.00
	4	2015	8.30
	5	2015	8.50
	6	2015	8.80
	7	2015	9.00

Here we start with the first row before the current row and end with the first row after the current row:

#### ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

1	2015	7.00	
2	2015	7.50	
3	2015	8.00	← preceding row
4	2015	8.30	
5	2015	8.50	← following row
6	2015	8.80	
7	2015	9.00	

current row →

It's not mandatory to include the current row though. In the example below we start and end before the current row:

### ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING

1	2015	7.00	← preceding row 3
2	2015	7.50	
3	2015	8.00	← preceding row 1
4	2015	8.30	
5	2015	8.50	
6	2015	8.80	
7	2015	9.00	

current row →

We can do more interesting things using the ***frame\_exclusion*** part.

***frame\_exclusion*** allows to exclude some specific rows from the window frame, even if they would be included according to the ***frame\_start*** and ***frame\_end*** options. What's worth mentioning is that ***frame\_exclusion*** works exactly the same regardless of the selected mode. Possible values are:

- `EXCLUDE CURRENT ROW` - exclude the current row.
- `EXCLUDE GROUP` - exclude the current row and all peer rows, i.e rows that have the same value in the sorting column.
- `EXCLUDE TIES` - exclude all peer rows, but not the current row.

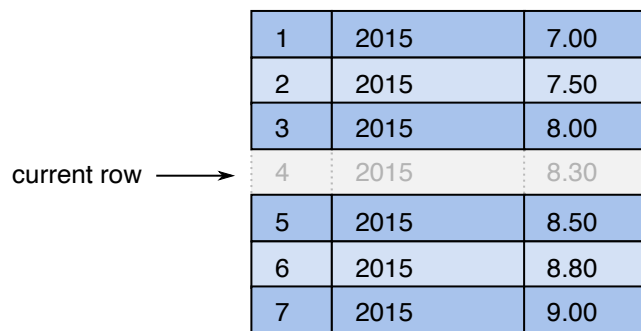


- `EXCLUDE NO OTHERS` - exclude nothing. This is the default option in case you omit the ***frame\_exclusion*** part altogether.

Let me show you an example.

Here we want to select all rows from the beginning of the partition to the end of the partition, but exclude the current row:

### ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING EXCLUDE CURRENT ROW

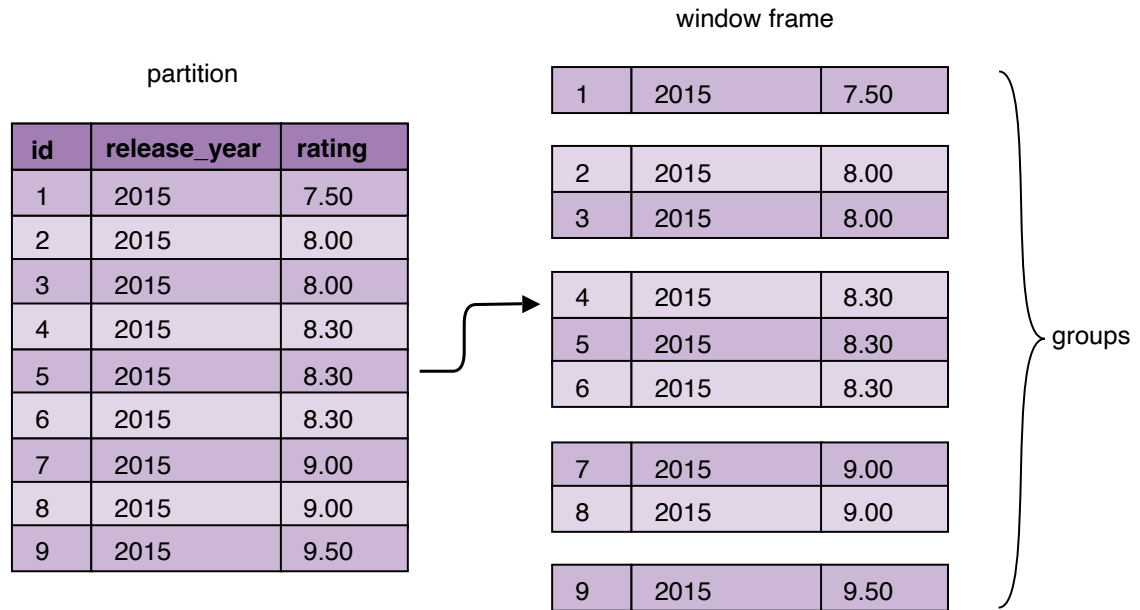


1	2015	7.00
2	2015	7.50
3	2015	8.00
4	2015	8.30
5	2015	8.50
6	2015	8.80
7	2015	9.00

The rest of ***frame\_exclusion*** options become interesting only in the case when the partition has duplicate values in the sorting column. I haven't included them in the examples above on purpose, because there's an important caveat when the `ROWS` mode is used together with sorting duplicates. In that case, the rows with duplicated sorting values are processed in an unspecified order, so their relative positions are not deterministic. This can lead to incorrect results when `offset PRECEDING` and `offset FOLLOWING` clauses are specified. The next section will explain the rest of the ***frame\_exclusion*** options.

## Groups mode

`GROUPS` mode is made exactly for the case when the sorting column contains duplicates. Therefore in this paragraph I'll use a sample of input rows that contains duplicates. In the `GROUPS` mode rows with duplicate sorting values are grouped together:



### GROUPS mode

The syntax looks as follows:

**GROUPS BETWEEN** frame\_start **AND** frame\_end [ frame\_exclusion ]

The **frame\_start** and **frame\_end** parameters accept the same options as in **ROWS** mode, but the meaning of some of them differ:

- **UNBOUNDED PRECEDING** and **UNBOUNDED FOLLOWING** work the same and mean either the first row or the last row of the current partition.
- **offset PRECEDING** and **offset FOLLOWING** now work with regard to groups. You can use them to specify a number of groups before or after the current group to be taken into account.
- **CURRENT ROW** also gets a different meaning, which might seem a bit misleading. When used as **frame\_start** it means the first row in a group containing the current row. When used as **frame\_end** it means the last row in a group containing the current row.

As always it's best to look at some examples.

Let's start with a default option. It simply includes all partition rows:

## GROUPS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

	1	2015	7.50
	2	2015	8.00
	3	2015	8.00
	4	2015	8.30
current row →	5	2015	8.30
	6	2015	8.30
	7	2015	9.00
	8	2015	9.00
	9	2015	9.50

The example below shows the real power of the `GROUPS` mode. We start with the first row in the partition and we want to include everything up to the current group:

## GROUPS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

	1	2015	7.50	
	2	2015	8.00	
	3	2015	8.00	
	4	2015	8.30	
current row →	5	2015	8.30	} current group
	6	2015	8.30	
	7	2015	9.00	
	8	2015	9.00	
	9	2015	9.50	

In the case of any duplicates we can be sure that all of them will be either included in the calculation.

The example below is symmetrical. Here we start with the current group and end at the end of the partition. As you can see, the meaning of `CURRENT ROW` changes depending on whether it used to define a beginning or an end of a window frame. Once again all duplicates are included:

## GROUPS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

1	2015	7.50
2	2015	8.00
3	2015	8.00
4	2015	8.30
5	2015	8.30
6	2015	8.30
7	2015	9.00
8	2015	9.00
9	2015	9.50

current row →

current group

Now, let's explicitly include other groups too. In the below example we start with the first group before the current group and end with the first group after the current group:

## GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING

1	2015	7.50
2	2015	8.00
3	2015	8.00
4	2015	8.30
5	2015	8.30
6	2015	8.30
7	2015	9.00
8	2015	9.00
9	2015	9.50

current row →

preceding group

following group

We can make it more complicated by using frame exclusions. For example, the statement below gives us the same result as above, but with the current group excluded:

## GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING EXCLUDE GROUP

1	2015	7.50	
2	2015	8.00	} preceding group
3	2015	8.00	
4	2015	8.30	} current group
current row → 5	2015	8.30	
6	2015	8.30	
7	2015	9.00	} following group
8	2015	9.00	
9	2015	9.50	

Here we exclude not the whole current group, but only the current row:

### GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING EXCLUDE CURRENT ROW

1	2015	7.50	
2	2015	8.00	} preceding group
3	2015	8.00	
4	2015	8.30	
current row → 5	2015	8.30	
6	2015	8.30	
7	2015	9.00	} following group
8	2015	9.00	
9	2015	9.50	

In the last example we exclude ties, i.e all peer rows, but we leave the current row intact:

### GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING EXCLUDE TIES

1	2015	7.50	
2	2015	8.00	} preceding group
3	2015	8.00	
4	2015	8.30	← tie row
current row → 5	2015	8.30	
6	2015	8.30	← tie row
7	2015	9.00	} following group
8	2015	9.00	
9	2015	9.50	

## Range mode

**RANGE** mode is different from the previous two, because it doesn't tie the rows together in any way. It instructs the database to work on a given range of values instead. The values that it looks at are the values of the sorting column. Postgres imposes a requirement, that in this mode you can put only one column in the **ORDER BY** clause.

The syntax is as follows:

```
RANGE BETWEEN frame_start AND frame_end [ frame_exclusion ]
```

Instead of specifying the number of rows or groups, here we have to specify the maximum difference of values that the window frame should comprise. Both ***frame\_start*** and ***frame\_end*** have to be expressed in the same units as the sorting column is.

Let's look at some examples.

In the below one we want to include all rows which sorting values differ no more than by 0.5 from the current row. The boundaries are inclusive, which means that the rows that differ by exactly 0.5 will be taken into consideration:

```
RANGE BETWEEN 0.5 PRECEDING AND 0.5 FOLLOWING
```

1	2015	7.50	
2	2015	8.00	
3	2015	8.00	0.5 preceding
4	2015	8.30	
5	2015	8.30	
6	2015	8.30	0.2 following
7	2015	8.50	
8	2015	9.00	
9	2015	9.50	

current row →

We can also mix the range with `CURRENT ROW`, which surprisingly means *the current group*. The effect is similar to what we saw in `GROUPS` mode. Again, all duplicates are included:

#### RANGE BETWEEN 0.5 PRECEDING AND CURRENT ROW

1	2015	7.50	
2	2015	8.00	
3	2015	8.00	0.5 preceding
4	2015	8.30	
5	2015	8.30	
6	2015	8.30	current group
7	2015	8.50	
8	2015	9.00	
9	2015	9.50	

0.5 preceding {

current row →

Frame exclusion options will work exactly the same as in the other modes. For example, to exclude the current row (which in this context really means *the current row*):

#### RANGE BETWEEN 0.5 PRECEDING AND CURRENT ROW EXCLUDE CURRENT ROW

1	2015	7.50
2	2015	8.00
3	2015	8.00
4	2015	8.30
5	2015	8.30
6	2015	8.30
7	2015	8.50
8	2015	9.00
9	2015	9.50

## Real-world examples

Now, it's finally time to do some realistic examples (or at least as close to being realistic as possible). Let's start with the problem I mentioned at the beginning of the article.

**Example 1. For each film find an average rating of all strictly better films in its release year.**

id	release_year	rating	avg_of_better
1	2015	7.00	8.50
2	2015	7.00	8.50
3	2015	7.00	8.50
4	2015	8.00	8.67
5	2015	8.50	9.00
6	2015	8.50	9.00
7	2015	9.00	[NULL]

*Result rows with average ratings of better films*

Because it's a real-world example we can't just assume that the input set will not contain duplicates in the `rating` column. Therefore using `ROWS` mode would give us a incorrect result. We have to choose `GROUPS` mode instead.

What we need here is all the films that are *strictly better* than the current one. We need to exclude the current row and all others that are rated the same as the current row, regardless of the order they come in.



To achieve that we should start with the first row in the group after the current group and finish at the end of the partition:

```
SELECT
  f.id, f.release_year, f.rating,
  AVG(rating) OVER (PARTITION BY release_year ORDER BY rating
    GROUPS BETWEEN 1 FOLLOWING AND UNBOUNDED FOLLOWING)
  AS avg_of_better
FROM films f
ORDER BY release_year, rating;
```

Of course, there are many correct solutions. We can also start with the current group and exclude it:

```
SELECT
  f.id, f.release_year, f.rating,
  AVG(rating) OVER (PARTITION BY release_year ORDER BY rating
    GROUPS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING EXCLUDE GROUP)
  AS avg_of_better
FROM films f
ORDER BY release_year, rating;
```

## Example 2. How many other films have the same rank as me?

id	release_year	rating	count_of_equal
1	2015	7.00	2
2	2015	7.00	2
3	2015	7.00	2
4	2015	8.00	0
5	2015	8.50	1
6	2015	8.50	1
7	2015	9.00	0

*Result rows with a count of equally rated films*

Now we need to select all the rows belonging to the current row's peer group and count them:

```
SELECT
  f.id, f.release_year, f.rating,
  COUNT(*) OVER (PARTITION BY release_year ORDER BY rating
    GROUPS BETWEEN CURRENT ROW AND CURRENT ROW EXCLUDE CURRENT ROW)
  AS count_of_equal
FROM films f
ORDER BY release_year, rating;
```

We use `CURRENT ROW` as both the beginning and the end of the partition in order to narrow the window frame down to the current group only. Remember that we're in the `GROUPS` mode, so `CURRENT ROW` actually means the current group. The last thing is to exclude the actual current row, so we add `EXCLUDE CURRENT ROW` clause, which always excludes just the current row. If you think that this syntax is misleading, don't worry, you're not the only one.

### Example 3. Find the rank of an immediately better rated film

id	release_year	rating	rating_of_better
1	2015	7.00	8.00
2	2015	7.00	8.00
3	2015	7.00	8.00
4	2015	8.00	8.50
5	2015	8.50	9.00
6	2015	8.50	9.00
7	2015	9.00	[NULL]

*Result rows with a rating of an immediately better film*

This example becomes tricky when we think about duplicates. There can be many rows with the same value as the current row, but we're interested in the first one that has a greater value. We need to skip all the duplicates:

**SELECT**

```
f.id, f.release_year, f.rating,
FIRST_VALUE(rating) OVER (PARTITION BY release_year ORDER BY rating
    GROUPS BETWEEN 1 FOLLOWING AND 1 FOLLOWING) AS rating_of_better
FROM films f
ORDER BY release_year, rating;
```

Here, with the `GROUPS` mode, we narrow down the framing window to the group immediately after the current group. Because this is a single group, all sorting values in the group are the same. We can choose any of them, e.g. by using `FIRST_VALUE()`.

We can also use the solution from Example 1 and use `MIN()` function. As always, there are many correct answers.

## Example 4. How many films are better by 0.5 or less?

The last example requires `RANGE` mode. One important thing to remember about is excluding the current group. In this example, we don't consider equally rated films as better.

id	release_year	rating	count_of_better
1	2015	7.00	0
2	2015	7.00	0
3	2015	7.00	0
4	2015	8.00	2
5	2015	8.50	1
6	2015	8.50	1
7	2015	9.00	0

*Result rows with a count of films with ratings higher by 0.5 or less*

This example allows us to use the `RANGE` mode. Specifying the upper boundary is easy - `0.5 FOLLOWING`. But the lower one is more problematic. To start with exactly the first strictly better film, we need to include the current group and everything above (`BETWEEN CURRENT ROW ...`) but then exclude the current group again (`EXCLUDE GROUP`). We're not in the `GROUPS`, so we can't just say `1 FOLLOWING`:

```
SELECT
  f.id, f.release_year, f.rating,
COUNT(*) OVER (PARTITION BY release_year ORDER BY rating
  RANGE BETWEEN CURRENT ROW AND 0.5 FOLLOWING EXCLUDE GROUP)
AS count_of_better
FROM films f
ORDER BY release_year, rating;
```

## Summary

Window functions are a very powerful SQL feature that can be extremely useful when you need your relational database system to do more complicated calculations for you. This article described window frames, which make window functions even more powerful. They allow you to flexibly narrow down the set of rows being used for calculations, so you can solve problems that previously couldn't be solved with window functions only.

It's worth to keep track of the latest SQL features being introduced to the popular relational database systems. It might take some time to learn and fully understand them, but if you want your database engine to do the heavy lifting, then they will prove useful for you.

*If you like my style of explaining things, you can check my article about other advanced SQL feature - [Common Table Expressions](#).*

## Resources

- [PostgreSQL documentation](#)
- [Modern SQL](#)

Share this post



Previous

---

© 2021 Michał Konarski. Powered by Jekyll using the So Simple Theme.

