Advanced SQL - window functions

BY MICHAŁ KONARSKI NOVEMBER 09, 2017

This post starts a series of articles discussing advanced SQL concepts that are well supported in popular database management systems for quite some time, but somehow many people still don't know about their existence. I'd like to explain them with examples, first giving a problem to solve using "plain old" SQL and then showing a better solution using advanced SQL.

The first feature that I'd like to present is **window functions**.

In this article I'll be using PostgreSQL 10, because it's the most feature-rich open source database available. Version 10 has been just released, but window functions have been available since 8.4, so any modern version will be fine.

> *This post was written in 2017, but everything it describes works in later versions of Postgres.*

Problem

As I promised, let's start with a problem. We'll be working with a very simple one-table database. The table contains information about films: years they were released in, ID of a category and their ratings according to some imaginary movie database:

films		
PK	**id**	*integer*
	release_year	*integer*
	category_id	*integer*
	rating	*numeric*

Films table schema

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

Input rows

Here comes the task:

Example 1. For each film find an average rating for all films released in the same year.

The result should look like this. All films released in the same year have the same average:

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

Result rows with year's average

Stop here for a second and think how would you tackle this problem using plain old SQL concepts like `JOIN`, `GROUP BY` and other things that come to your mind. There are at least few possible solutions.

One of the them is to use a subquery computing averages for all distinct years and joining them back with the query fetching all films:

```
SELECT
  f.id, f.release_year, f.rating, years.year_avg
```

```
FROM films f
LEFT JOIN (
  SELECT f.release_year, AVG(rating) AS year_avg
  FROM films f
  GROUP BY f.release_year
) years ON f.release_year = years.release_year
```

It doesn't look very complicated so far. Why don't we solve one more problem then?

Example 2. For each film find average ratings for all films released in the same year and separately in the same category.

And I'm expecting the following. Again, the same categories have equal values:

id	release_year	category_id	rating	year_avg	category_avg
1	2015	1	8.00	8.50	8.20
2	2015	2	8.50	8.50	7.90
3	2015	3	9.00	8.50	9.00
4	2016	2	8.20	8.30	7.90
5	2016	1	8.40	8.30	8.20
6	2017	2	7.00	7.00	7.90

Result rows with year's and category's averages

Looks easy. We just have to count the category averages in a similar way and join them together with the previous query:

```
SELECT
  f.id, f.release_year, f.category_id, f.rating,
  years.year_avg, categories.category_avg
FROM films f
LEFT JOIN (
  SELECT f.release_year, AVG(rating) AS year_avg
  FROM films f
  GROUP BY f.release_year
) years ON f.release_year = years.release_year
LEFT JOIN (
  SELECT f.category_id, AVG(rating) AS category_avg
  FROM films f
```

```
  GROUP BY f.category_id
) categories ON f.category_id = categories.category_id
```

The query gets more and more complicated though. It takes some time to read it and realize what exactly are we joining here.

Please notice also a pattern: we select a set of rows from a table and then join them with aggregated versions of the same row set. But we can't just use the `GROUP BY` in the main query because we want to get the full list of films as a result. Thus we have to copy-paste the main query to each subquery. Just imagine what if the main query would be complicated itself with a lot of joins, `WHERE` clauses or even its own grouping... Ideally we'd like to have a way to do some computations on a row set, but not altering it at the same time.

And this is exactly what **window functions** are all about.

Solution - window functions

PostgreSQL documentation has a nice definition of what window functions are:

> *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.*

In other words window functions allow to get aggregated results without actually making the result set aggregated. Let's see how they work in practice.

A simplified syntax looks like this:

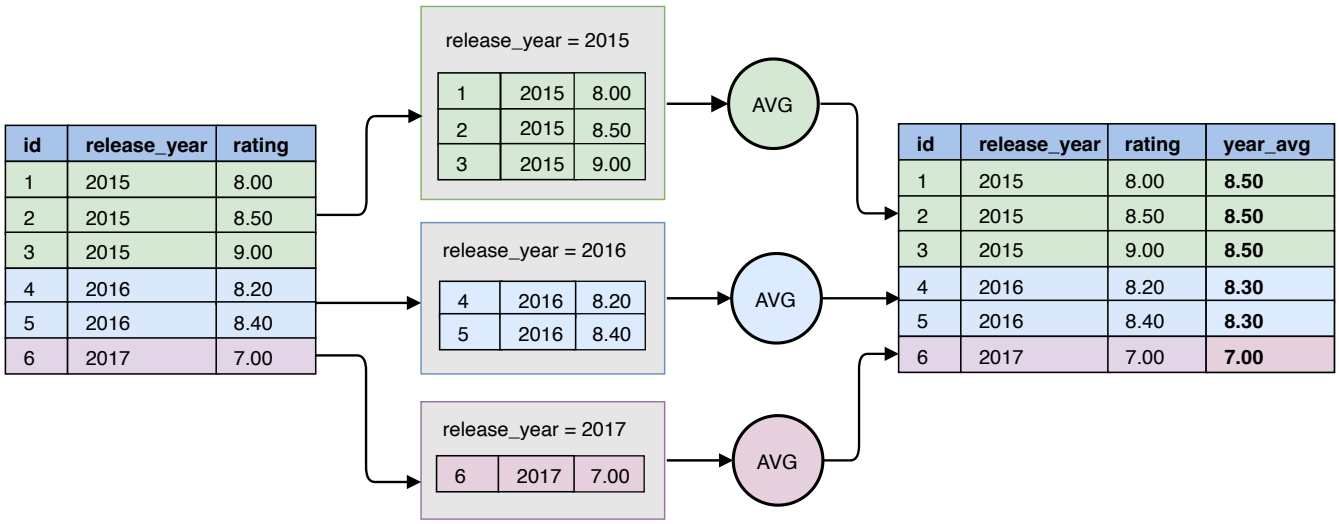
```
SELECT
function_name OVER ( window_definition )
FROM (...)
```

`window_definition` defines the set of rows that the current row is related to (I'm going to call it *a window*) and `function_name` specifies the function that we're gonna use to operate on rows in each window. For full syntax see the documentation.

Let's get back to the initial problem, where we needed to calculate year's average for each film. The solution using window functions is much simpler:

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

The window here is defined by a `PARTITION BY` clause. It instructs the database to divide the row set into smaller parts, partitions, putting all rows with the same `release_year` together. Then the aggregate function `AVG(score)` is run against each partition and the result is added to each row.



Window functions partitioning

As you can see all input rows are transfered to the result set, safe and sound. Additionally, any condition that we set on a main query applies to a window functions input also. In other words if we had added a `WHERE` clause filtering out some rows, these rows also would have been missing from a window function computation.

Window functions are a powerful feature. We can choose from a wide range of functions to use and ways to define windows. I'll mention just few interesting

possibilities here.

Example 3. For each film find its ranking position within its release year.

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

Result rows with year's ranking position

This task is different, because each row now has a distinct value within a partition - its position according to the `rating`. To solve this we have to use one of the order-aware functions - `RANK()`:

```
SELECT
 f.id, f.release_year, f.category_id, f.rating,
 RANK() OVER (PARTITION BY release_year ORDER BY rating DESC) AS year_rank
FROM films f
```

`RANK()` returns the position of a row within a window (with appropriate gaps when two or more rows have the same rank). To make it possible we had not only to partition the row set by a release year, but also to ensure that the rows inside each partition are sorted properly (otherwise we would get just rubbish). That's why we used the `ORDER BY` clause.

Example 4. For each film find its general ranking position.

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

Result rows with general ranking position

It's also possible to have the `ORDER BY` without `PARTITION BY`:

```
SELECT
 f.id, f.release_year, f.category_id, f.rating,
 RANK() OVER (ORDER BY rating DESC) AS general_rank
FROM films f
```

This way we instructed the database to create one big partition with all rows. It's useful when we want to operate on the whole row set altogether.

Example 5. For each film find the rating of the best film in its release year.

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

Result rows with year's best rating

```
SELECT
 f.id, f.release_year, f.category_id, f.rating,
 FIRST_VALUE(rating) OVER (PARTITION BY release_year ORDER BY rating DESC)
FROM films f
```

In the query above I used a new function `FIRST_VALUE()` which returns the requested value of the first row in a window. There are also other similar functions like `LAST_VALUE()` or `NTH_VALUE()`, returning value of the last or specific row, respectively. What's worth mentioning here, it's possible to change the boundaries of a window, so that it doesn't contain the whole partition. This can be done by using `RANGE` or `ROWS` clause.

Example 6. For each film find an average rating of all better films in its release year.

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

Result rows with an average ratings of better films

```
SELECT
 f.id, f.release_year, f.category_id, f.rating,
 AVG(rating) OVER (PARTITION BY release_year ORDER BY rating DESC
   ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING)
FROM films f
```

The `ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING` part instructs database to set the lower boundary on the window. Now, instead of going all the way down to the partition's end, it stops at the row right before the current row. So, effectively we operate only on rows that have higher `rating`.

Things get more complicated when the `rating` column contains duplicates. To achieve the same result we would need to use `RANGE` modifier instead of `ROWS`, but unfortunately Postgres doesn't currently support the `1 PRECEDING` part in that case.

A lot of new features have been added to Postgres since version 10. See my article about window frames, which covers the latest developments in window functions.

Conclusion

Window functions are my favorite advanced SQL feature. They simply allow to do aggregations without actually aggregating the result set. They are a flexible way to create sophisticated SQL queries, that otherwise would need to be long, complicated and hard to read and maintain. PostgreSQL and other databases offer a wide variety of different functions and options to specify the exact subset of rows we'd like to operate on.

Window functions were introduced in SQL 2003. Quite some time ago. Therefore, almost all popular RDBMSes implement them at least to some extent. The only exceptions are MySQL and SQLite. When it comes to MySQL however, it has been announced that the upcoming version 8.0 will support window functions.

Even that window functions are often considered as "advanced SQL", I believe that they are something that every SQL-oriented software developer should be familiar with.

Resources

- Modern SQL
- PostgreSQL documentation

Share this post

Previous Next