SQL Window functions basics

With PostgreSQL

Clément Prévost Follow

May 26, 2017 · 6 min read

> *For many developers, window functions are this mystical feature they only use once a year to build the famous Top-k query. In fact, they are part of the sql:2003 standard (yes, that's 10 years from now) and your DBMS should already have a great and optimised support for windowing functions.*

If you should keep only one relevant information from this article this must be that windowing functions are a select post-processing toolset. It allow us to iterate over the result of a select to compute values based on a wider view than just one row. You can use them to create groups of rows on which you may filter or create various aggregations. With this tool, you enable yourself to reason about the “next row” or “this group of rows” for each row in a select result, which is impossible in SQL 92.

Window functions concepts

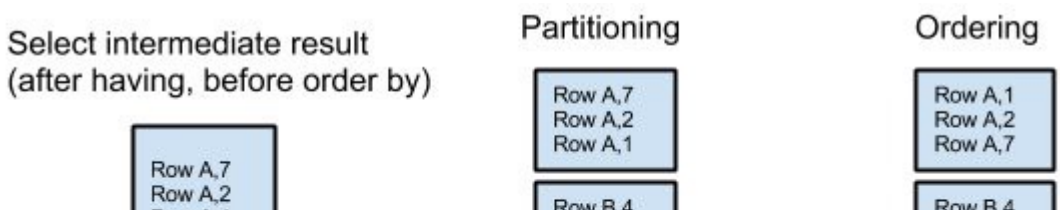
Window functions are calculated between the having and order by clauses. For each row of the partition, one can calculate a result based on the partition data. The processing order of the various SELECT parts is quite important to fully understand the intermediate data schema we are working with.

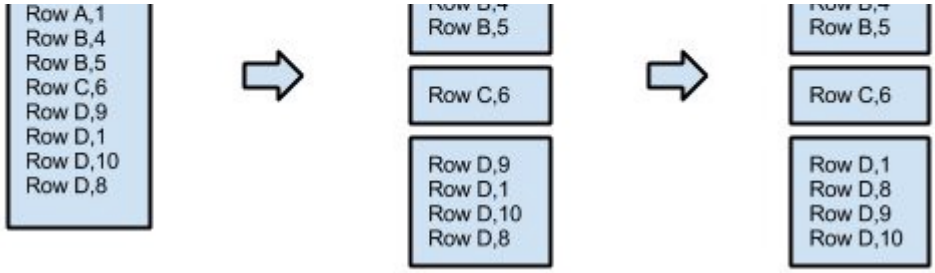
```
7 - SELECT
1 - [ FROM ... ]
2 - [ WHERE ... ]
3 - [ GROUP BY ... ]
4 - [ HAVING ... ]
5 - WINDOW window_name AS (
        [ PARTITION BY ... ]
        [ ORDER BY ... ]
        [ RANGE ... ]
    )
6 - [ ORDER BY ... ]
```

pg_partition_6.sql hosted with ♡ by GitHub view raw

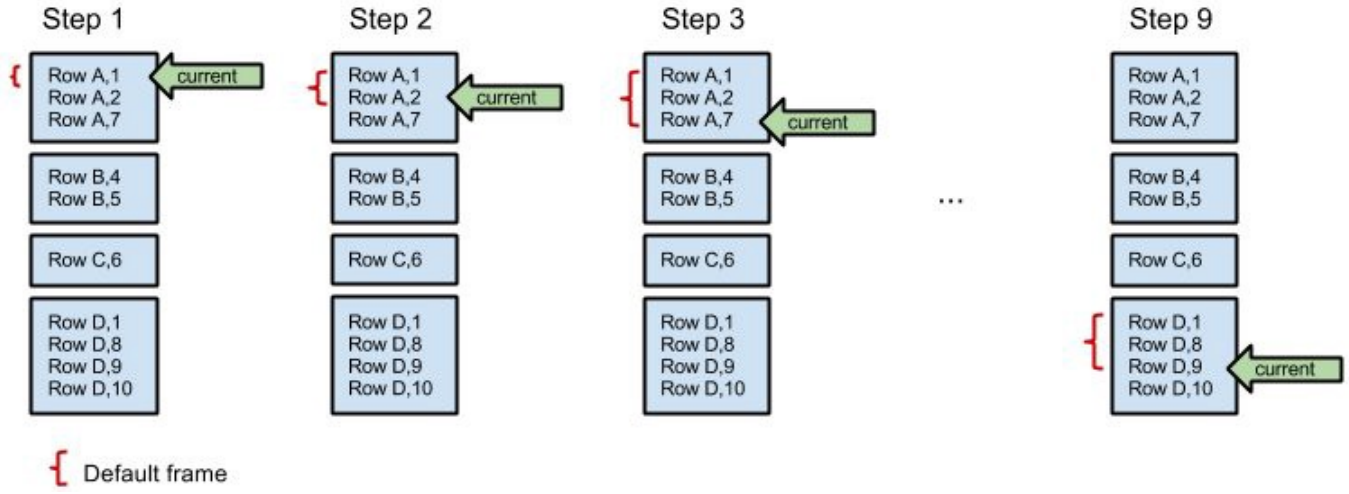
To be simplistic, window functions allow you to iterate over the result of your select before ordering the final result.

Window function works on row groups called partitions or windows, you can control the ordering and the window frame or range of these windows. A schema will explain it better than a thousand word:





On this first schema, the window function engine post-process the select result to break it into many parts called partitions. These partitions may have an inner ordering, different from the final result ordering.



This schema describe the behavior of the default window frame. The engine iterate through the result rows and store on each row the result of the aggregate function within the window frame.

By default, the frame goes from the first partition row to the current row, declaring an ORDER BY clause changes the default frame to the full partition. (cf. examples below) Due to this odd behaviour, you should always be aware of every aggregate function behaviour or set explicitly your window frame using the RANGE clause.

What? Show me an example!

The basics

Given a sample database containing 5000 employes randomly assigned to 5 categories from A to E:

```
SELECT * FROM employe LIMIT 10;

id  |       name         | salary | category
-----+--------------------+--------+----------
   1 | Employe name 1     |   1919 | D
   2 | Employe name 2     |   2188 | A
   3 | Employe name 3     |   5907 | A
   4 | Employe name 4     |   7554 | D
   5 | Employe name 5     |   8136 | B
   6 | Employe name 6     |   5933 | A
   7 | Employe name 7     |   2550 | D
   8 | Employe name 8     |   9539 | E
   9 | Employe name 9     |   3217 | E
  10 | Employe name 10    |   4925 | B
```

Now, imagine that you have to display a table with the employe information, the min, max and average salary of the employe category along with every employe row. Without window functions, the dummy way to do this kind of group-by-but-keep-every-row is the self-join, or more plausible, using php/java/ruby/whatever post-processing script.

However, grouping data and aggregating it is typically a database job, so let's tell PostgreSQL to do the heavy lifting for us:

```sql
SELECT
  *,
  min(salary)
  OVER report_by_category,
  avg(salary)
  OVER report_by_category,
  max(salary)
  OVER report_by_category
FROM employe
WINDOW report_by_category AS (
  PARTITION BY category )
LIMIT 5;
```

window-function-1.sql hosted with ♡ by GitHub view raw

```
 id  |       name         | salary | category | min  |       avg        | max
-----+--------------------+--------+----------+------+------------------+------
1003 | Employe name 1003 |   6515 | C        | 1009 | 5550.74427480916 | 9996
1005 | Employe name 1005 |   8187 | C        | 1009 | 5550.74427480916 | 9996
1001 | Employe name 1001 |   6106 | D        | 1001 | 5507.41912512716 | 9976
```

```
1002 | Employe name 1002 |   2491 | E        | 1001 | 5551.63221884498 | 9997
1004 | Employe name 1004 |   5130 | E        | 1001 | 5551.63221884498 | 9997
```

pg_export_1.txt hosted with ♡ by GitHub view raw

Now, each row contains informations computed with the whole partition. This is powerful as we might use any window or aggregate functions. But this comes with great responsibility. As always in SQL, it is surprisingly easy to write unmaintainable code. You should always document your window definitions and give them a meaningful name.

Playing with the window frame

While playing with window functions and using more advanced features, you may come to a point where everything goes crazy and your results are false. If your data is clean, then it's likely to be a window frame issue:

```sql
SELECT
  *,
  count(*)
  OVER report_by_category AS count,
  count(*)
  OVER (report_by_category
    ORDER BY salary)      AS ordered_count
FROM
  employe
WINDOW
    report_by_category AS (
    PARTITION BY
      category
  )
LIMIT 5;
```

window-function-2.sql hosted with ♡ by GitHub view raw

```
 id  |       name        | salary | category | count | ordered_count
-----+-------------------+--------+----------+-------+---------------
 2456 | Employe name 2456 |   1008 | A        |  1035 |             1
 2209 | Employe name 2209 |   1015 | A        |  1035 |             2
 3692 | Employe name 3692 |   1022 | A        |  1035 |             3
 3885 | Employe name 3885 |   1031 | A        |  1035 |             4
  436 | Employe name 436  |   1035 | A        |  1035 |             5
```

window-function-3.txt hosted with ♡ by GitHub view raw

Here, we ordered one of the two count function partitions by a salary column which should not impact our results. However, the two functions did not behaved the same way. This is because the unordered window frame range defaults to the whole partition, but the ordered window frame range is different. It goes from the first partition row (based on the order clause) to the current partition row.

Solving this issue is as simple as explicitly setting the default window frame to the whole partition using the RANGE clause.

```sql
SELECT
  *,
  count(*)
  OVER report_by_category         AS count,
  count(*)
  OVER ordered_report_by_category AS ordered_count
FROM
  employe
WINDOW
    report_by_category AS (
    PARTITION BY
      category
  ),
    ordered_report_by_category AS (
    PARTITION BY
      category
    ORDER BY
      salary
    RANGE
    BETWEEN UNBOUNDED PRECEDING
    AND UNBOUNDED FOLLOWING
  )
LIMIT 5;
```

window-function-3.sql hosted with ♡ by GitHub view raw

```
 id  |       name        | salary | category | count | ordered_count
------+-------------------+--------+----------+-------+---------------
 2456 | Employe name 2456 |   1008 | A        |  1035 |          1035
 2209 | Employe name 2209 |   1015 | A        |  1035 |          1035
```

```
3692 | Employe name 3692 |   1022 | A        |  1035 |          1035
3885 | Employe name 3885 |   1031 | A        |  1035 |          1035
 436 | Employe name 436  |   1035 | A        |  1035 |          1035
```

window-function-4.txt hosted with ♡ by GitHub view raw

Here we explicitly set the ordered window frame to the whole partition and finally, our simple count function behave as expected.

Use this example as a headache avoidance guideline: ordering a window without defining the frame is equivalent to developing vicious and undetectable bugs.

Know your enemy

This example is here to show you the limitations of windowing functions. As we explained in the introduction, you are playing with a post-processing tool, not a magical silver bullet. You have to understand the data flow between the various select clauses. As a result the windowing functions performance relies directly on the select result size after the HAVING clause.

To prove this point, we rely on the Top-K query. The goal is to retrieve the first K rows of each categories of products, of each employe department, the most productive shops of each country, …

We Here is how you retrieve the 2 employes with the highest salaries of each category:

```
SELECT *
FROM (
       SELECT
         *,
         row_number()
         OVER ordered_report_by_category
       FROM
         employe
       WINDOW ordered_report_by_category AS (
         PARTITION BY
           category
         ORDER BY
           salary DESC
         RANGE
         BETWEEN UNBOUNDED PRECEDING
```

```
          AND UNBOUNDED FOLLOWING
        )
      ) tmp
WHERE
  row_number <= 2
;
```

window-function-4.sql hosted with ♡ by GitHub view raw

```
 id  |       name        | salary | category | row_number
-----+-------------------+--------+----------+------------
 4023 | Employe name 4023 |   9996 | A        |          1
 3795 | Employe name 3795 |   9993 | A        |          2
 1050 | Employe name 1050 |   9990 | B        |          1
 2664 | Employe name 2664 |   9988 | B        |          2
  956 | Employe name 956  |   9996 | C        |          1
 2843 | Employe name 2843 |   9986 | C        |          2
  607 | Employe name 607  |   9976 | D        |          1
 4983 | Employe name 4983 |   9970 | D        |          2
 1455 | Employe name 1455 |   9997 | E        |          1
 1528 | Employe name 1528 |   9990 | E        |          2
```

window-function-5.txt hosted with ♡ by GitHub view raw

The setup is a little bit different here. We are working with the same table but populated with 10M rows. I also added a btree index on the category and salary columns but the query still run in about 80 sec on my laptop.

What you are saying with this query is:

- Retrieve ALL my employe data
- Compute a row number based on the salary column for each category
- Pass this result to another query
- Only keep rows with row_number >= 2

Any kind of index won't help you here, the query semantic told the DBMS to retrieve ALL your employe data without complains and he is just doing his job.

An efficient way of computing this result is by using the UNION technique:

```sql
SELECT *
FROM
  (
    (SELECT *
     FROM employe
     WHERE category = 'A'
     ORDER BY salary DESC
     LIMIT 2)
    UNION (SELECT *
           FROM employe
           WHERE category = 'B'
           ORDER BY salary DESC
           LIMIT 2)
    UNION (SELECT *
           FROM employe
           WHERE category = 'C'
           ORDER BY salary DESC
           LIMIT 2)
    UNION (SELECT *
           FROM employe
           WHERE category = 'D'
           ORDER BY salary DESC
           LIMIT 2)
    UNION (SELECT *
           FROM employe
           WHERE category = 'E'
           ORDER BY salary DESC
           LIMIT 2)
  ) AS tmp
ORDER BY category, salary DESC;
```

window-function-5.sql hosted with ♡ by GitHub view raw

```
  id  |       name        | salary | category
------+-------------------+--------+----------
 4023 | Employe name 4023 |   9996 | A
 3795 | Employe name 3795 |   9993 | A
 1050 | Employe name 1050 |   9990 | B
 2664 | Employe name 2664 |   9988 | B
  956 | Employe name 956  |   9996 | C
 2843 | Employe name 2843 |   9986 | C
```

```
 9     607 | Employe name 607  |   9976 | D
10    4983 | Employe name 4983 |   9970 | D
11    1455 | Employe name 1455 |   9997 | E
12    1528 | Employe name 1528 |   9990 | E
```

window-function-5.txt hosted with ♡ by GitHub view raw

(Look at the raw version) Here, the semantic drastically differ from the window functions query even if the final result is the same. The various select on each category are able to use the salary indexe to reduce ASAP the number of working rows. On the same laptop this query run in 10 ms, about 8000 times faster than the window function version.

Keeping in mind that window function are just a post-processing toolset along with the select clauses computation order helps you avoid these kind of mistakes.

What can I use this for?

If you are more productive writing a custom aggregation using your favorite scripting language, the only use case for you is performance improvement:

- Your DBMS is typically written in a low level language and have been optimized for years.
- Your queries are written in a declarative fashion that makes plenty of room for automatic optimizations.
- If you filter on window function results, you may significantly cut the data flow size between the database and you application (TopK query), resulting in less object creation and OOP overhead.

The other use cases being:

- Building complex reports directly in SQL
- Create complex migration scripts directly in SQL, avoiding comings and goings between application code and database.
- Showing off your incredible skills and mastery of the SQL language

Great article, now give me some data to play with!

Here is a query that creates the table from the example section:

```sql
CREATE TABLE employe AS (
  SELECT
    n                                          AS id,
    'Employe name' || n                        AS name,
    trunc(random() * 9000 + 1000)              AS salary,
    chr((65 + trunc(random() * 5)) :: INTEGER) AS category
  FROM
       generate_series(1, 5000, 1) AS n
);
```

window-function-6.sql hosted with ♡ by GitHub view raw

Please have a quick look of the more in-depth tutorial from depesz.com and read the PostgtreSQL documentation about the detailed list of window functions and don't forget that every aggregation function is available for windowing.

Further reading

- http://www.depesz.com/2012/11/20/window-window-on-the-wall/
- http://www.postgresql.org/docs/9.1/static/queries-table-expressions.html#QUERIES-WINDOW
- http://www.postgresql.org/docs/9.1/static/sql-select.html#SQL-WINDOW
- http://www.postgresql.org/docs/9.1/static/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS
- http://www.postgresql.org/docs/9.1/static/functions-window.html
- http://www.postgresql.org/docs/9.1/static/tutorial-window.html

. . .

Originally published at inovia.fr on June 7, 2013.