Window functions in PostgreSQL 8.4

Оглавление

Introduction

Very often we need access to previous/next rows in row set to calculate particular values for columns. The classical case is running totals. If we have two columns "Number" and "Amount" we may want third column "Total" where in first row we have value of Amount in second – Amount(1) + Amount(2). In third – Amount(1) + Amount(2) + Amount(3), etc. To summarize data we use GROUP BY clause, but the main trouble is that each row of row set may take part only in one group and therefore in one summary. In that paradigm we need to multiply rows to get a separate copy of particular row for each summary. For running total we need n*n/2 rows for n rows in result.

But a great feature **windowing** was introduced by SQL:2003. Windowing allows us to do 2 major things

- Aggregate calculation over rows of the query results. GROUP BY needs separate subset of rows to create each summary value. Windowing allow us to create summary values on any subset of query results without need to query the subset separately for each summary value. In other words we calculate summaries during querying the result rows and mix summaries and single values within one row.
- Access particular previous and next rows while calculating values of current row.

But since the SQL:2003 standard was announced we had no support for windowing in PostgreSQL. At last, in version 8.4 we get the feature ready to use. In this article I show you the power and ease of window functions in the context of practical tasks.

I won't describe windowing. There are many articles about it: SQL:2008, PostgreSQL documentation, PostgreSQL window function presentation, Window functions for ORACLE, Windowing brief and list of publications, etc.

Here I want to show practical use of windowing because it allows us to design very efficient queries. Without windowing we ought to use procedural languages for those tasks to solve them really efficiently.

Simple common cases

Some simple usages for window functions as a “good form”.

Running totals

How to use window sum() over().

```
select
	*,
	sum(Amount) over(order by Number) Total
from Docs;
```

Or grouping by date

```
select
	doc_date,
	sum(sum(amount)) over(order by doc_date) Total
from Docs
group by doc_date;
```

Gap analysis

How to use window lead() over()

I had to find gaps in sequences in two different projects; see my post for details. And even invent a nice solution where I utilized EXCEPT operator. Window functions allow one do gap search in more natural way as “next minus current > 1”. To implement it we, first, should get distances to next number for each row:

```
select
	Number,
	lead(Number) over(order by Number) - Number distance_to_next
from Docs;
```

And then get the list of holes

```
select * from (
	select
		Number+1 gap_start,
		lead(Number) over(order by Number) - Number - 1 gap_length
	from Docs ) t
where gap_length > 1
```

Motivation table

This section shows "nth_value() over()".

HR department often use KPI to motivate personnel. They calculate KPI for each employee and show the rate to the top N employees. This is done separately for each department. Assume we have the following sales database:

```
create sequence sales_seq;
create table Sales
(
	SaleID int primary key default nextval('sales_seq'),
	Department varchar(30),
      SalesManager varchar(30),
      Subject varchar(100),
      Amount numeric
);

insert into Sales (Department, SalesManager, Subject, Amount) Values
('Computers', 'John Dale', 'Notebook', 100),
('Computers', 'Sam Dakota', 'Desktop computer', 100),
('Computers', 'Sam Dakota', 'Desktop computer', 70),
('Computers', 'Eve Nicolas', 'Pocket PC', 270),
('Computers', 'Eve Nicolas', 'Smartphone', 150),
('Cars', 'Nick Hardy', 'Mercedes', 300),
('Cars', 'James Wilson', 'BMW', 100),
('Cars', 'Tom Sawyer', 'Audi', 170);
```

So we need to group sales by sales manager and order them by department and amount

```
select
      Department,
      Salesmanager,
      sum(Amount) as Amount
from Sales
group by Department, SalesManager
order by Department, Amount desc;
```

Department	Sales Manager	Amount
"Cars"	"Nick Hardy"	300
"Cars"	"Tom Sawyer"	170
"Cars"	"James Wilson"	100
"Computers"	"Eve Nicolas"	420
"Computers"	"Sam Dakota"	170
"Computers"	"John Dale"	100

Now we need to calculate ratio for each amount to Nick Hardy's and Tom Sawyer's Amount in Cars department and to Eve Nicolas's and Sam Dakota's in Computers department. To do that we use nth_value() window function:

```
select
      Department,
      Salesmanager,
      sum(Amount) as Amount,
      (nth_value(sum(Amount),1) over w / sum(Amount))
            ::numeric(18,1) "rate to top 1",
      (nth_value(sum(Amount),2) over w / sum(Amount))
            ::numeric(18,1) "rate to top 2"
from Sales
group by Department, SalesManager
window w as (
      partition by Department
      order by Amount desc
      ROWS BETWEEN UNBOUNDED PRECEDING
      AND UNBOUNDED FOLLOWING)

order by Department, Amount desc;
```

Department	Sales Manager	Amount	Rate to top 1	Rate to top 2
"Cars"	"Nick Hardy"	300	1.0	0.6
"Cars"	"Tom Sawyer"	170	1.8	1.0
"Cars"	"James Wilson"	100	3.0	1.7
"Computers"	"Eve Nicolas"	420	1.0	0.4
"Computers"	"Sam Dakota"	170	2.5	1.0
"Computers"	"John Dale"	100	4.2	1.7

In the query we expand window from default first-to-current range to first-to-last range. This allows us to calculate rates for the top 1 and top 2 too.

Derivative and Integral

This section shows lead() over() and lag() over.

Now it is possible to effectively calculate recurrent formulas. Recurrent formulas are the formulas of kind

$$X_{i+1} = f(x_i)$$

Such formulas are often used for mathematical calculations like integrating, deriving etc. Here I show you how simple it is with window functions.

Derivative

Assume we have a table, that contains points of some curve in form of ordered points (x,y). And we need to calculate derivative of a function in each point. To show the real power of window functions I select three-point approximation for the derivative. In plpgsql it looks as follows

```
create or replace function derive_three_point(
      t real, -- The point to calculate derivative at.
      x0 real, x1 real, x2 real, -- Three sequential points
      f0 real, f1 real, f2 real) -- to approximate func.
returns real as $$
declare
      a real;
      b real;
      h1 real;
      h2 real;
begin
      h1 = x1 - x0;
      h2 = x2 - x0;
      a = (f2-f0-h2/h1*(f1-f0))/((h2*h2)-h1*h2);
      b = (f1-f0-a*(h1*h1))/h1;
      return 2*a*(t-x0) + b;
end;
$$  LANGUAGE plpgsql;
```

As we use 3 points to calculate the derivative, we should mention that at the beginning we should use current and 2 next points when at the end we should use current and 2 previous points. So we need to calculate 5 points for each row and use 3 most suitable. In SQL it may be written as follows:

```
-- Create a table to store points
create table Func(x real, y real);
```

```
-- Create points using y=sin(x) for
-- x: [0;1] with step 0.1
insert into Func
select x, sin(x) from
      (select x::real/10 x from generate_series(0,10) x) t;

-- Calculate the derivative
select
      x,
      case
            -- when at left-most point - use next points
            when x_1 is null then derive_three_point(x, x, x1, x2, y, y1, y2)

            -- when in right-most point - use previous point
            when x1 is null then derive_three_point(x, x_2, x_1, x, y_2, y_1,
y)

            -- in the middle use centralized formula
            else derive_three_point(x, x_1, x, x1, y_1, y, y1)

      end::numeric(18,3) derivative,

      -- and also calculate exact value of derivate in all points
      (cos(x))::numeric(18,3) exact_derivative
from
(
      -- Here we prepare 5 points ( current +/- 2 pints)
      select
            lag(x, 2) over w x_2,
            lag(x, 1) over w x_1,
            x,
            lead(x, 1) over w x1,
            lead(x, 2) over w x2,

            lag(y, 2) over w y_2,
            lag(y, 1) over w y_1,
            y,
            lead(y, 1) over w y1,
            lead(y, 2) over w y2
      from Func
      window w as (order by x)
) coef;
```

X	Approximate derivative	Exact derivative
0	1.003	1.000
0.1	0.993	0.995
0.2	0.978	0.980
0.3	0.954	0.955
0.4	0.920	0.921
0.5	0.876	0.878
0.6	0.824	0.825
0.7	0.764	0.765
0.8	0.696	0.697
0.9	0.621	0.622
1	0.542	0.540

Integral

Query for the integral is a little bit more complex. Integral is a running sum of elementary squares under the function we need two phases. 1 – to calculate elementary squares, 2- to calculate running sum. In PostgreSQL window function can't be nested. So we should use a sub-query technique. If our function equals to zero outside the x'es range within the Func table, we have our integral equals to zero at left-most point. It allows us easily us use current and previous row to run calculation and never think about edge effects so far (in previous case we had to).

```
select
      x,
      (coalesce(sum(ydx) over(order by x), 0))::numeric(18,3) integral,
      (-cos(x) + 1)::numeric(18,3) exact_integral
from
(
      select
            x,
            (y + lag(y) over w) / 2 * (x - lag(x) over w) ydx
      from Func
      window w as (order by x)
) t;
```

X	Approximate elementary function	Exact elementary function
0	0.000	0.000
0.1	0.005	0.005
0.2	0.020	0.020
0.3	0.045	0.045
0.4	0.079	0.079
0.5	0.122	0.122
0.6	0.175	0.175
0.7	0.235	0.235
0.8	0.303	0.303
0.9	0.378	0.378
1	0.459	0.460

One more interesting task to discuss here is noise reduction. Here I mean the technique to determine and remove points that were measured with definitely high error. Not the audio noise (the)

Stock control with FIFO costing

This section shows the combination of window functions in simple but real life example.

Stock control is a set of simple techniques to manage the stock. One of them is FIFO / LIFO method. This method helps us to manage stock when we buy identical parts from different suppliers under the different prices. We put all parts in single box but know the price for each part we then borrow from the box. And also what is the cost of the currently stocked parts.

In this article I show the use of window functions to implement FIFO method. To control parts movement through the stock we need an entity that helps us to control cost and size of every part set that arrives or leaves the stock.

```
create sequence number_seq;
create table Move
(
	Number int primary key default nextval('number_seq'),
	PartCount int,
	Cost numeric,
	Direction int -- 1 - receipt, 2 - shipment
);
```

This entity describes a set of parts under particular price. So all parts move are sequentially numbered. Now we need to link all moves in order to be able to say what moves of type “receipt” provide parts for each move of type “shipment”.

Now make some test moves:

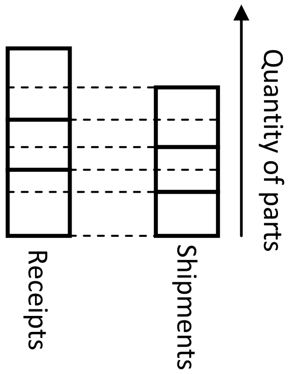
```
insert into Move(PartCount, Cost, Direction) Values
-- receipts go with supplier cost
(30, 30, 1), (20, 24, 1), (30, 36, 1),
-- shipment cost will be defind with FIFO costing
(15, null, 2), (50, null, 2), (5, null, 2);
```

What we now have in the Move table:

```
select
	Number, PartCount, Cost,
	(Cost/PartCount)::numeric(18,2) Price,
	Direction
from Move order by Number;
```

Number	Parts count	Cost	Part price	Direction
1	30	30.00	1.00	1
2	20	24.00	1.20	1
3	30	36.00	1.20	1
4	15			2
5	50			2
6	5			2

To correlate receipts and shipments we can stack them in parallel stacked bars and the links (as mentioned above) will be clearly seen.



Bold rectangles are moves, vertical spaces between dashed lines are links. So to find links we need to enumerate dashed lines and link left and right moves of each line. The vertical distance between dashed lines is the count of parts for the link. Each column may be got as running total for each type of move:

```
select
      Number ReceiptNumber,
      sum(PartCount) over (order by Number) Total
from Move
where Direction = 1;
```

Receipt number	Total count
1	30
2	50
3	80

The stacked bar is built but we have no zero value so it is better to rewrite query as follows

```
-- Bottoms of receipts
select
      Number ReceiptNumber,
      sum(PartCount) over (order by Number) - PartCount Total
from Move
where Direction = 1

union all

-- Top of last receipt
select 0, sum(PartCount) from Move where Direction = 1;
```

Receipt number	Total count
1	0
2	30
3	50
0	80

Now just combine receipts and shipments as follows

```
-- Bottoms of receipts
select
      0 ShipmentNumber,
      Number ReceiptNumber,
      sum(PartCount) over (order by Number) - PartCount Total
from Move where Direction = 1

union all

-- Top of last receipt
select 0, 0, sum(PartCount) from Move where Direction = 1

union all

-- Bottoms of shipments
select
      Number ShipmentNumber,
      0 ReceiptNumber,
      sum(PartCount) over (order by Number) - PartCount Total
from Move where Direction = 2

union all

-- Top of last shipment
select 0, 0, sum(PartCount) from Move where Direction = 2

order by Total;
```

Shipment number	Receipt number	Total count
0	1	0
4	0	0
5	0	15
0	2	30
0	3	50
6	0	65
0	0	70
0	0	80

There can be duplicate rows if sum(receipts) = sum(shipments). But we can't filter them as we use window functions that run after WHERE clause is evaluated. We will do it in the next step.

Each row of the query represents a link. But it has only one end defined (receipt or shipment). The other end we can define if prolong last nonzero value from previous rows. We have no such window function. But if we order rows by running total count (as it is done in table above) it will be simply maximum of previous values. The count of parts of link is defined as next total value minus current one. The window function for next value is lead(). So, we can write the following query

```
select
      max(ReceiptNumber) over (order by Total) ReceiptNumber,
      max(ShipmentNumber) over (order by Total) ShipmentNumber,
      lead(Total) over (order by Total) - Total Count
from
(
      -- Bottoms of receipts
      select
            0 ShipmentNumber,
            Number ReceiptNumber,
            sum(Count) over (order by Number) - Count Total
      from Move
      where Direction = 1

      union all

      -- Top of last receipt
      select 0, 0, sum(Count) from Move where Direction = 1

      union all

      -- Bottoms of shipments
      select
            Number ShipmentNumber,
            0 ReceiptNumber,
            sum(Count) over (order by Number) - Count Total
      from Move
      where Direction = 2

      union all

      -- Top of last shipment
      select 0, 0, sum(Count) from Move where Direction = 2
) t
where -- less than maximum shipment
      Total <= (select sum(Count) from Move where Direction = 2)
```

Receipt number	Shipment number	Link parts count
1	4	0
1	4	15
1	5	15
2	5	20
3	5	15
3	6	5
3	6	

Links are built. We need to remove unnecessary rows and insert rows into the table for the next use:

```
create table RSLink
(
      ShipmentNumber int,
      ReceiptNumber int,
      PartCount int
);


insert into RSLink(ReceiptNumber, ShipmentNumber, PartCount)
select * from
(
      select
            max(ReceiptNumber) over (order by Total) ReceiptNumber,
            max(ShipmentNumber) over (order by Total) ShipmentNumber,
            lead(Total) over(order by Total) - Total PartCount
      from
      (
            -- Bottoms of receipts
            select
                  Number ReceiptNumber,
                  0 ShipmentNumber,
                  sum(PartCount) over (order by Number) - PartCount Total
            from Move
            where Direction = 1

            union all

            -- Top of last receipt
            select 0, 0, sum(PartCount) from Move where Direction = 1

            union all

            -- Bottoms of shipments
            select
                  0 ReceiptNumber,
                  Number ShipmentNumber,
                  sum(PartCount) over (order by Number) - PartCount Total
            from Move
            where Direction = 2

            union all

            -- Top of last shipment
            select 0, 0, sum(PartCount) from Move where Direction = 2
      ) t
      where -- less than maximum shipment
            Total <= (select sum(PartCount) from Move where Direction = 2)
) t2
where PartCount <> 0 and PartCount is not null;
```

Receipt number	Shipment number	Part count
1	4	15
1	5	15
2	5	20
3	5	15
3	6	5

Now we are ready to calculate shipment costs using FIFO costing

```
update Move shipment
set Cost = fifo.ShipmentCost
from
(
      select
            link.ShipmentNumber,
            sum(receipt.Cost*link.PartCount/receipt.PartCount) ShipmentCost
      from RSLink link join Move receipt on receipt.Number =
link.ReceiptNumber
      group by link.ShipmentNumber
) fifo
where fifo.ShipmentNumber = shipment.Number;
```

Now look what we get

```
select
      Number, PartCount, Cost,
      (Cost/PartCount)::numeric(18,2) Price,
      Direction
from Move order by Number;
```

Number	Parts count	Cost	Part price	Direction
1	30	30.00	1.00	1
2	20	24.00	1.20	1
3	30	36.00	1.20	1
4	15	15.00	1.00	2
5	50	57.00	1.14	2
6	5	6.00	1.20	2

As we see, costs for shipment are calculated and prices slightly differ from the prices in receipts. What we now can easily do is to calculate stock balance:

```
select
       sum(PartCount * (3-Direction*2)) "remaining parts",
       sum(Cost      * (3-Direction*2)) "cost of remaining parts"
from Move;
```

Remaining parts	Cost of remaining parts
10	90.00

Conclusion

Congratulations! PostgreSQL lovers are on the new level of efficiency and effectiveness. All queries mentioned above are very fast (single table scan) and this become possible due to window functions. However, in version 8.4 some features can’t be found. As for me I’d like to have a filter based on value of window functions. There is a clause for aggregates: WHERE -> GROUP BY -> HAVING. So it’s good to have something like WHERE -> window calculations -> WINDOW_HAVING. And as far as window can be easily materialized in memory (by removing first and adding next row) there can be very useful to treat window as sub-query and allow all operations that allowed to real sub-queries.

bye.