EDB
Postgres® for the AI Generation

Window Functions Are Easier and More Powerful Than You Think

Vik Fearing
Swiss PGDay, Rapperswil
June 27, 2024

What are we doing here?

This presentation is all about window functions, a powerful feature of the SQL Standard, almost fully implemented in PostgreSQL.

You will learn about the handful of pure window functions provided by the Standard and PostgreSQL.

More importantly, you will learn how to write window specifications, with elaborate frames that can eliminate hundreds of lines of application code.

Why me?

- Why not? You could do this, too!
- I have been involved in PostgreSQL development since 2008, with a special interest in SQL language features.
- I am a member of the SQL Standards committee.

The slides are available here:

https://www.pgday.ch/common/slides/2024_windowfunctions.pdf

Sample Data

EDB

Sample Data

The data we will be using for the examples comes from the *pagila* sample database available on GitHub. This is a port of the *sakila* sample database from MySQL.

We will look at customer names and the stores they shop at.

xzilla/**pagila**

PostgreSQL Sample Database

1 Contributor · 0 Issues · 57 Stars · 16 Forks

Sample Data

Let’s focus on a small portion of the customers so that we can fit all this on the slides. 😊

```
SELECT store_id, first_name, last_name
FROM customer
WHERE first_name >= 'W'
ORDER BY first_name, last_name;
```

Sample Data

store_id	first_name	last_name
1	WADE	DELVALLE
1	WALLACE	SLONE
2	WALTER	PERRYMAN
1	WANDA	PATTERSON
2	WARREN	SHERROD
2	WAYNE	TRUONG
1	WENDY	HARRISON
2	WESLEY	BULL
2	WILLARD	LUMPKIN
2	WILLIAM	SATTERFIELD
2	WILLIE	MARKHAM
2	WILLIE	HOWELL
2	WILMA	RICHARDS
2	YOLANDA	WEAVER
2	YVONNE	WATKINS
1	ZACHARY	HITE

Concepts

Concepts

- A **row** is a sequence of one or more values. The number of values in a row is the row’s **degree**.
- A **table** is a collection of zero or more rows. The number of rows in a table is the table’s **cardinality**.
- A **base table** is a table that is persisted, and is generally what is thought of when the term “table” is used. It is created with the `CREATE TABLE` statement.
- A **derived table** is a table that is derived from one or more other tables. It is the result of the `FROM` clause along with any associated `JOIN` clauses.
- A **grouped table** is a derived table divided into groups according to a `GROUP BY` clause. The resulting derived table has one row per group. Functions operating on groups are called **aggregates**.
- A **windowed table** is table with one or more windows defined. Functions operating on windows are called **window functions**.

Window Specifications

EDB

Window Specifications

We are going to add a `WINDOW` clause to our query.

It starts with the keyword `WINDOW` and then is a list of comma-separated window specifications.

The most simple specification is just nothing.

```
WINDOW w AS ()
```

This defines the window as being over the entire resultset of the query.

ROW_NUMBER()

The most basic of all window functions is **ROW_NUMBER()**, which simply assigns a monotonically increasing number to each row.

```
SELECT store_id, first_name, last_name,
       ROW_NUMBER() OVER w
FROM customer
WHERE first_name >= 'W'
WINDOW w AS ()
ORDER BY first_name, last_name;
```

ROW_NUMBER()

store_id	first_name	last_name	row_number
1	WADE	DELVALLE	16
1	WALLACE	SLONE	14
2	WALTER	PERRYMAN	8
1	WANDA	PATTERSON	1
2	WARREN	SHERROD	11
2	WAYNE	TRUONG	10
1	WENDY	HARRISON	2
2	WESLEY	BULL	12
2	WILLARD	LUMPKIN	15
2	WILLIAM	SATTERFIELD	7
2	WILLIE	MARKHAM	9
2	WILLIE	HOWELL	6
2	WILMA	RICHARDS	5
2	YOLANDA	WEAVER	4
2	YVONNE	WATKINS	3
1	ZACHARY	HITE	13

ROW_NUMBER()

If we want the window functions to operate over sorted data, we have to sort the data in the window specification. This is independent of the main `ORDER BY` clause.

```
SELECT store_id, first_name, last_name,
       ROW_NUMBER() OVER w
FROM customer
WHERE first_name >= 'W'
WINDOW w AS (ORDER BY first_name)
ORDER BY first_name, last_name;
```

ROW_NUMBER()

store_id	first_name	last_name	row_number
1	WADE	DELVALLE	1
1	WALLACE	SLONE	2
2	WALTER	PERRYMAN	3
1	WANDA	PATTERSON	4
2	WARREN	SHERROD	5
2	WAYNE	TRUONG	6
1	WENDY	HARRISON	7
2	WESLEY	BULL	8
2	WILLARD	LUMPKIN	9
2	WILLIAM	SATTERFIELD	10
2	WILLIE	HOWELL	12
2	WILLIE	MARKHAM	11
2	WILMA	RICHARDS	13
2	YOLANDA	WEAVER	14
2	YVONNE	WATKINS	15
1	ZACHARY	HITE	16

What's this!?

ROW_NUMBER(), RANK(), and DENSE_RANK()

Incomplete sorting can produce unpredictable results. If we have more columns to sort by, we could add those. If we don’t, or if we don’t want to have a total ordering, we can use ranking functions.

```
SELECT store_id, first_name, last_name,
       ROW_NUMBER() OVER w,
       RANK() OVER w,
       DENSE_RANK() OVER w
FROM customer
WHERE first_name >= 'W'
WINDOW w AS (ORDER BY first_name)
ORDER BY first_name, last_name;
```

ROW_NUMBER(), RANK(), and DENSE_RANK()

store_id	first_name	last_name	row_number	rank	dense_rank
1	WADE	DELVALLE	1	1	1
1	WALLACE	SLONE	2	2	2
2	WALTER	PERRYMAN	3	3	3
1	WANDA	PATTERSON	4	4	4
2	WARREN	SHERROD	5	5	5
2	WAYNE	TRUONG	6	6	6
1	WENDY	HARRISON	7	7	7
2	WESLEY	BULL	8	8	8
2	WILLARD	LUMPKIN	9	9	9
2	WILLIAM	SATTERFIELD	10	10	10
2	WILLIE	HOWELL	12	11	11
2	WILLIE	MARKHAM	11	11	11
2	WILMA	RICHARDS	13	13	12
2	YOLANDA	WEAVER	14	14	13
2	YVONNE	WATKINS	15	15	14
1	ZACHARY	HITE	16	16	15

Distribution Functions

There are two distribution window functions provided:

`PERCENT_RANK()` is defined the same as the function of the same name in popular spreadsheet applications. Its value for each row is defined as the rank of the row minus 1 divided by the number of rows minus 1. This results in values ranging from 0.0 to 1.0.

`CUME_DIST()` is defined as the *statistical cumulative distribution* function. It is computed as the number of rows prior to or peer with the current row divided by the total number of rows.

Window Partitions

EDB

ROW_NUMBER(), RANK(), and DENSE_RANK()

We have ranked all of our customers by name, but we want to actually do this for each store and not globally. We can partition the window to achieve this.

```
SELECT store_id, first_name, last_name,
       ROW_NUMBER() OVER w,
       RANK() OVER w,
       DENSE_RANK() OVER w
FROM customer
WHERE first_name >= 'W'
WINDOW w AS (
    PARTITION BY store_id
    ORDER BY first_name)
ORDER BY store_id, first_name, last_name;
```

ROW_NUMBER(), RANK(), and DENSE_RANK()

store_id	first_name	last_name	row_number	rank	dense_rank
1	WADE	DELVALLE	1	1	1
1	WALLACE	SLONE	2	2	2
1	WANDA	PATTERSON	3	3	3
1	WENDY	HARRISON	4	4	4
1	ZACHARY	HITE	5	5	5
2	WALTER	PERRYMAN	1	1	1
2	WARREN	SHERROD	2	2	2
2	WAYNE	TRUONG	3	3	3
2	WESLEY	BULL	4	4	4
2	WILLARD	LUMPKIN	5	5	5
2	WILLIAM	SATTERFIELD	6	6	6
2	WILLIE	HOWELL	8	7	7
2	WILLIE	MARKHAM	7	7	7
2	WILMA	RICHARDS	9	9	8
2	YOLANDA	WEAVER	10	10	9
2	YVONNE	WATKINS	11	11	10

LAG() and LEAD()

It can be useful in real life to look at previous or following rows in the window, perhaps even do some calculations between another row and the current row. In this demo, the value is not so useful, but it illustrates the functionality nicely.

```
SELECT store_id, first_name, last_name,
       LAG(last_name) OVER w,
       LEAD(last_name) OVER w
FROM customer
WHERE first_name >= 'W'
WINDOW w AS (
    PARTITION BY store_id
    ORDER BY first_name)
ORDER BY store_id, first_name, last_name;
```

LAG() and LEAD()

store_id	first_name	last_name	lag	lead
1	WADE	DELVALLE		SLONE
1	WALLACE	SLONE	DELVALLE	PATTERSON
1	WANDA	PATTERSON	SLONE	HARRISON
1	WENDY	HARRISON	PATTERSON	HITE
1	ZACHARY	HITE	HARRISON	
2	WALTER	PERRYMAN		SHERROD
2	WARREN	SHERROD	PERRYMAN	TRUONG
2	WAYNE	TRUONG	SHERROD	BULL
2	WESLEY	BULL	TRUONG	LUMPKIN
2	WILLARD	LUMPKIN	BULL	SATTERFIELD
2	WILLIAM	SATTERFIELD	LUMPKIN	MARKHAM
2	WILLIE	HOWELL	MARKHAM	RICHARDS
2	WILLIE	MARKHAM	SATTERFIELD	HOWELL
2	WILMA	RICHARDS	HOWELL	WEAVER
2	YOLANDA	WEAVER	RICHARDS	WATKINS
2	YVONNE	WATKINS	WEAVER	

`LAG()` and `LEAD()`

These functions take two extra, optional arguments:

- The first, *offset*, is how many rows back (or forward) we wish to go. If not specified, `1` is used.
- The second, *default*, is a value in case we fall off the beginning (or end) of the partition. If not specified, `NULL` is used.

These functions can also treat null values specially.

- `LAG(`*`expr`*`) RESPECT NULLS` will fetch the value for *expr* from the previous row, regardless of what it is.
- `LAG(`*`expr`*`) IGNORE NULLS` will fetch the value for *expr* from the first row going backwards that isn’t null.

NTILE()

The NTILE() function allows us to split the partition into evenly sized quantiles, or at least as evenly as possible.

```
SELECT store_id, first_name, last_name,
       NTILE(5) OVER w
FROM customer
WHERE first_name >= 'W'
WINDOW w AS (
    PARTITION BY store_id
    ORDER BY first_name)
ORDER BY store_id, first_name, last_name;
```

NTILE()

store_id	first_name	last_name	ntile
1	WADE	DELVALLE	1
1	WALLACE	SLONE	2
1	WANDA	PATTERSON	3
1	WENDY	HARRISON	4
1	ZACHARY	HITE	5
2	WALTER	PERRYMAN	1
2	WARREN	SHERROD	1
2	WAYNE	TRUONG	1
2	WESLEY	BULL	2
2	WILLARD	LUMPKIN	2
2	WILLIAM	SATTERFIELD	3
2	WILLIE	HOWELL	4
2	WILLIE	MARKHAM	3
2	WILMA	RICHARDS	4
2	YOLANDA	WEAVER	5
2	YVONNE	WATKINS	5

Quick Recap

We have seen how to write a window specification, giving it an ordering and potential partitioning key.

We have seen the following functions that all work over entire partitions:

- `ROW_NUMBER()`
- `RANK()`
- `DENSE_RANK()`
- `PERCENT_RANK()`
- `CUME_DIST()`
- `LAG(`*`expr`*`[, `*`offset`*`[, `*`default`*`]]) [RESPECT|IGNORE NULLS]`
- `LEAD(`*`expr`*`[, `*`offset`*`[, `*`default`*`]]) [RESPECT|IGNORE NULLS]`
- `NTILE(`*`n`*`)`

Window Frames

EDB

Window Frames

Window frames are defined over an ordered window partition.

This means that the **ORDER BY** clause is necessary for the frame to make any sense. The **PARTITION BY** clause is optional; without it, there will just be one partition over the entire window.

Here is the sample data for our window frames:

```
SELECT customer_id, payment_date, rental_id, amount
FROM payment
WHERE customer_id = 318
ORDER BY customer_id, payment_date;
```

Window Frames

customer_id	payment_date	rental_id	amount
318	2006-12-26 15:52:46	3376	7.99
318	2007-01-19 11:03:20	2634	2.99
318	2007-01-29 17:08:14	3337	0.99
318	2007-02-04 3:10:32	3974	2.99
318	2007-02-07 1:31:03	224	9.99
318	2007-03-19 1:46:40	7649	0.99
318	2007-03-29 0:36:16	2643	2.99
318	2007-04-04 17:59:55	10023	5.99
318	2007-04-11 2:47:11	3732	4.99
318	2007-04-17 17:48:09	7853	0.99
318	2007-04-30 5:10:59	4356	8.99
318	2007-05-24 11:52:10	14276	2.99

Window Frames

Window frames declare which rows participate in the calculation for the window function. The simplest frame is the frame that covers the entire partition. Frames cannot cross partition lines.

```
WINDOW
    w AS (
        PARTITION BY customer_id
        ORDER BY payment_date
        ROWS BETWEEN UNBOUNDED PRECEDING
                 AND UNBOUNDED FOLLOWING)
```

FIRST_VALUE() and LAST_VALUE()

Let’s reduce the frame to just 7 rows: 3 before, the current row, and three after:

```
SELECT customer_id, payment_date, rental_id, amount,
       FIRST_VALUE(rental_id) OVER w,
       LAST_VALUE(rental_id) OVER w
FROM payment
WHERE customer_id = 318
WINDOW
    w AS (
        PARTITION BY customer_id
        ORDER BY payment_date
        ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING)
ORDER BY customer_id, payment_date;
```

FIRST_VALUE() and LAST_VALUE()

customer_id	payment_date	rental_id	amount	first_value	last_value
318	2006-12-26 15:52:46	3376	7.99	3376	3974
318	2007-01-19 11:03:20	2634	2.99	3376	224
318	2007-01-29 17:08:14	3337	0.99	3376	7649
318	2007-02-04 3:10:32	3974	2.99	3376	2643
318	2007-02-07 1:31:03	224	9.99	2634	10023
318	2007-03-19 1:46:40	7649	0.99	3337	3732
318	2007-03-29 0:36:16	2643	2.99	3974	7853
318	2007-04-04 17:59:55	10023	5.99	224	4356
318	2007-04-11 2:47:11	3732	4.99	7649	14276
318	2007-04-17 17:48:09	7853	0.99	2643	14276
318	2007-04-30 5:10:59	4356	8.99	10023	14276
318	2007-05-24 11:52:10	14276	2.99	3732	14276

`FIRST_VALUE()` and `LAST_VALUE()`

These functions can also get the first (or last) value in the frame that isn’t the null value.

```
FIRST_VALUE(expr) RESPECT NULLS OVER w
FIRST_VALUE(expr) IGNORE NULLS OVER w
```

NTH_VALUE()

The FIRST_VALUE() and LAST_VALUE() functions are special cases of the NTH_VALUE() function.

```
NTH_VALUE(expr, offset) [FROM FIRST|LAST] [RESPECT|IGNORE NULLS]
```

Notice there is no default value if the offset falls off the frame!

```
FIRST_VALUE(expr) → NTH_VALUE(expr, 1) FROM FIRST
LAST_VALUE(expr) → NTH_VALUE(expr, 1) FROM LAST
```

PostgreSQL does not (yet) implement `FROM LAST` or `IGNORE NULLS`.

Quick Recap

We have learned the most basic framing for window specifications, using `ROWS` between two offsets from the current row being processed or `UNBOUNDED` to go all the way to the beginning (or end) of the frame.

We have seen the following functions that all work over frames:

- `FIRST_VALUE(`*`expr`*`) [RESPECT|IGNORE NULLS]`
- `LAST_VALUE(`*`expr`*`) [RESPECT|IGNORE NULLS]`
- `NTH_VALUE(`*`expr, offset`*`) [FROM FIRST|LAST] [RESPECT|IGNORE NULLS]`

The partition-level window functions seen earlier will completely ignore any specified frame. In fact, according to the standard, they cannot even be used over a window specification that contains a framing clause.

Aggregates Over Window Frames

EDB

Aggregates over window frames

All aggregate functions can operate over a window frame. This allows us to do interesting calculations such as running totals and rolling averages.

This is where windows really shine.

Running Totals

We can use the SUM() aggregate function over all the rows from the start of the partition up to the current row being processed. This will give us a running total.

```
SELECT customer_id, payment_date, rental_id, amount,
       SUM(amount) OVER w
FROM payment
WHERE customer_id = 318
WINDOW
    w AS (
        PARTITION BY customer_id
        ORDER BY payment_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
ORDER BY customer_id, payment_date;
```

Running Totals

customer_id	payment_date	rental_id	amount	running_total
318	2006-12-26 15:52:46	3376	7.99	7.99
318	2007-01-19 11:03:20	2634	2.99	10.98
318	2007-01-29 17:08:14	3337	0.99	11.97
318	2007-02-04 3:10:32	3974	2.99	14.96
318	2007-02-07 1:31:03	224	9.99	24.95
318	2007-03-19 1:46:40	7649	0.99	25.94
318	2007-03-29 0:36:16	2643	2.99	28.93
318	2007-04-04 17:59:55	10023	5.99	34.92
318	2007-04-11 2:47:11	3732	4.99	39.91
318	2007-04-17 17:48:09	7853	0.99	40.90
318	2007-04-30 5:10:59	4356	8.99	49.89
318	2007-05-24 11:52:10	14276	2.99	52.88

Running Totals

Let’s get the running totals per date instead of per timestamp.

```
SELECT customer_id,
       CAST(payment_date AS DATE),
       rental_id,
       amount,
       SUM(amount) OVER w
FROM payment
WHERE customer_id = 63
WINDOW
    w AS (
        PARTITION BY customer_id
        ORDER BY CAST(payment_date AS DATE)
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
ORDER BY customer_id, CAST(payment_date AS DATE)
FETCH FIRST 11 ROWS ONLY;
```

Running Totals

customer_id	payment_date	rental_id	amount	running_total
63	2007-01-21	3923	8.99	8.99
63	2007-02-07	9795	0.99	9.98
63	2007-02-25	15060	5.99	15.97
63	2007-03-06	6847	8.99	24.96
63	2007-03-10	5585	6.99	31.95
63	2007-03-11	13624	8.99	40.94
63	2007-03-11	9007	0.99	41.93
63	2007-03-16	4587	4.99	46.92
63	2007-03-16	13089	0.99	47.91
63	2007-03-17	5832	4.99	52.90
63	2007-03-21	9549	3.99	56.89

What is going on *here*?

Running Totals

The problem is we need to look *ahead* and include all of the rows past the current row where the value is the same. This is done by changing **ROWS** to **RANGE**.

```
SELECT customer_id,
       CAST(payment_date AS DATE),
       rental_id,
       amount,
       SUM(amount) OVER w
FROM payment
WHERE customer_id = 63
WINDOW
    w AS (
        PARTITION BY customer_id
        ORDER BY CAST(payment_date AS DATE)
        RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
ORDER BY customer_id, CAST(payment_date AS DATE)
FETCH FIRST 11 ROWS ONLY;
```

Running Totals

customer_id	payment_date	rental_id	amount	running_total
63	2007-01-21	3923	8.99	8.99
63	2007-02-07	9795	0.99	9.98
63	2007-02-25	15060	5.99	15.97
63	2007-03-06	6847	8.99	24.96
63	2007-03-10	5585	6.99	31.95
63	2007-03-11	13624	8.99	41.93
63	2007-03-11	9007	0.99	41.93
63	2007-03-16	4587	4.99	47.91
63	2007-03-16	13089	0.99	47.91
63	2007-03-17	5832	4.99	52.90
63	2007-03-21	9549	3.99	56.89

Rolling Averages

Let's get more advanced! Let's get the rolling averages for 1 day, 3 days, and 7 days for all payments towards the beginning of our data.

First, let's get the payments per day:

```
SELECT CAST(payment_date AS DATE) AS payment_date,
       SUM(amount) AS amount
FROM payment
GROUP BY CAST(payment_date AS DATE)
ORDER BY CAST(payment_date AS DATE)
OFFSET 10
FETCH NEXT 20 ROWS ONLY
```

Rolling Averages

payment_date	amount
2006-12-05	24.92
2006-12-06	51.89
2006-12-07	51.88
2006-12-08	55.87
2006-12-09	15.96
2006-12-10	54.88
2006-12-11	66.85
2006-12-12	58.9
2006-12-13	45.87
2006-12-14	46.87
2006-12-15	50.87
2006-12-16	78.85
2006-12-17	67.8
2006-12-18	53.86
2006-12-19	94.8
2006-12-20	117.72
2006-12-21	111.76
2006-12-22	66.85
2006-12-23	138.7
2006-12-24	100.76

Rolling Averages

Now we can put our windows **on top of** the existing aggregates!

```
SELECT CAST(payment_date AS DATE) AS payment_date,
       SUM(amount) AS amount,
       ROUND(AVG(SUM(amount)) OVER w3, 3) AS "3-day average",
       ROUND(AVG(SUM(amount)) OVER w7, 3) AS "7-day average"
FROM payment
GROUP BY CAST(payment_date AS DATE)
WINDOW
    w3 AS (
        ORDER BY CAST(payment_date AS DATE)
        RANGE BETWEEN '3 days' PRECEDING AND CURRENT ROW),
    w7 AS (
        ORDER BY CAST(payment_date AS DATE)
        RANGE BETWEEN '7 days' PRECEDING AND CURRENT ROW)
ORDER BY CAST(payment_date AS DATE)
OFFSET 10
FETCH NEXT 20 ROWS ONLY
```

Rolling Averages

payment_date	amount	3-day average	7-day average
2006-12-05	24.92	40.645	43.02
2006-12-06	51.89	44.643	45.39
2006-12-07	51.88	45.39	49.63
2006-12-08	55.87	46.14	47.388
2006-12-09	15.96	43.9	42.273
2006-12-10	54.88	44.648	44.645
2006-12-11	66.85	48.39	46.89
2006-12-12	58.9	49.148	47.644
2006-12-13	45.87	56.625	50.263
2006-12-14	46.87	54.623	49.635
2006-12-15	50.87	50.628	49.509
2006-12-16	78.85	55.615	52.381
2006-12-17	67.8	61.098	58.861
2006-12-18	53.86	62.845	58.734
2006-12-19	94.8	73.828	62.228
2006-12-20	117.72	83.545	69.58
2006-12-21	111.76	94.535	77.816
2006-12-22	66.85	97.783	80.314
2006-12-23	138.7	108.758	91.293
2006-12-24	100.76	104.518	94.031

Extensions and Abbreviations

EDB

Extensions and Abbreviations

That rolling averages query was *a lot*! Even for the overly verbose SQL it is a lot.

```
SELECT CAST(payment_date AS DATE) AS payment_date,
       SUM(amount) AS amount,
       ROUND(AVG(SUM(amount)) OVER w3, 3) AS "3-day average",
       ROUND(AVG(SUM(amount)) OVER w7, 3) AS "7-day average"
FROM payment
GROUP BY CAST(payment_date AS DATE)
WINDOW
    w3 AS (
        ORDER BY CAST(payment_date AS DATE)
        RANGE BETWEEN '3 days' PRECEDING AND CURRENT ROW),
    w7 AS (
        ORDER BY CAST(payment_date AS DATE)
        RANGE BETWEEN '7 days' PRECEDING AND CURRENT ROW)
ORDER BY CAST(payment_date AS DATE)
OFFSET 10
FETCH NEXT 20 ROWS ONLY
```

Extensions and Abbreviations

We can separate the partitioning and ordering from the framing, like so:

```
WINDOW
    w3 AS (
        ORDER BY CAST(payment_date AS DATE)
        RANGE BETWEEN '3 days' PRECEDING AND CURRENT ROW),
    w7 AS (
        ORDER BY CAST(payment_date AS DATE)
        RANGE BETWEEN '7 days' PRECEDING AND CURRENT ROW)
```

```
WINDOW
    w  AS (ORDER BY CAST(payment_date AS DATE)),
    w3 AS (w RANGE BETWEEN '3 days' PRECEDING AND CURRENT ROW),
    w7 AS (w RANGE BETWEEN '7 days' PRECEDING AND CURRENT ROW)
```

Extensions and Abbreviations

If the frame we want ends at the current row, we can omit the **`BETWEEN`** and just specify the start:

```
WINDOW
    w  AS (ORDER BY CAST(payment_date AS DATE)),
    w3 AS (w RANGE BETWEEN '3 days' PRECEDING AND CURRENT ROW),
    w7 AS (w RANGE BETWEEN '7 days' PRECEDING AND CURRENT ROW)
```

```
WINDOW
    w  AS (ORDER BY CAST(payment_date AS DATE)),
    w3 AS (w RANGE '3 days' PRECEDING),
    w7 AS (w RANGE '7 days' PRECEDING)
```

Extensions and Abbreviations

Here is our final version, which is quite concise and much easier to read.

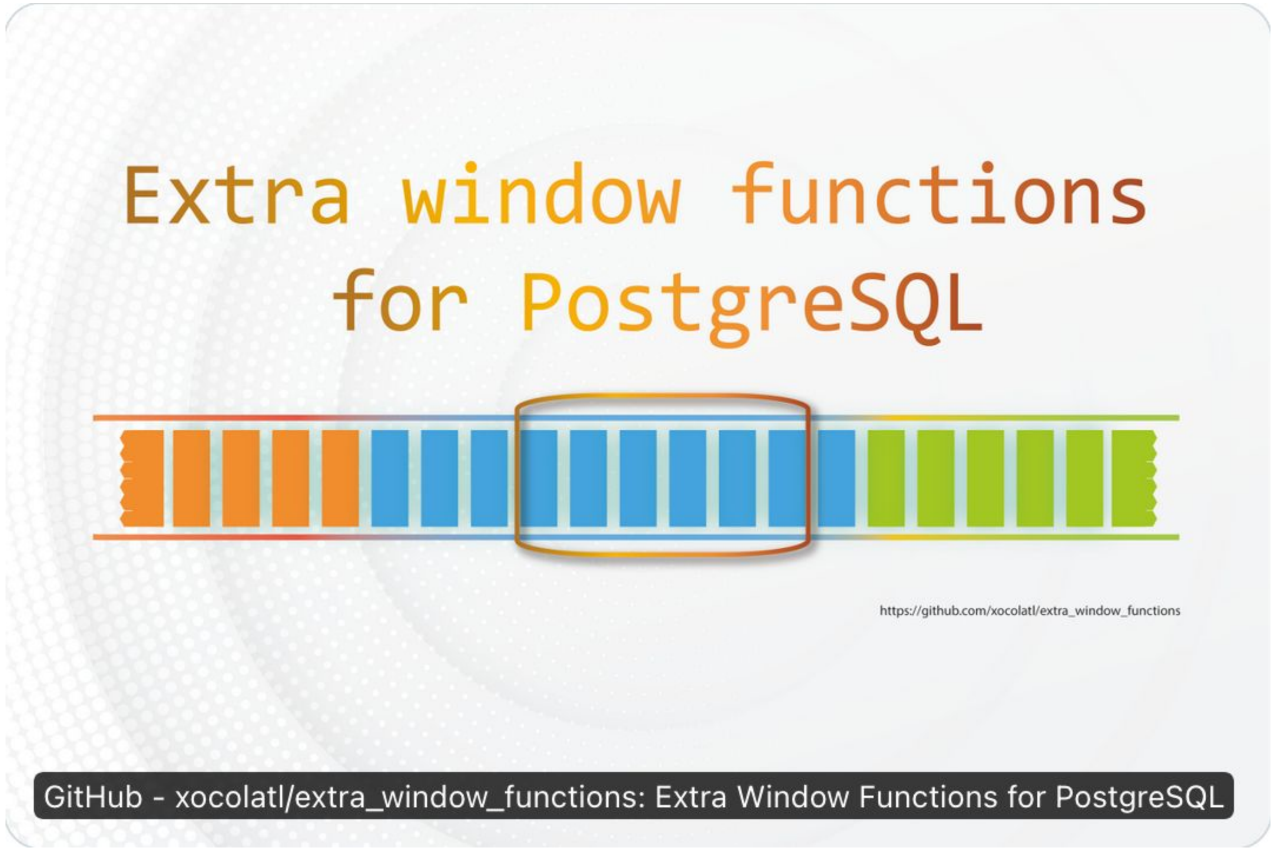
```
SELECT CAST(payment_date AS DATE) AS payment_date,
       SUM(amount) AS amount,
       ROUND(AVG(SUM(amount)) OVER w3, 3) AS "3-day average",
       ROUND(AVG(SUM(amount)) OVER w7, 3) AS "7-day average"
FROM payment
GROUP BY CAST(payment_date AS DATE)
WINDOW
    w  AS (ORDER BY CAST(payment_date AS DATE)),
    w3 AS (w RANGE '3 days' PRECEDING),
    w7 AS (w RANGE '7 days' PRECEDING)
ORDER BY CAST(payment_date AS DATE)
OFFSET 10
FETCH NEXT 20 ROWS ONLY
```

Creating Your Own

EDB

Creating Your Own

It easy to create your own custom aggregate and use it over a window frame. It is much harder to create your own custom window function. Here is a git repository that shows how it can be done.


Extra window functions for PostgreSQL
https://github.com/xocolatl/extra_window_functions
GitHub - xocolatl/extra_window_functions: Extra Window Functions for PostgreSQL

Please leave feedback!

Vik Fearing, EDB