Postgres Window Magic

Bruce Momjian

This presentation explains the many window function facilities and how they can be used to produce useful SQL query results.

 http://momjian.us/presentations

Last updated: May, 2017

Outline

1. Introduction to window functions
2. Window function syntax
3. Window syntax with generic aggregates
4. Window-specific functions
5. Window function examples
6. Considerations

1. Introduction to Window Functions

https://www.flickr.com/photos/conalg/

Postgres Data Analytics Features

- Aggregates
- Optimizer
- Server-side languages, e.g. PL/R
- **Window functions**
- Bitmap heap scans
- Tablespaces
- Data partitioning
- Materialized views
- Common table expressions (CTE)
- BRIN indexes
- GROUPING SETS, ROLLUP, CUBE
- Parallelism
- Sharding (in progress)

What Is a Window Function?

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. Behind the scenes, the window function is able to access more than just the current row of the query result.

https://www.postgresql.org/docs/current/static/tutorial-window.html

Keep Your Eye on the Red (Text)

https://www.flickr.com/photos/alltheaces/

Count to Ten

```
SELECT *
FROM generate_series(1, 10) AS f(x);
```

```
 x
----
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
```

All the queries used in this presentation are available at http://momjian.us/main/writings/pgsql/window.sql.

Simplest Window Function

```
SELECT x, SUM(x) OVER ()
FROM generate_series(1, 10) AS f(x);
```

```
 x  | sum
----+-----
  1 |  55
  2 |  55
  3 |  55
  4 |  55
  5 |  55
  6 |  55
  7 |  55
  8 |  55
  9 |  55
 10 |  55
```

Two OVER Clauses

```
SELECT x, COUNT(x) OVER (), SUM(x) OVER ()
FROM generate_series(1, 10) AS f(x);
```

```
 x  | count | sum
----+-------+-----
  1 |    10 |  55
  2 |    10 |  55
  3 |    10 |  55
  4 |    10 |  55
  5 |    10 |  55
  6 |    10 |  55
  7 |    10 |  55
  8 |    10 |  55
  9 |    10 |  55
 10 |    10 |  55
```

WINDOW Clause

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS ();
```

```
 x  | count | sum
----+-------+-----
  1 |    10 |  55
  2 |    10 |  55
  3 |    10 |  55
  4 |    10 |  55
  5 |    10 |  55
  6 |    10 |  55
  7 |    10 |  55
  8 |    10 |  55
  9 |    10 |  55
 10 |    10 |  55
```

Let's See the Defaults

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (RANGE BETWEEN
             UNBOUNDED PRECEDING AND CURRENT ROW);
```

x	count	sum
1	10	55
2	10	55
3	10	55
4	10	55
5	10	55
6	10	55
7	10	55
8	10	55
9	10	55
10	10	55

2. Window Function Syntax

https://www.flickr.com/photos/bgreenlee/

Window Syntax

```
WINDOW (
      [PARTITION BY …]
      [ORDER BY …]
      [
        { RANGE | ROWS }
        { frame_start | BETWEEN frame_start AND frame_end }
      ]
)
```

where *frame_start* and *frame_end* can be:

- UNBOUNDED PRECEDING
- *value* PRECEDING
- CURRENT ROW
- *value* FOLLOWING
- UNBOUNDED FOLLOWING

Bracketed clauses are optional, braces are selected.

https://www.postgresql.org/docs/current/static/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS

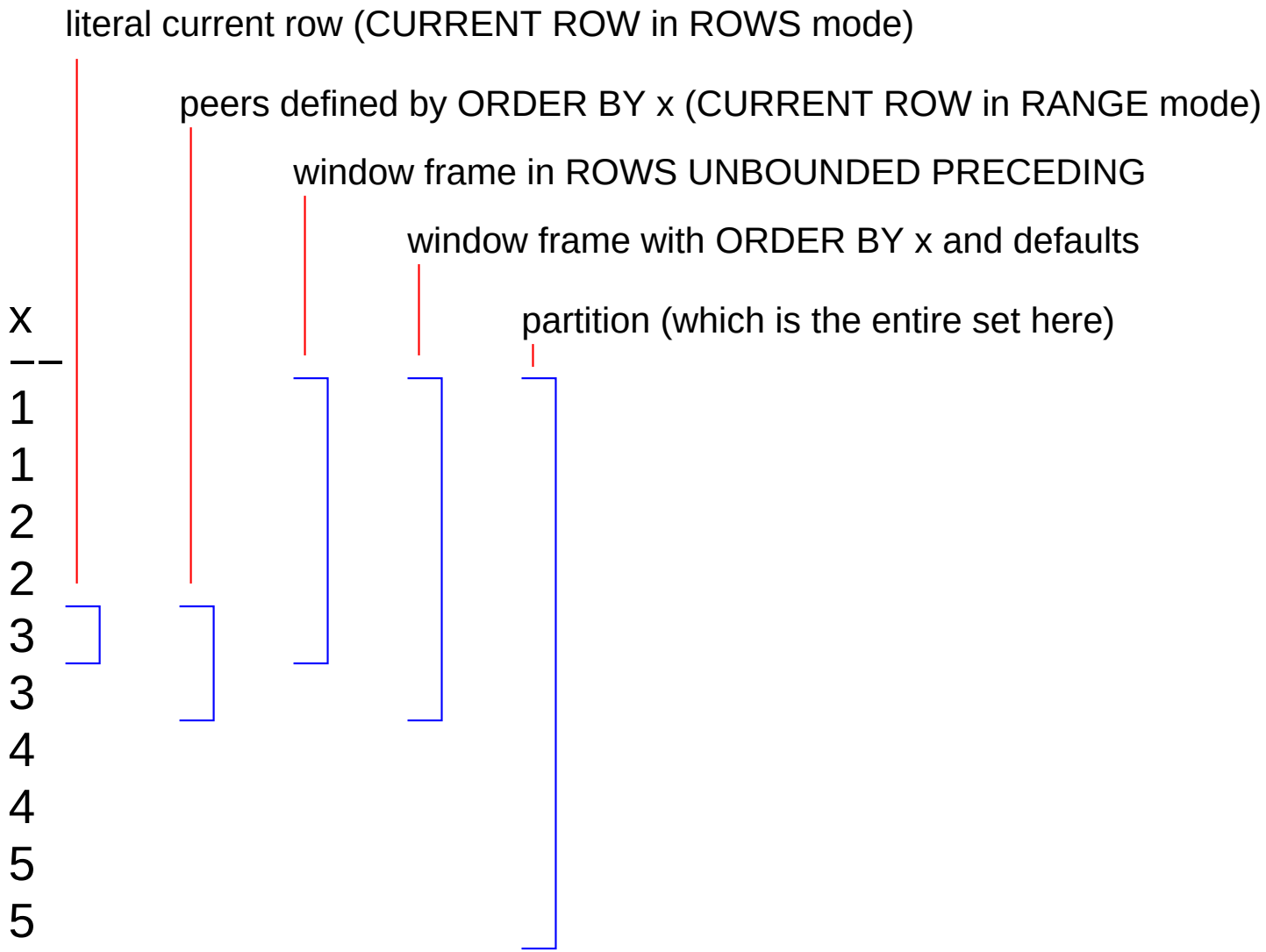
What Are the Defaults?

```
(RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

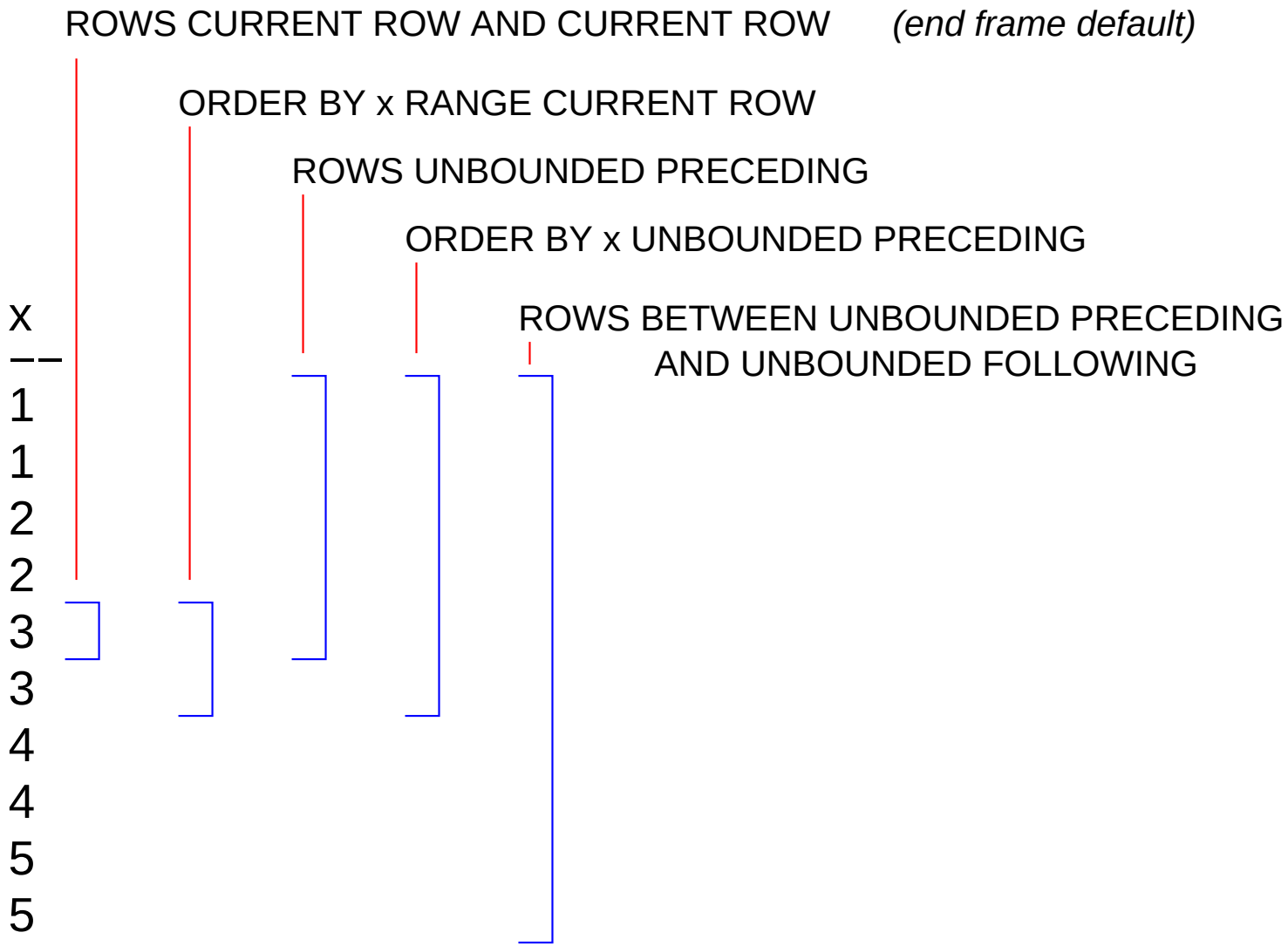
- No PARTITION BY (the set is a single partition)
- No ORDER BY (all rows are peers of CURRENT ROW)
- RANGE, not ROWS (CURRENT ROW includes all peers)

Since PARTITION BY and ORDER BY are not defaults but RANGE is the default, CURRENT ROW defaults to representing all rows.

Visual Window Terms



SQL for Window Frames



3. Window Syntax with Generic Aggregates

https://www.flickr.com/photos/azparrot/

Back to the Last Query

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (RANGE BETWEEN
               UNBOUNDED PRECEDING AND CURRENT ROW);
```

x	count	sum
1	10	55
2	10	55
3	10	55
4	10	55
5	10	55
6	10	55
7	10	55
8	10	55
9	10	55
10	10	55

ROWS Instead of RANGE

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ROWS BETWEEN
               UNBOUNDED PRECEDING AND CURRENT ROW);
```

x	count	sum
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
6	6	21
7	7	28
8	8	36
9	9	45
10	10	55

Default End Frame (Current Row)

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ROWS UNBOUNDED PRECEDING);
```

x	count	sum
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
6	6	21
7	7	28
8	8	36
9	9	45
10	10	55

Only Current Row

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ROWS BETWEEN
               CURRENT ROW AND CURRENT ROW);
```

x	count	sum
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5
6	1	6
7	1	7
8	1	8
9	1	9
10	1	10

Use Defaults

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ROWS CURRENT ROW);
```

```
 x  | count | sum
----+-------+-----
  1 |     1 |   1
  2 |     1 |   2
  3 |     1 |   3
  4 |     1 |   4
  5 |     1 |   5
  6 |     1 |   6
  7 |     1 |   7
  8 |     1 |   8
  9 |     1 |   9
 10 |     1 |  10
```

Unbounded Following

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ROWS BETWEEN
                CURRENT ROW AND UNBOUNDED FOLLOWING);
```

x	count	sum
1	10	55
2	9	54
3	8	52
4	7	49
5	6	45
6	5	40
7	4	34
8	3	27
9	2	19
10	1	10

PRECEDING

```
SELECT x, COUNT(*) OVER w, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ROWS BETWEEN
               1 PRECEDING AND CURRENT ROW);
```

x	count	count	sum
1	1	1	1
2	2	2	3
3	2	2	5
4	2	2	7
5	2	2	9
6	2	2	11
7	2	2	13
8	2	2	15
9	2	2	17
10	2	2	19

PRECEDING ignores nonexistent rows; they are not NULLs.

Use FOLLOWING

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ROWS BETWEEN
                CURRENT ROW AND 1 FOLLOWING);
```

x	count	sum
1	2	3
2	2	5
3	2	7
4	2	9
5	2	11
6	2	13
7	2	15
8	2	17
9	2	19
10	1	10

3 Preceding

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ROWS BETWEEN
                3 PRECEDING AND CURRENT ROW);
```

x	count	sum
1	1	1
2	2	3
3	3	6
4	4	10
5	4	14
6	4	18
7	4	22
8	4	26
9	4	30
10	4	34

Order By

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ORDER BY x);
```

```
 x  | count | sum
----+-------+-----
  1 |     1 |   1
  2 |     2 |   3
  3 |     3 |   6
  4 |     4 |  10
  5 |     5 |  15
  6 |     6 |  21
  7 |     7 |  28
  8 |     8 |  36
  9 |     9 |  45
 10 |    10 |  55
```

Current Row peers are rows with equal values for order by columns, or all partition rows if order by is not specified.

Default Frame Specified

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ORDER BY x RANGE BETWEEN
             UNBOUNDED PRECEDING AND CURRENT ROW);
```

x	count	sum
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
6	6	21
7	7	28
8	8	36
9	9	45
10	10	55

Only Current Row

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_series(1, 10) AS f(x)
WINDOW w AS (ORDER BY x RANGE CURRENT ROW);
```

```
 x  | count | sum
----+-------+-----
  1 |     1 |   1
  2 |     1 |   2
  3 |     1 |   3
  4 |     1 |   4
  5 |     1 |   5
  6 |     1 |   6
  7 |     1 |   7
  8 |     1 |   8
  9 |     1 |   9
 10 |     1 |  10
```

Create Table with Duplicates

```
CREATE TABLE generate_1_to_5_x2 AS
        SELECT ceil(x/2.0) AS x
        FROM generate_series(1, 10) AS f(x);

SELECT * FROM generate_1_to_5_x2;

 x
---
 1
 1
 2
 2
 3
 3
 4
 4
 5
 5
```

Empty Window Specification

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS ();
```

```
 x | count | sum
---+-------+-----
 1 |    10 |  30
 1 |    10 |  30
 2 |    10 |  30
 2 |    10 |  30
 3 |    10 |  30
 3 |    10 |  30
 4 |    10 |  30
 4 |    10 |  30
 5 |    10 |  30
 5 |    10 |  30
```

RANGE With Duplicates

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x);
```

```
 x | count | sum
---+-------+-----
 1 |     2 |   2
 1 |     2 |   2
 2 |     4 |   6
 2 |     4 |   6
 3 |     6 |  12
 3 |     6 |  12
 4 |     8 |  20
 4 |     8 |  20
 5 |    10 |  30
 5 |    10 |  30
```

Show Defaults

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x RANGE BETWEEN
             UNBOUNDED PRECEDING AND CURRENT ROW);
```

x	count	sum
1	2	2
1	2	2
2	4	6
2	4	6
3	6	12
3	6	12
4	8	20
4	8	20
5	10	30
5	10	30

Rows

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x ROWS BETWEEN
             UNBOUNDED PRECEDING AND CURRENT ROW);
```

x	count	sum
1	1	1
1	2	2
2	3	4
2	4	6
3	5	9
3	6	12
4	7	16
4	8	20
5	9	25
5	10	30

RANGE on CURRENT ROW

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x RANGE CURRENT ROW);
```

```
 x | count | sum
---+-------+-----
 1 |     2 |   2
 1 |     2 |   2
 2 |     2 |   4
 2 |     2 |   4
 3 |     2 |   6
 3 |     2 |   6
 4 |     2 |   8
 4 |     2 |   8
 5 |     2 |  10
 5 |     2 |  10
```

Rows on Current Row

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x ROWS CURRENT ROW);
```

```
 x | count | sum
---+-------+-----
 1 |     1 |   1
 1 |     1 |   1
 2 |     1 |   2
 2 |     1 |   2
 3 |     1 |   3
 3 |     1 |   3
 4 |     1 |   4
 4 |     1 |   4
 5 |     1 |   5
 5 |     1 |   5
```

Partition By

```
SELECT x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (PARTITION BY x);
```

```
 x | count | sum
---+-------+-----
 1 |     2 |   2
 1 |     2 |   2
 2 |     2 |   4
 2 |     2 |   4
 3 |     2 |   6
 3 |     2 |   6
 4 |     2 |   8
 4 |     2 |   8
 5 |     2 |  10
 5 |     2 |  10
```

Same as RANGE CURRENT ROW because the partition matches the window frame.

Create Two Partitions

```
SELECT int4(x > 2), x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (PARTITION BY x > 2);
```

```
 int4 | x | count | sum
------+---+-------+-----
    0 | 1 |     4 |   6
    0 | 1 |     4 |   6
    0 | 2 |     4 |   6
    0 | 2 |     4 |   6
    1 | 3 |     6 |  24
    1 | 3 |     6 |  24
    1 | 4 |     6 |  24
    1 | 4 |     6 |  24
    1 | 5 |     6 |  24
    1 | 5 |     6 |  24
```

Order By

```
SELECT int4(x > 2), x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (PARTITION BY x > 2 ORDER BY x);
```

```
 int4 | x | count | sum
------+---+-------+-----
    0 | 1 |     2 |   2
    0 | 1 |     2 |   2
    0 | 2 |     4 |   6
    0 | 2 |     4 |   6
    1 | 3 |     2 |   6
    1 | 3 |     2 |   6
    1 | 4 |     4 |  14
    1 | 4 |     4 |  14
    1 | 5 |     6 |  24
    1 | 5 |     6 |  24
```

Show Defaults

```
SELECT int4(x > 2), x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (PARTITION BY x > 2 ORDER BY x RANGE BETWEEN
             UNBOUNDED PRECEDING AND CURRENT ROW);
```

int4	x	count	sum
0	1	2	2
0	1	2	2
0	2	4	6
0	2	4	6
1	3	2	6
1	3	2	6
1	4	4	14
1	4	4	14
1	5	6	24
1	5	6	24

Rows

```
SELECT int4(x > 2), x, COUNT(x) OVER w, SUM(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (PARTITION BY x > 2 ORDER BY x ROWS BETWEEN
             UNBOUNDED PRECEDING AND CURRENT ROW);
```

int4	x	count	sum
0	1	1	1
0	1	2	2
0	2	3	4
0	2	4	6
1	3	1	3
1	3	2	6
1	4	3	10
1	4	4	14
1	5	5	19
1	5	6	24

4. Window-Specific Functions

https://www.flickr.com/photos/michaeljohnbutton/

ROW_NUMBER

```
SELECT x, ROW_NUMBER() OVER w
FROM generate_1_to_5_x2
WINDOW w AS ();
```

```
 x | row_number
---+------------
 1 |          1
 1 |          2
 2 |          3
 2 |          4
 3 |          5
 3 |          6
 4 |          7
 4 |          8
 5 |          9
 5 |         10
```

ROW_NUMBER takes no arguments and operates on partitions, not window frames. https://www.postgresql.org/docs/current/static/functions-window.html

Lag

```
SELECT x, LAG(x, 1) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x);
```

```
 x |  lag
---+--------
 1 | (null)
 1 |      1
 2 |      1
 2 |      2
 3 |      2
 3 |      3
 4 |      3
 4 |      4
 5 |      4
 5 |      5
```

LAG(2)

```
SELECT x, LAG(x, 2) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x);
```

```
 x |  lag
---+--------
 1 | (null)
 1 | (null)
 2 |      1
 2 |      1
 3 |      2
 3 |      2
 4 |      3
 4 |      3
 5 |      4
 5 |      4
```

LAG and LEAD

```
SELECT x, LAG(x, 2) OVER w, LEAD(x, 2) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x);
```

```
 x |  lag   |  lead
---+--------+--------
 1 | (null) |      2
 1 | (null) |      2
 2 |      1 |      3
 2 |      1 |      3
 3 |      2 |      4
 3 |      2 |      4
 4 |      3 |      5
 4 |      3 |      5
 5 |      4 | (null)
 5 |      4 | (null)
```

These operate on partitions. Defaults can be specified for nonexistent rows.

First_value and Last_value

```
SELECT x, FIRST_VALUE(x) OVER w, LAST_VALUE(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x);
```

```
 x | first_value | last_value
---+-------------+------------
 1 |           1 |          1
 1 |           1 |          1
 2 |           1 |          2
 2 |           1 |          2
 3 |           1 |          3
 3 |           1 |          3
 4 |           1 |          4
 4 |           1 |          4
 5 |           1 |          5
 5 |           1 |          5
```

These operate on window frames.

UNBOUNDED Window Frame

```
SELECT x, FIRST_VALUE(x) OVER w, LAST_VALUE(x) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x ROWS BETWEEN
             UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING);
```

x	first_value	last_value
1	1	5
1	1	5
2	1	5
2	1	5
3	1	5
3	1	5
4	1	5
4	1	5
5	1	5
5	1	5

NTH_VALUE

```
SELECT x, NTH_VALUE(x, 3) OVER w, NTH_VALUE(x, 7) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x);
```

```
 x | nth_value | nth_value
---+-----------+-----------
 1 |    (null) |    (null)
 1 |    (null) |    (null)
 2 |         2 |    (null)
 2 |         2 |    (null)
 3 |         2 |    (null)
 3 |         2 |    (null)
 4 |         2 |         4
 4 |         2 |         4
 5 |         2 |         4
 5 |         2 |         4
```

This operates on window frames.

Show Defaults

```
SELECT x, NTH_VALUE(x, 3) OVER w, NTH_VALUE(x, 7) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x RANGE BETWEEN
             UNBOUNDED PRECEDING AND CURRENT ROW);
```

```
 x | nth_value | nth_value
---+-----------+-----------
 1 |    (null) |    (null)
 1 |    (null) |    (null)
 2 |         2 |    (null)
 2 |         2 |    (null)
 3 |         2 |    (null)
 3 |         2 |    (null)
 4 |         2 |         4
 4 |         2 |         4
 5 |         2 |         4
 5 |         2 |         4
```

UNBOUNDED Window Frame

```
SELECT x, NTH_VALUE(x, 3) OVER w, NTH_VALUE(x, 7) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x ROWS BETWEEN
             UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING);
```

```
 x | nth_value | nth_value
---+-----------+-----------
 1 |         2 |         4
 1 |         2 |         4
 2 |         2 |         4
 2 |         2 |         4
 3 |         2 |         4
 3 |         2 |         4
 4 |         2 |         4
 4 |         2 |         4
 5 |         2 |         4
 5 |         2 |         4
```

RANK and DENSE_RANK

```
SELECT x, RANK() OVER w, DENSE_RANK() OVER w
FROM generate_1_to_5_x2
WINDOW w AS ();
```

```
 x | rank | dense_rank
---+------+------------
 1 |    1 |          1
 1 |    1 |          1
 2 |    1 |          1
 2 |    1 |          1
 3 |    1 |          1
 3 |    1 |          1
 4 |    1 |          1
 4 |    1 |          1
 5 |    1 |          1
 5 |    1 |          1
```

These operate on CURRENT ROW peers in the partition.

Show Defaults

```
SELECT x, RANK() OVER w, DENSE_RANK() OVER w
FROM generate_1_to_5_x2
WINDOW w AS (RANGE BETWEEN
             UNBOUNDED PRECEDING AND CURRENT ROW);
```

```
 x | rank | dense_rank
---+------+------------
 1 |    1 |          1
 1 |    1 |          1
 2 |    1 |          1
 2 |    1 |          1
 3 |    1 |          1
 3 |    1 |          1
 4 |    1 |          1
 4 |    1 |          1
 5 |    1 |          1
 5 |    1 |          1
```

Rows

```
SELECT x, RANK() OVER w, DENSE_RANK() OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ROWS BETWEEN
               UNBOUNDED PRECEDING AND CURRENT ROW);
```

```
 x | rank | dense_rank
---+------+------------
 1 |    1 |          1
 1 |    1 |          1
 2 |    1 |          1
 2 |    1 |          1
 3 |    1 |          1
 3 |    1 |          1
 4 |    1 |          1
 4 |    1 |          1
 5 |    1 |          1
 5 |    1 |          1
```

Operates on Peers, so Needs ORDER BY

```
SELECT x, RANK() OVER w, DENSE_RANK() OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x);
```

```
 x | rank | dense_rank
---+------+------------
 1 |    1 |          1
 1 |    1 |          1
 2 |    3 |          2
 2 |    3 |          2
 3 |    5 |          3
 3 |    5 |          3
 4 |    7 |          4
 4 |    7 |          4
 5 |    9 |          5
 5 |    9 |          5
```

PERCENT_RANK, CUME_DIST, NTILE

```
SELECT x, (PERCENT_RANK() OVER w)::numeric(10, 2),
       (CUME_DIST() OVER w)::numeric(10, 2), NTILE(3) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (ORDER BY x);
```

x	percent_rank	cume_dist	ntile
1	0.00	0.20	1
1	0.00	0.20	1
2	0.22	0.40	1
2	0.22	0.40	1
3	0.44	0.60	2
3	0.44	0.60	2
4	0.67	0.80	2
4	0.67	0.80	3
5	0.89	1.00	3
5	0.89	1.00	3

PERCENT_RANK is ratio of rows less than current row, excluding current row. CUME_DIST is ratio of rows <= current row.

Partition By

```
SELECT int4(x > 2), x, RANK() OVER w, DENSE_RANK() OVER w
FROM generate_1_to_5_x2
WINDOW w AS (PARTITION BY x > 2 ORDER BY x)
ORDER BY 1,2;
```

```
 int4 | x | rank | dense_rank
------+---+------+------------
    0 | 1 |    1 |          1
    0 | 1 |    1 |          1
    0 | 2 |    3 |          2
    0 | 2 |    3 |          2
    1 | 3 |    1 |          1
    1 | 3 |    1 |          1
    1 | 4 |    3 |          2
    1 | 4 |    3 |          2
    1 | 5 |    5 |          3
    1 | 5 |    5 |          3
```

PARTITION BY and Other Rank Functions

```
SELECT int4(x > 2), x, (PERCENT_RANK() OVER w)::numeric(10,2),
       (CUME_DIST() OVER w)::numeric(10,2), NTILE(3) OVER w
FROM generate_1_to_5_x2
WINDOW w AS (PARTITION BY x > 2 ORDER BY x)
ORDER BY 1,2;
```

int4	x	percent_rank	cume_dist	ntile
0	1	0.00	0.50	1
0	1	0.00	0.50	1
0	2	0.67	1.00	2
0	2	0.67	1.00	3
1	3	0.00	0.33	1
1	3	0.00	0.33	1
1	4	0.40	0.67	2
1	4	0.40	0.67	2
1	5	0.80	1.00	3
1	5	0.80	1.00	3

5. Window Function Examples

https://www.flickr.com/photos/fishywang/

Create *emp* Table and Populate

```
CREATE TABLE emp (
    id SERIAL,
    name TEXT NOT NULL,
    department TEXT,
    salary NUMERIC(10, 2)
);

INSERT INTO emp (name, department, salary) VALUES
        ('Andy', 'Shipping', 5400),
        ('Betty', 'Marketing', 6300),
        ('Tracy', 'Shipping', 4800),
        ('Mike', 'Marketing', 7100),
        ('Sandy', 'Sales', 5400),
        ('James', 'Shipping', 6600),
        ('Carol', 'Sales', 4600);
```

https://www.postgresql.org/docs/current/static/tutorial-window.html

Emp Table

```
SELECT * FROM emp ORDER BY id;

 id | name   | department | salary
----+--------+------------+---------
  1 | Andy   | Shipping   | 5400.00
  2 | Betty  | Marketing  | 6300.00
  3 | Tracy  | Shipping   | 4800.00
  4 | Mike   | Marketing  | 7100.00
  5 | Sandy  | Sales      | 5400.00
  6 | James  | Shipping   | 6600.00
  7 | Carol  | Sales      | 4600.00
```

Generic Aggregates

```
SELECT COUNT(*), SUM(salary),
       round(AVG(salary), 2) AS avg
FROM emp;

 count |    sum     |   avg
-------+------------+---------
     7 | 40200.00 | 5742.86
```

Group By

```
SELECT department, COUNT(*), SUM(salary),
       round(AVG(salary), 2) AS avg
FROM emp
GROUP BY department
ORDER BY department;
```

```
 department | count |   sum    |   avg
------------+-------+----------+---------
 Marketing  |     2 | 13400.00 | 6700.00
 Sales      |     2 | 10000.00 | 5000.00
 Shipping   |     3 | 16800.00 | 5600.00
```

ROLLUP

```
SELECT department, COUNT(*), SUM(salary),
       round(AVG(salary), 2) AS avg
FROM emp
GROUP BY ROLLUP(department)
ORDER BY department;
```

```
 department | count |   sum    |   avg
------------+-------+----------+---------
 Marketing  |     2 | 13400.00 | 6700.00
 Sales      |     2 | 10000.00 | 5000.00
 Shipping   |     3 | 16800.00 | 5600.00
 (null)     |     7 | 40200.00 | 5742.86
```

Emp.name and *Salary*

```
SELECT name, salary
FROM emp
ORDER BY salary DESC;
```

```
 name  | salary
-------+---------
 Mike  | 7100.00
 James | 6600.00
 Betty | 6300.00
 Andy  | 5400.00
 Sandy | 5400.00
 Tracy | 4800.00
 Carol | 4600.00
```

Over

```
SELECT name, salary, SUM(salary) OVER ()
FROM emp
ORDER BY salary DESC;
```

```
 name  | salary  |   sum
-------+---------+----------
 Mike  | 7100.00 | 40200.00
 James | 6600.00 | 40200.00
 Betty | 6300.00 | 40200.00
 Andy  | 5400.00 | 40200.00
 Sandy | 5400.00 | 40200.00
 Tracy | 4800.00 | 40200.00
 Carol | 4600.00 | 40200.00
```

Cumulative Totals Using ORDER BY

```
SELECT name, salary,
       SUM(salary) OVER (ORDER BY salary DESC)
FROM emp
ORDER BY salary DESC;
```

```
 name  | salary  |   sum
-------+---------+----------
 Mike  | 7100.00 |  7100.00
 James | 6600.00 | 13700.00
 Betty | 6300.00 | 20000.00
 Andy  | 5400.00 | 30800.00
 Sandy | 5400.00 | 30800.00
 Tracy | 4800.00 | 35600.00
 Carol | 4600.00 | 40200.00
```

Cumulative totals are often useful for time-series rows.

Window AVG

```
SELECT name, salary,
       round(AVG(salary) OVER (), 2) AS avg
FROM emp
ORDER BY salary DESC;
```

```
 name  | salary  |   avg
-------+---------+---------
 Mike  | 7100.00 | 5742.86
 James | 6600.00 | 5742.86
 Betty | 6300.00 | 5742.86
 Andy  | 5400.00 | 5742.86
 Sandy | 5400.00 | 5742.86
 Tracy | 4800.00 | 5742.86
 Carol | 4600.00 | 5742.86
```

Difference Compared to Average

```
SELECT name, salary,
       round(AVG(salary) OVER (), 2) AS avg,
       round(salary - AVG(salary) OVER (), 2) AS diff_avg
FROM emp
ORDER BY salary DESC;
```

name	salary	avg	diff_avg
Mike	7100.00	5742.86	1357.14
James	6600.00	5742.86	857.14
Betty	6300.00	5742.86	557.14
Andy	5400.00	5742.86	-342.86
Sandy	5400.00	5742.86	-342.86
Tracy	4800.00	5742.86	-942.86
Carol	4600.00	5742.86	-1142.86

Compared to the Next Value

```
SELECT name, salary,
       salary - LEAD(salary, 1) OVER
                (ORDER BY salary DESC) AS diff_next
FROM emp
ORDER BY salary DESC;
```

```
 name  | salary  | diff_next
-------+---------+-----------
 Mike  | 7100.00 |    500.00
 James | 6600.00 |    300.00
 Betty | 6300.00 |    900.00
 Sandy | 5400.00 |      0.00
 Andy  | 5400.00 |    600.00
 Tracy | 4800.00 |    200.00
 Carol | 4600.00 |    (null)
```

Compared to Lowest-Paid Employee

```
SELECT name, salary,
       salary - LAST_VALUE(salary) OVER w AS more,
       round((salary - LAST_VALUE(salary) OVER w) /
        LAST_VALUE(salary) OVER w * 100) AS pct_more
FROM emp
WINDOW w AS (ORDER BY salary DESC ROWS BETWEEN
             UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
ORDER BY salary DESC;
```

name	salary	more	pct_more
Mike	7100.00	2500.00	54
James	6600.00	2000.00	43
Betty	6300.00	1700.00	37
Andy	5400.00	800.00	17
Sandy	5400.00	800.00	17
Tracy	4800.00	200.00	4
Carol	4600.00	0.00	0

RANK and DENSE_RANK

```
SELECT name, salary, RANK() OVER s, DENSE_RANK() OVER s
FROM emp
WINDOW s AS (ORDER BY salary DESC)
ORDER BY salary DESC;
```

name	salary	rank	dense_rank
Mike	7100.00	1	1
James	6600.00	2	2
Betty	6300.00	3	3
Andy	5400.00	4	4
Sandy	5400.00	4	4
Tracy	4800.00	6	5
Carol	4600.00	7	6

Departmental Average

```
SELECT name, department, salary,
       round(AVG(salary) OVER
             (PARTITION BY department), 2) AS avg,
       round(salary - AVG(salary) OVER
             (PARTITION BY department), 2) AS diff_avg
FROM emp
ORDER BY department, salary DESC;
```

name	department	salary	avg	diff_avg
Mike	Marketing	7100.00	6700.00	400.00
Betty	Marketing	6300.00	6700.00	-400.00
Sandy	Sales	5400.00	5000.00	400.00
Carol	Sales	4600.00	5000.00	-400.00
James	Shipping	6600.00	5600.00	1000.00
Andy	Shipping	5400.00	5600.00	-200.00
Tracy	Shipping	4800.00	5600.00	-800.00

WINDOW Clause

```
SELECT name, department, salary,
       round(AVG(salary) OVER d, 2) AS avg,
       round(salary - AVG(salary) OVER d, 2) AS diff_avg
FROM emp
WINDOW d AS (PARTITION BY department)
ORDER BY department, salary DESC;
```

```
 name  | department | salary  |   avg   | diff_avg
-------+------------+---------+---------+----------
 Mike  | Marketing  | 7100.00 | 6700.00 |   400.00
 Betty | Marketing  | 6300.00 | 6700.00 |  -400.00
 Sandy | Sales      | 5400.00 | 5000.00 |   400.00
 Carol | Sales      | 4600.00 | 5000.00 |  -400.00
 James | Shipping   | 6600.00 | 5600.00 |  1000.00
 Andy  | Shipping   | 5400.00 | 5600.00 |  -200.00
 Tracy | Shipping   | 4800.00 | 5600.00 |  -800.00
```

Compared to Next Department Salary

```
SELECT name, department, salary,
       salary - LEAD(salary, 1) OVER
       (PARTITION BY department
        ORDER BY salary DESC) AS diff_next
FROM emp
ORDER BY department, salary DESC;
```

name	department	salary	diff_next
Mike	Marketing	7100.00	800.00
Betty	Marketing	6300.00	(null)
Sandy	Sales	5400.00	800.00
Carol	Sales	4600.00	(null)
James	Shipping	6600.00	1200.00
Andy	Shipping	5400.00	600.00
Tracy	Shipping	4800.00	(null)

Departmental and Global Ranks

```
SELECT name, department, salary, RANK() OVER s AS dept_rank,
       RANK() OVER (ORDER BY salary DESC) AS global_rank
FROM emp
WINDOW s AS (PARTITION BY department ORDER BY salary DESC)
ORDER BY department, salary DESC;
```

name	department	salary	dept_rank	global_rank
Mike	Marketing	7100.00	1	1
Betty	Marketing	6300.00	2	3
Sandy	Sales	5400.00	1	4
Carol	Sales	4600.00	2	7
James	Shipping	6600.00	1	2
Andy	Shipping	5400.00	2	4
Tracy	Shipping	4800.00	3	6

6. Considerations

https://www.flickr.com/photos/10413717@N08/

Tips

- Do you want to split the set? (PARTITION BY creates multiple partitions)
- Do you want an order in the partition? (use ORDER BY)
- How do you want to handle rows with the same ORDER BY values?
  - RANGE vs ROW
  - RANK vs DENSE_RANK
- Do you need to define a window frame?
- Window functions can define their own partitions, ordering, and window frames.
- Multiple window names can be defined in the WINDOW clause.
- Pay attention to whether window functions operate on frames or partitions.

Window Function Summary

Scope	Type	Function	Description
frame	computation	*generic aggs.*	e.g. `SUM`, `AVG`
	row access	`FIRST_VALUE`	first frame value
		`LAST_VALUE`	last frame value
		`NTH_VALUE`	*n*th frame value
partition	row access	`LAG`	row before current
		`LEAD`	row after current
		`ROW_NUMBER`	current row number
	ranking	`CUME_DIST`	cumulative distribution
		`DENSE_RANK`	rank without gaps
		`NTILE`	rank in *n* partitions
		`PERCENT_RANK`	percent rank
		`RANK`	rank with gaps

Window functions never process rows outside their partitions.
However, without PARTITION BY the partition is the entire set.

Conclusion

http://momjian.us/presentations

https://www.flickr.com/photos/10318765@N03/