Window Functions

- Apart from grouping and aggregation, PostgreSQL provides another way to perform computations based on the values of several records.
- It can be done using window functions.
- Grouping and aggregation mean one single output record for every group of several input records.
- Window functions can do similar things, but they are executed for every record, and the number of records in the output and the input is the same.

- Window functions are evaluated after grouping and aggregation.
- For this reason, the only places in the `SELECT` query where the window functions are allowed are select-list and the `ORDER BY` clause.

Window Definitions

- The syntax of the window functions is as follows:
  - ```
    <function_name> (<function_arguments>)
    OVER(
       [PARTITION BY <expression_list>]
       [ORDER BY <order_by_list>]
       [<frame_clause>])
    ```
- The construct in the parentheses after the `OVER` keyword is called the **window definition**.
    ```

- Window functions, in general, work like aggregating functions.
- They process sets of records.
- These sets are built separately for each processed record and can overlap.
- That's why, unlike the normal aggregating functions, window functions are evaluated for each row.

- For each record, a set of rows to be processed by a window function is built in the following way:

- At the beginning, the `PARTITION BY` clause is processed. All the records that have the same values after evaluating the expressions from `expression_list` as the current row is taken. The set of these rows is called the **partition**. The current row is also included in the partition.
  If no `PARTITION BY` is specified, it means that all the rows will be included in a single partition at this step.

- Next, the partition is sorted according to the `ORDER BY` clause, which has the same syntax and logic as the `ORDER BY` clause in the `SELECT` statement. If the `ORDER BY` clause is omitted, then all of the records of the set are considered to have the same position.

- The last part of the window definition is called the **frame clause**.
- The frame clause can be one of the following:
  - `ROWS | RANGE | GROUPS <frame_start> [<frame_exclusion>]`
  - `ROWS | RANGE | GROUPS BETWEEN <frame_start> AND <frame_end> [<frame_exclusion>]`

- Processing the frame clause means taking a subset from the whole partition to pass it to the window function.
- The subset is called the **window frame**.

- The frame has its start and end points.
- The definition of these points has different semantics depending on the type of the window frame.
- The start and end points can be any of the following:
  - `UNBOUNDED PRECEDING`: The very first record of the partition
  - `<offset> PRECEDING`: A record that is placed several records before the current one
  - `CURRENT ROW`: The current row itself
  - `<offset> FOLLOWING`: A record that is placed several records after the current record
  - `UNBOUNDED FOLLOWING`: The very last record of the partition
- These definitions make sense only when the partition is sorted, therefore, usage of the frame clause is only allowed when the `ORDER BY` clause is there.
- The start point should precede the end point.
  - That's why, for example, `ROWS BETWEEN CURRENT ROW AND 1 PRECEDING` is not correct.

- A window frame can be defined using one of the three modes: the `ROWS` mode, the `GROUPS` mode, or the `RANGE` mode.
- The mode affects the meaning of the `CURRENT ROW` and the `offset` in the definition of the boundaries of the window frame.

- **The ROWS mode**: The `CURRENT ROW` points to the current row itself. The `offset` must be an expression that returns a non-negative integer. The `offset` then points to a row that's that number of rows before or after the current row.

- **The GROUPS mode**: The `GROUPS` mode deals with *peer groups*, which are groups of rows that have the same position in the ordered list according to the rule set by the `ORDER BY` clause. When defining the start point of the window frame, the `CURRENT ROW` points to the first row of the same peer group where the current row belongs. The `offset` specifies the number of the preceding peer group, and the window would start from the first row of that group. When defining the end point, they would point to the last rows of the respective peer groups. The `offset`, again, must be an integer expression that returns a non-negative number.

- **The RANGE mode**: The `CURRENT ROW` points to the beginning or the end of the same peer group as the current row, just like in the `GROUPS` mode. The `offset` specifies a difference between the current row and the start or end point of the window frame. This works only when the `ORDER BY` clause has only one expression in its list. Depending on the type of that expression, the `offset` may also have a different type. For example, when the rows are ordered by a field of the `date` type, the `offset` should have the `interval` type. The `offset` should be non-negative.

- If `frame_end` is omitted, `CURRENT ROW` is used by default.
- If the whole frame clause is omitted, the frame will be built using the `RANGE UNBOUNDED PRECEDING` definition.

- It's possible to exclude certain rows from the window frame.
- The `EXCLUDE` construct is used for this.
- The construct can be one of the following:
  - `EXCLUDE CURRENT ROW`: Will remove the current row from the window frame
  - `EXCLUDE GROUP`: Will remove the whole peer group of the current row
  - `EXCLUDE TIES`: Excludes the peer group of the current row, but leaves the current row in
  - `EXCLUDE NO OTHERS`: Doesn't exclude anything; this is the default

- Look at the following example of a window definition:
  - ```
    OVER (
       PARTITION BY a
       ORDER BY b
       ROWS BETWEEN UNBOUNDED PRECEDING AND 5 FOLLOWING)
    ```
    ```

- Here's another example:
  - ```
    OVER (
       PARTITION BY a % 2
       ORDER BY b
       GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    ```
    ```

- An empty window definition means that all the records will form a single partition.
- In this case, the behavior of a window function will be similar to aggregation without a `GROUP BY` clause, when all rows are aggregated.

The WINDOW Clause

- Window definitions can be quite long, and in many cases, it isn't convenient to use them in the select-list.
- Several window functions can use the same or similar window definitions.
- PostgreSQL provides a way to define windows and give them names that can be used in the `OVER` clause in window functions.

- This is done using the `WINDOW` clause of the `SELECT` statement, which is specified after the `HAVING` clause, as follows:
  - ```
    SELECT count() OVER w,
           sum(b) OVER w,
           avg(b) OVER (w ORDER BY c
                          ROWS BETWEEN
                          1 PRECEDING AND 1 FOLLOWING)
    FROM table1
    WINDOW w AS (PARTITION BY a)
    ```
    ```

- When the same window definition is used several times, PostgreSQL will optimize the execution of the query by building partitions only once and then reusing the results.

Using Window Functions

- Any aggregating function can be used as a window function, with the exception of ordered-set and hypothetical-set aggregates.
- User-defined aggregating functions can also be used as window functions.
- The presence of the `OVER` clause indicates that the function is used as a window function.

- When the aggregating function is used as a window function, it will aggregate the rows that belong to the window frame of a current row.
- A typical use case for window functions are computing statistical values of different kinds.

```
car_portal=> WITH
car_portal-> monthly_data AS (
car_portal(>   SELECT   date_trunc('month', advertisement_date) AS month, count(*) as cnt
car_portal(>   FROM     car_portal_app.advertisement
car_portal(>   GROUP BY date_trunc('month', advertisement_date))
car_portal-> SELECT to_char(month,'YYYY-MM') as month, cnt,
car_portal->        sum(cnt) OVER (w ORDER BY month) AS cnt_year,
car_portal->        round(avg(cnt) OVER (ORDER BY month ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING), 1) AS mov_avg,
car_portal->        round(cnt / sum(cnt) OVER w * 100,1) AS ratio_year
car_portal-> FROM   monthly_data
car_portal-> WINDOW w AS (PARTITION BY date_trunc('year',month));
  month   | cnt | cnt_year | mov_avg | ratio_year
----------+-----+----------+---------+------------
 2014-01 |  42 |       42 |    40.3 |        5.8
 2014-02 |  49 |       91 |    44.5 |        6.7
 2014-03 |  30 |      121 |    56.8 |        4.1
 2014-04 |  57 |      178 |    69.0 |        7.8
 2014-05 | 106 |      284 |    73.0 |       14.6
 2014-06 | 103 |      387 |    81.0 |       14.2
 2014-07 |  69 |      456 |    86.0 |        9.5
 2014-08 |  70 |      526 |    74.0 |        9.6
 2014-09 |  82 |      608 |    60.6 |       11.3
 2014-10 |  46 |      654 |    54.2 |        6.3
 2014-11 |  36 |      690 |    49.8 |        5.0
 2014-12 |  37 |      727 |    35.2 |        5.1
 2015-01 |  48 |       48 |    32.5 |       84.2
 2015-02 |   9 |       57 |    31.3 |       15.8
(14 rows)
```

- There are several window functions that are not aggregating functions.
- They are used to get the values of other records within the partition, to calculate the rank of the current row among all rows, and to generate row numbers.

```
car_portal=> WITH
car_portal-> monthly_data AS (
car_portal(>   SELECT   date_trunc('month', advertisement_date) AS month, count(*) as cnt
car_portal(>   FROM     car_portal_app.advertisement
car_portal(>   GROUP BY date_trunc('month', advertisement_date))
car_portal-> SELECT   to_char(month,'YYYY-MM') as month, cnt,
car_portal->          cnt - lag(cnt) OVER (ORDER BY month) as prev_m,
car_portal->          cnt - lag(cnt, 12) OVER (ORDER BY month) as prev_y,
car_portal->          rank() OVER (w ORDER BY cnt DESC) as rank
car_portal-> FROM     monthly_data
car_portal-> WINDOW   w AS (PARTITION BY date_trunc('year',month))
car_portal-> ORDER BY month DESC;
  month   | cnt | prev_m | prev_y | rank
---------+-----+--------+--------+------
 2015-02 |   9 |    -39 |    -40 |    2
 2015-01 |  48 |     11 |      6 |    1
 2014-12 |  37 |      1 |        |   10
 2014-11 |  36 |    -10 |        |   11
 2014-10 |  46 |    -36 |        |    8
 2014-09 |  82 |     12 |        |    3
 2014-08 |  70 |      1 |        |    4
 2014-07 |  69 |    -34 |        |    5
 2014-06 | 103 |     -3 |        |    2
 2014-05 | 106 |     49 |        |    1
 2014-04 |  57 |     27 |        |    6
 2014-03 |  30 |    -19 |        |   12
 2014-02 |  49 |      7 |        |    7
 2014-01 |  42 |        |        |    9
(14 rows)
```

- A more detailed description of these functions is available in the documentation
  at http://www.postgresql.org/docs/current/static/functions-window.html

Window Functions with Grouping and Aggregation

- As window functions are evaluated after grouping, it's possible to use aggregating functions inside window functions, but not the other way around.
- The code shown here is correct:
  - `sum(count(*)) OVER()`
- The following approach will also work:
  - `sum(a) OVER(ORDER BY count(*))`
- However, `sum(count(*) OVER())` is wrong.

- For example, to calculate the rank of the seller accounts by the number of advertisements they make, the following query can be used:

```
car_portal=> SELECT   seller_account_id,
car_portal->          dense_rank() OVER(ORDER BY count(*) DESC)
car_portal-> FROM     car_portal_app.advertisement
car_portal-> GROUP BY seller_account_id;
```

seller_account_id	dense_rank
26	1
128	2
28	2
126	2
111	3
80	4
57	4
11	4
10	4
13	4
83	5
108	5
96	5
89	5

23	16
33	16
55	16
75	16
22	16
20	16
67	16
87	16
129	17
144	17
84	17
79	17

(114 rows)