Partitioning and window functions

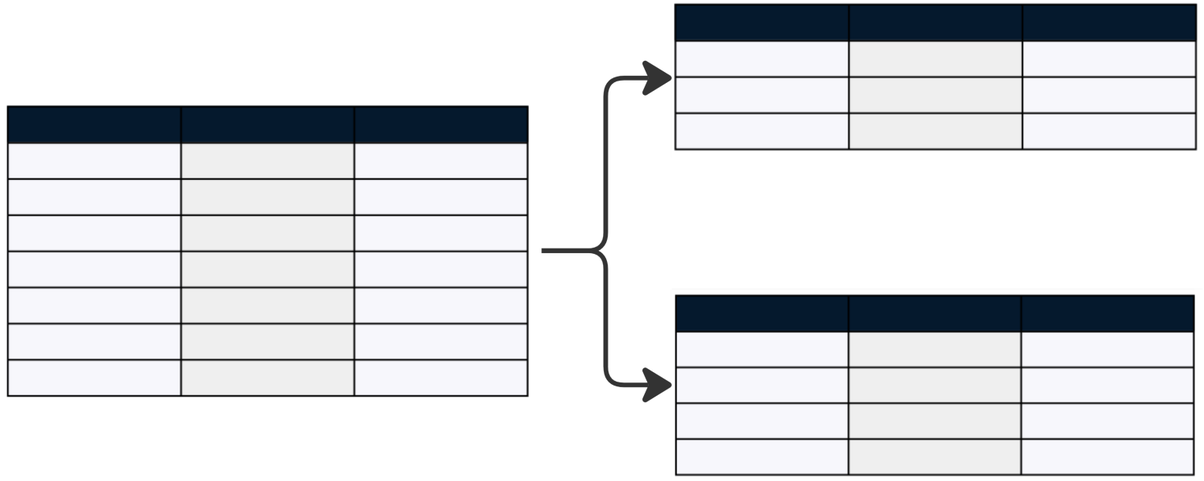
TIME SERIES ANALYSIS IN POSTGRESQL

Jasmin Ludolf
Content Developer, DataCamp

datacamp

Partitioning

- Split a large table into smaller subsets, or partitions
- Useful when dealing with large datasets
- Time series data:
  - new data everyday
  - partition by range of dates



A range

- Range boundaries:
  - inclusive at lower end
  - exclusive at upper end
- Alternative to range: partition by event or category

Range 1 : 2000-2010

2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009

Range 2 : 2010-2020

2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020

Partition by

- `PARTITION BY` `RANGE`

```
CREATE TABLE timetable (
    date_info DATE,
    train_id INTEGER,
    departure_time TIMESTAMP,
    arrival_time TIMESTAMP,
    delay INTEGER
) PARTITION BY RANGE (date_info);
```

Partition of

- `PARITION OF` `FOR VALUES FROM`
- `timetable` is a partitioned table, partitioned on `date_info` using the `RANGE` method

```
CREATE TABLE timetable_y2020 PARTITION OF timetable
    FOR VALUES FROM ('2020-01-01') to ('2020-12-31');
```

Window functions

- Simplify regular SQL queries
- Returns a value for each row in a table (can depend on other rows)

- **Window** : set of rows on which the function operates
- **Window function** : the function used on those rows, or window
- Aggregate functions are comparative, but the result is not grouped into one value

Over clause

- The `OVER` clause indicates a window function
- Example: `regional_timetable` date, times, id, delays, and region
- If no `PARTITION BY` : the window function treats the entire table as one partition

```
SELECT region, train_id, delay, AVG(delay) OVER (PARTITION BY region)
FROM regional_timetable;
```

```
|region|train_id|   delay|     avg|
|------|--------|--------|--------|
| north|       6|00:00:05|00:00:05|
| north|       2|00:00:03|00:00:05|
| north|       5|00:00:07|00:00:05|
|  east|       1|00:00:10|00:00:21|
|  east|       3|00:00:32|00:00:21|
| south|       8|00:01:00|00:01:00|
```

Let's practice!

TIME SERIES ANALYSIS IN POSTGRESQL

Top items with window functions

TIME SERIES ANALYSIS IN POSTGRESQL

Jasmin Ludolf
Content Developer, DataCamp

datacamp

Table description

- `temperatures_monthly` table
- Fields:
  - `station_id` - weather station id
  - `year_month` - year and month
  - `t_monthly_min` - monthly minimum temperature in Celsius
  - `t_monthly_max` - monthly maximum temperature in Celsius
  - `t_monthly_avg` - monthly average temperature in Celsius

Traditional aggregation

```
SELECT station_id, year_month, t_monthly_min
FROM temperatures_monthly AS tm
JOIN
(
  SELECT
        station_id, min(t_monthly_min) AS t_monthly_min
    FROM temperatures_monthly
    WHERE year_month BETWEEN '2018-01-01' AND '2018-12-31'
    GROUP BY station_id
) as p
USING(station_id, t_monthly_min)
WHERE year_month BETWEEN '2018-01-01' AND '2018-12-31'
ORDER BY station_id, t_monthly_min;
```

Traditional aggregation

station_id	year_month	t_monthly_min
1	2018-12-01	5.4
2	2018-12-01	6.6
3	2018-01-01	1.6
3	2018-02-01	1.6
4	2018-02-01	-19.0
6	2018-01-01	2.6
8	2018-02-01	-16.3

Problems:

- Station `5` is missing, it had no data for 2018
- Two rows for station `3`, two rows had the same minimum temp `3`
- Only gives the coldest single month, what if we want the two coldest months?

Using row_number

- We need to number each row for each station
- `ROW_NUMBER()` : numbers the rows per partition, in ascending order

```
SELECT station_id, year_month, t_monthly_min,
    ROW_NUMBER() OVER (PARTITION BY station_id ORDER BY t_monthly_min) AS rank
FROM temperatures_monthly AS tm
WHERE year_month BETWEEN '2018-01-01' AND '2018-12-31'
ORDER BY station_id, t_monthly_min;
```

Numbering the rows

station_id	year_month	t_monthly_min	rank
1	2018-12-01	5.4	1
1	2018-02-01	6.2	2
1	2018-01-01	7.4	3
1	2018-11-01	9.0	4
1	2018-03-01	10.6	5
1	2018-04-01	14.7	6
1	2018-10-01	16.6	7
1	2018-05-01	17.2	8
1	2018-06-01	21.9	9
1	2018-09-01	24.4	10
1	2018-07-01	28.1	11
1	2018-08-01	28.1	12
2	2018-12-01	6.6	1
2	2018-01-01	7.6	2
2	2018-02-01	8.1	3

- Numbering restarts with station `2`
- Because we partitioned by `station_id`

Lowest temperatures

```
SELECT * FROM
(
    SELECT station_id, year_month, t_monthly_min,
    ROW_NUMBER() OVER
          (PARTITION BY station_id ORDER BY t_monthly_min) AS rank
    FROM temperatures_monthly
    WHERE year_month BETWEEN '2018-01-01' AND '2018-12-31'
) AS q
WHERE rank < 3
ORDER BY station_id, rank;
```

Lowest temperatures

station_id	year_month	t_monthly_min	rank
1	2018-12-01	5.4	1
1	2018-02-01	6.2	2
2	2018-12-01	6.6	1
2	2018-01-01	7.6	2
3	2018-01-01	1.6	1
3	2018-02-01	1.6	2

Highest values

- To reverse the order, add `DESC`:

```
ROW_NUMBER() OVER (PARTITION BY station_id ORDER BY t_monthly_min DESC)
```

Let's practice!

TIME SERIES ANALYSIS IN POSTGRESQL

Ranking functions

TIME SERIES ANALYSIS IN POSTGRESQL

Jasmin Ludolf
Content Developer, DataCamp

datacamp

Row number gives sequential numbering

```
SELECT
    station_id,
    year_month,
    t_monthly_min,
    ROW_NUMBER() OVER (
      PARTITION BY station_id
      ORDER BY t_monthly_min) AS rank
FROM temperatures_monthly AS tm
WHERE year_month BETWEEN '2018-01-01'
    AND '2018-12-31'
ORDER BY station_id, t_monthly_min;
```

```
|station_id|year_month|t_monthly_min|rank|
|----------|----------|-------------|----|
|         1|2018-12-01|          5.4|   1|
|         1|2018-02-01|          6.2|   2|
|         1|2018-01-01|          7.4|   3|
|         1|2018-11-01|          9.0|   4|
|         1|2018-03-01|         10.6|   5|
|         1|2018-04-01|         14.7|   6|
|         1|2018-10-01|         16.6|   7|
|         1|2018-05-01|         17.2|   8|
|         1|2018-06-01|         21.9|   9|
|         1|2018-09-01|         24.4|  10|
|         1|2018-07-01|         28.1|  11|
...
```

Rank

- `RANK()` : assigns a rank to each row within each partition according to `ORDER BY` , starting with `1` , repeats rank for similar values

```
...
RANK() OVER (
        PARTITION BY station_id
        ORDER BY t_monthly_min DESC) AS rank
...
```

```
...
ROW_NUMBER() OVER (
        PARTITION BY station_id
        ORDER BY t_monthly_min DESC) AS row
...
```

Rank vs. row

`RANK()`

year_month	t_monthly_min	rank
2018-02-01	1.6	1
2018-01-01	1.6	1
2018-03-01	2	3
2018-11-01	2.5	4
2018-12-01	2.7	5
2018-04-01	3.4	6
2018-10-01	4.1	7
2018-09-01	4.7	8

...

`ROW_NUMBER()`

year_month	t_monthly_min	rank
2018-02-01	1.6	1
2018-01-01	1.6	2
2018-03-01	2	3
2018-11-01	2.5	4
2018-12-01	2.7	5
2018-04-01	3.4	6
2018-10-01	4.1	7
2018-09-01	4.7	8

...

Dense rank

- `DENSE_RANK()` : assigns a rank to each row within each partition according to `ORDER BY`, starting with `1`, repeats rank for similar values
  - Doesn't skip a rank

```
...
DENSE_RANK() OVER (
    PARTITION BY station_id
    ORDER BY t_monthly_min DESC) AS rank
...
```

Rank vs. dense rank

`RANK()`

year_month	t_monthly_min	rank
2018-02-01	1.6	1
2018-01-01	1.6	1
2018-03-01	2	3
2018-11-01	2.5	4
2018-12-01	2.7	5
2018-04-01	3.4	6
2018-10-01	4.1	7
2018-09-01	4.7	8

...

`DENSE_RANK()`

year_month	t_monthly_min	rank
2018-02-01	1.6	1
2018-01-01	1.6	1
2018-03-01	2	2
2018-11-01	2.5	3
2018-12-01	2.7	4
2018-04-01	3.4	5
2018-10-01	4.1	6
2018-09-01	4.7	7

...

Percent rank

- `PERCENT_RANK` : assigns a rank to each row within each partition according to `ORDER BY` , as a percentage
  - (rank - 1) / (total partition rows - 1)
  - Float values from `0` to `1`

```
...
PERCENT_RANK() OVER (
    PARTITION BY station_id
    ORDER BY t_monthly_min DESC) AS percent_rank
...
```

Percent rank output

```
|year_month|t_monthly_min|percent_rank|
|----------|-------------|------------|
|2018-02-01|          1.6|           0|
|2018-01-01|          1.6|           0|
|2018-03-01|            2|        0.18|
|2018-11-01|          2.5|        0.27|
|2018-12-01|          2.7|        0.36|
|2018-04-01|          3.4|        0.45|
|2018-10-01|          4.1|        0.54|
|2018-09-01|          4.7|        0.63|
...
```

Let's practice!

TIME SERIES ANALYSIS IN POSTGRESQL