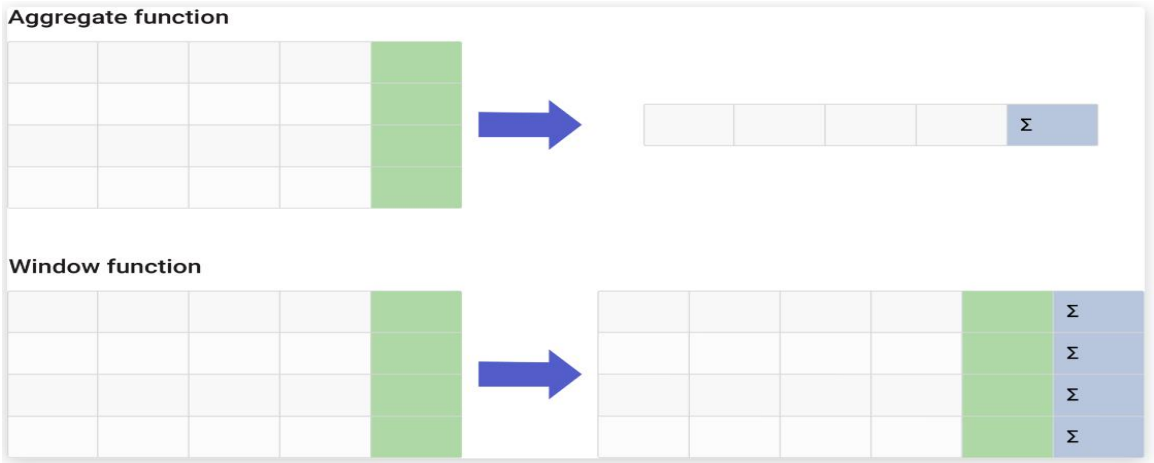
Ahmet YÜCE

-- WINDOW FUNCTION

-- By using window functions, you can perform certain complex business calculations with only a few lines of code. This is especially helpful with time series data analysis and calculations like moving averages, running totals, rankings, etc. Also, by using window functions, you will have much cleaner and more readable code that is easier to maintain.

-- **A window function** performs a calculation across a set of table rows that are somehow **related to the current row.** They are also called **OVER functions** or **analytic functions**. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of **a window function does not cause rows to become grouped into a single output row – 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.

-- In SQL, window functions operate on a set of rows called a window frame. They return a single value for each row from the underlying query. The window frame (or simply window) is defined using the **OVER()** clause. This clause also allows defining a window based on a specific column (similar to GROUP BY). To calculate the returned values, window functions may use aggregate functions, but they will use them with the OVER() clause. Note that **the rows are not collapsed;** we still have one row for each of our transactions.



List of Window Functions:

- Ranking Functions
  - `row_number()`
  - `rank()`
  - `dense_rank()`
- Distribution Functions
  - `percent_rank()`
  - `cume_dist()`
- Analytic Functions
  - `lead()`

    - `lag()`
    - `ntile()`
    - `first_value()`
    - `last_value()`
    - `nth_value()`
- Aggregate Functions
    - `avg()`
    - `count()`
    - `max()`
    - `min()`
    - `sum()`

- **Aggregate** functions – These are regular aggregate functions that you have probably used with `GROUP BY`. However, they can also be used with `OVER()`. Unlike regular aggregations used in combination with `GROUP BY`, when they are used with `OVER()`, rows are not collapsed. Each record gets its own calculated values. This group of functions represents `sum`, `avg`, `min`, `max`, and `count`.
- **Ranking** window functions – These are used to assign a rank or row number to each record inside a partition. The most famous functions in this group are `rank()`, `dense_rank()`, and `row_number()`.

city	price	row_number	rank	dense_rank
		over(order by price)		
Paris	7	1	1	1
Rome	7	2	1	1
London	8.5	3	3	2
Berlin	8.5	4	3	2
Moscow	9	5	5	3
Madrid	10	6	6	4
Oslo	10	7	6	4

- **Positional/ Analytic** window functions – Functions like `first_value`, `last_value`, `lead`, and `lag` return a single value from a particular row in each window frame (there are no aggregations). This "value" can be the value of the first/last record in each window frame, or it can return a value from the previous row or from the next row (lead/lag).

first_value(sold) OVER
(PARTITION BY city ORDER BY month)

city	month	sold	first_value
Paris	1	500	500
Paris	2	300	500
Paris	3	400	500
Rome	2	200	200
Rome	3	300	200
Rome	4	500	200

last_value(sold) OVER
(PARTITION BY city ORDER BY month
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)

city	month	sold	last_value
Paris	1	500	400
Paris	2	300	400
Paris	3	400	400
Rome	2	200	500
Rome	3	300	500
Rome	4	500	500

lag(sold) OVER(ORDER BY month)

order by month

month	sold	
1	500	NULL
2	300	500
3	400	300
4	100	400
5	500	100

lead(sold) OVER(ORDER BY month)

order by month

month	sold	
1	500	300
2	300	400
3	400	100
4	100	500
5	500	NULL

lag(sold, 2, 0) OVER(ORDER BY month)

order by month

month	sold	
1	500	0
2	300	0
3	400	500
4	100	300
5	500	400

offset=2

lead(sold, 2, 0) OVER(ORDER BY month)

order by month

month	sold	
1	500	400
2	300	100
3	400	500
4	100	0
5	500	0

offset=2

ntile(3)

city	sold	
Rome	100	1
Paris	100	1
London	200	1
Moscow	200	2
Berlin	200	2
Madrid	300	2
Oslo	300	3
Dublin	300	3

- **Distribution** functions – In this group, there are two famous functions: `cume_dist` and `percent_rank`. Both calculate where each row value stands in a group of other values inside the same group/partition/window frame.

cume_dist() OVER(ORDER BY sold)

city	sold	cume_dist
Paris	100	0.2
Berlin	150	0.4
Rome	200	0.8
Moscow	200	0.8
London	300	1

80% of values are less than or equal to this one

percent_rank() OVER(ORDER BY sold)

city	sold	percent_rank
Paris	100	0
Berlin	150	0.25
Rome	200	0.5
Moscow	200	0.5
London	300	1

without this row 50% of values are less than this row's value

Window functions differ from aggregate functions used with GROUP BY in that they:

- Use `OVER()` instead of `GROUP BY()` to define a set of rows.
- May use many functions other than aggregates (e.g. `RANK()`, `LAG()`, or `LEAD()`).
- Groups rows on the row's rank, percentile, etc. as well as its column value.
- Do not collapse rows.
- May use a sliding window frame (which depends on the current row).

-
- GROUP BY --> no usage of DINSTICT clause
- WF --> optional
- GROUP BY --> requires an AGGREGATE Function
- WF --> optional
- GROUP BY --> Ordering invalid
- WF --> ordering valid
- GROUP BY --> low performance
- WF --> high performance

Logical Order of Operations in SQL

1. `FROM`, `JOIN`
2. `WHERE`
3. `GROUP BY`
4. aggregate functions
5. `HAVING`
6. **window functions**
7. `SELECT`
8. `DISTINCT`
9. `UNION`/`INTERSECT`/`EXCEPT`
10. `ORDER BY`
11. `OFFSET`
12. `LIMIT`/`FETCH`/`TOP`

You can use window functions in `SELECT` and `ORDER BY`. However, you can't put window functions anywhere in the `FROM`, `WHERE`, `GROUP BY`, or `HAVING` clauses.

Syntax:

```
SELECT city, month,
  sum(sold) OVER (
    PARTITION BY city
    ORDER BY month
    RANGE UNBOUNDED PRECEDING) total
FROM sales;
```

```
SELECT <column_1>, <column_2>,
  <window_function> OVER (
    PARTITION BY <...>
    ORDER BY <...>
    <window_frame>) <window_column_alias>
FROM <table_name>;
```

Named Window Definition

```
SELECT country, city,
  rank() OVER country_sold_avg
FROM sales
WHERE month BETWEEN 1 AND 6
GROUP BY country, city
HAVING sum(sold) > 10000
WINDOW country_sold_avg AS (
  PARTITION BY country
  ORDER BY avg(sold) DESC)
ORDER BY country, city;
```

```
SELECT <column_1>, <column_2>,
  <window_function>() OVER <window_name>
FROM <table_name>
WHERE <...>
GROUP BY <...>
HAVING <...>
WINDOW <window_name> AS (
  PARTITION BY <...>
  ORDER BY <...>
  <window_frame>)
ORDER BY <...>;
```

****** WINDOW FUNCTIONS USED WITH “OVER” ******

* AGG() + OVER() AS
* AGG() + OVER(OREDER BY.....) AS
* AGG() + OVER(PARTITION BY.....) AS
* AGG() + OVER(PARTITION BY..... OREDER BY.....) AS

- `OVER` denotes that this is a window function. Due to this keyword, sometimes window functions are also called OVER functions.
- `PARTITION BY` tells us how the rows are grouped into logical chunks/groups.
- Aggregate functions do not require an `ORDER BY`, by default it operates according to the previous value. They accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

`PARTITION BY` divides rows into multiple groups, called **partitions**, to which the window function is applied.

month	city	sold
1	Rome	200
2	Paris	500
1	London	100
1	Paris	300
2	Rome	300
2	London	400
3	Rome	400

PARTITION BY city

month	city	sold	sum
1	Paris	300	800
2	Paris	500	800
1	Rome	200	900
2	Rome	300	900
3	Rome	400	900
1	London	100	500
2	London	400	500

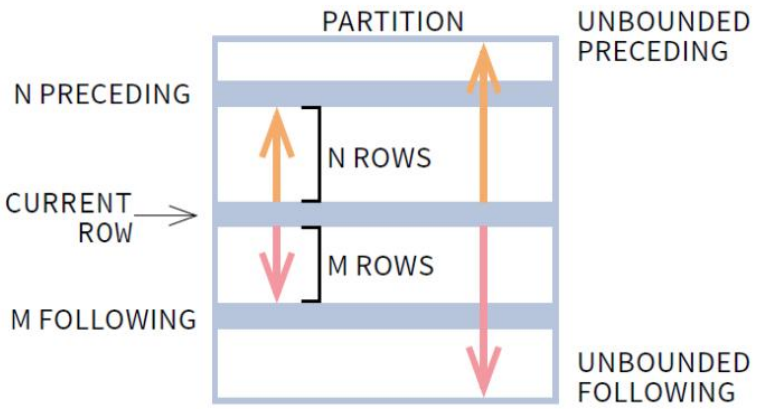
ORDER BY specifies the order of rows in each partition to which the window function is applied.

sold	city	month
200	Rome	1
500	Paris	2
100	London	1
300	Paris	1
300	Rome	2
400	London	2
400	Rome	3

PARTITION BY city ORDER BY month

sold	city	month
300	Paris	1
500	Paris	2
200	Rome	1
300	Rome	2
400	Rome	3
100	London	1
400	London	2

A **window frame** is a set of rows that are somehow related to the current row. The window frame is evaluated separately within each partition.



Code

```
ROWS | RANGE | GROUPS BETWEEN lower_bound AND upper_bound
```

The bounds can be any of the five options:

- `UNBOUNDED PRECEDING`
- `n PRECEDING`
- `CURRENT ROW`
- `n FOLLOWING`
- `UNBOUNDED FOLLOWING`

The `lower_bound` must be BEFORE the `upper_bound`.

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row →	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

1 row before the current row and 1 row after the current row

RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row →	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

values in the range between 3 and 5
ORDER BY must contain a single expression

EXAMPLES FROM SAMPLERETAIL DATABASE

VALUES THAT CANNOT BE DISPLAYED ON A ROW BASIS WITH GROUP BY CAN BE DISPLAYED WITH WINDOW FUNCTIONS.In the table below, unlike GROUP BY, we were able to print the total and annual total sales amounts of that product, the cumulative sales totals, and the sum of each line with the previous and next sales separately.

-- Aggregate Functions:

```
select product_id, model_year, list_price,
sum(list_price) over() as total_price,
sum(list_price) over(partition by model_year) as price_by_year,
sum(list_price) over(partition by model_year order by product_id) as cumulative,
sum(list_price) over(partition by model_year order by product_id rows between 1
preceding and 1 following) as window
from product.product;
```

1.SUM 2.SUM 3.SUM 4.SUM

	product_id	model_year	list_price	total	price_by_year	cumulative	window
1	1	2018	379.99	488109.84	25487.78	379.99	1129.98
2	2	2018	749.99	488109.84	25487.78	1129.98	2129.97
3	3	2018	999.99	488109.84	25487.78	2129.97	4649.97
4	4	2018	2899.99	488109.84	25487.78	5029.96	5220.97
5	5	2018	1320.99	488109.84	25487.78	6350.95	4690.97
6	6	2018	469.99	488109.84	25487.78	6820.94	5790.97
7	7	2018	3999.99	488109.84	25487.78	10820.93	6269.97
8	8	2018	1799.99	488109.84	25487.78	12620.92	8799.97
9	9	2018	2999.99	488109.84	25487.78	15620.91	6348.98

-- Analytic / Positional Functions: FIRST_VALUE, LAST_VALUE, LEAD, LAG, NTH_VALUE

- `first_value(expr)` - the value for the first row within the window frame
- `last_value(expr)` - the value for the last row within the window frame
- `lead(expr, offset, default)` - the value for the row *offset* rows after the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = NULL
- `lag(expr, offset, default)` - the value for the row *offset* rows before the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = NULL
- `nth_value(expr, n)` - the value for the *n*-th row within the window frame; *n* must be an integer. `ntile(n)` - divide rows within a partition as equally as possible into *n* groups, and assign each row its group number.
- `first_value()`, `last_value()`, and `nth_value()` do not require an `ORDER BY`. They accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).
- `ntile()`, `lead()`, and `lag()` require an `ORDER BY`. They do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).
- With the default window frame for `ORDER BY`, `RANGE UNBOUNDED PRECEDING`, `last_value()` returns the value for the current row.

```
-- FIRST_VALUE, LAST_VALUE

Select list_price, model_year,
first_value(list_price) over(order by model_year) as first_price_value_of_model_year,
last_value(list_price) over(order by model_year) as last_price_value_of_model_year,
first_value(list_price) over(order by model_year rows between 3 preceding and 3
following) first_of_3rows,
last_value(list_price) over(order by model_year rows between 3 preceding and 3 following)
as last_of_3rows
From product.product;
```

	list_price	model_year	first_price_value_of_model_year	last_price_value_of_model_year	first_of_3rows	last_of_3rows
172	1499.98	2018	159.99	2197.99	99.99	65.00
173	199.99	2018	159.99	2197.99	299.99	49.93
174	50.50	2018	159.99	2197.99	25.99	2197.99
175	65.00	2018	159.99	2197.99	1499.98	161.99
176	49.93	2018	159.99	2197.99	199.99	53.05
177	2197.99	2018	159.99	2197.99	50.50	66.17
178	161.99	2019	159.99	67.99	65.00	349.95
179	53.05	2019	159.99	67.99	49.93	199.99
180	66.17	2019	159.99	67.99	2197.99	299.99
181	349.95	2019	159.99	67.99	161.99	116.99
182	199.99	2019	159.99	67.99	53.05	66.99
183	299.99	2019	159.99	67.99	66.17	39.99
184	116.99	2019	159.99	67.99	349.95	699.98

```
--------
select model_year, list_price,
first_value(list_price) over(partition by model_year order by list_price) as
first_value_of_model_year,
last_value(list_price) over(partition by model_year order by list_price) as
last_value_of_model_year,
first_value(list_price) over(partition by model_year order by list_price rows between 3
preceding and 3 following) first_of_3rows,
last_value(list_price) over(partition by model_year order by list_price rows between 3
preceding and 3 following) as last_of_3rows
from product.product;
```

	model_year	list_price	first_value_of_model_year	last_value_of_model_year	first_of_3rows	last_of_3rows
172	2018	2799.99	2.00	2799.99	2197.99	3137.95
173	2018	2799.99	2.00	2799.99	2199.98	3989.99
174	2018	2998.00	2.00	2998.00	2498.00	4295.98
175	2018	3137.95	2.00	3137.95	2799.99	4295.98
176	2018	3989.99	2.00	3989.99	2799.99	4295.98
177	2018	4295.98	2.00	4295.98	2998.00	4295.98
178	2019	1.00	1.00	1.00	1.00	2.00
179	2019	1.00	1.00	1.00	1.00	3.00
180	2019	1.00	1.00	1.00	1.00	12.49
181	2019	2.00	1.00	2.00	1.00	19.99
182	2019	3.00	1.00	3.00	1.00	22.08
183	2019	12.49	1.00	12.49	1.00	28.88
184	2019	19.99	1.00	19.99	2.00	29.99
185	2019	22.08	1.00	22.08	3.00	34.58
186	2019	28.88	1.00	28.88	12.49	39.99
187	2019	29.99	1.00	29.99	19.99	41.35
188	2019	34.58	1.00	34.58	22.08	45.99

```sql
-- LEAD(to next rows), LAG(to previous rows)  default:1
select product_id, list_price,
sum(list_price) over() as total,
sum(list_price) over(order by product_id) as cumulative,
lag(list_price, 2) over(order by product_id) as previous_two_list_price,
lead(list_price, 3) over(order by product_id) as next_three_list_price,
sum(list_price) over(order by product_id rows between 1 preceding and 1 following) as
total_list_price_of_tria
from product.product
order by product_id;
```

-- lag(list_price, 2) over(order by product_id) as previous_two_list_price:

	product_id	list_price	total	cumulative	previous_two_list_price
1	1	23.99	234294.11	23.99	NULL
2	2	136.99	234294.11	160.98	NULL
3	3	599.00	234294.11	759.98	23.99
4	4	151.99	234294.11	911.97	136.99
5	5	199.99	234294.11	1111.96	599.00
6	6	89.95	234294.11	1201.91	151.99
7	7	59.99	234294.11	1261.90	199.99
8	8	99.99	234294.11	1361.89	89.95
9	9	121.99	234294.11	1483.88	59.99
10	10	174.99	234294.11	1658.87	99.99
11	11	29.99	234294.11	1688.86	121.99
12	12	499.99	234294.11	2188.85	174.99

-- Lead(list_price, 3) over(order by product_id) as next_three_list_price:

	product_id	list_price	total	cumulative	previous_two_list_price	next_three_list_price
1	1	23.99	234294.11	23.99	NULL	151.99
2	2	136.99	234294.11	160.98	NULL	199.99
3	3	599.00	234294.11	759.98	23.99	89.95
4	4	151.99	234294.11	911.97	136.99	59.99
5	5	199.99	234294.11	1111.96	599.00	99.99
6	6	89.95	234294.11	1201.91	151.99	121.99
7	7	59.99	234294.11	1261.90	199.99	174.99
8	8	99.99	234294.11	1361.89	89.95	29.99
9	9	121.99	234294.11	1483.88	59.99	499.99
10	10	174.99	234294.11	1658.87	99.99	99.99
11	11	29.99	234294.11	1688.86	121.99	249.99
12	12	499.99	234294.11	2188.85	174.99	67.99

-- sum(list_price) over(order by product_id rows between 1 preceding and 1 following) as total_list_price_of_tria:

	product_id	list_price	total	cumulative	previous_two_list_price	next_three_list_price	total_list_price_of_tria
1	1	23.99	234294.11	23.99	NULL	151.99	160.98
2	2	136.99	234294.11	160.98	NULL	199.99	759.98
3	3	599.00	234294.11	759.98	23.99	89.95	887.98
4	4	151.99	234294.11	911.97	136.99	59.99	950.98
5	5	199.99	234294.11	1111.96	599.00	99.99	441.93
6	6	89.95	234294.11	1201.91	151.99	121.99	349.93
7	7	59.99	234294.11	1261.90	199.99	174.99	249.93
8	8	99.99	234294.11	1361.89	89.95	29.99	281.97
9	9	121.99	234294.11	1483.88	59.99	499.99	396.97
10	10	174.99	234294.11	1658.87	99.99	99.99	326.97
11	11	29.99	234294.11	1688.86	121.99	249.99	704.97
12	12	499.99	234294.11	2188.85	174.99	67.99	629.97

```
-- NTILE(INT) : DIVIDES EQUAL GROUPS

select product_id, list_price, model_year,
ntile(10) over(partition by model_year order by list_price desc) as splitted_group_no
from product.product
```

	product_id	list_price	model_year	splitted_group_no
504	73	68.99	2021	8
505	43	68.75	2021	9
506	15	67.99	2021	9
507	49	65.89	2021	9
508	7	59.99	2021	9
509	63	54.99	2021	9
510	69	54.99	2021	9
511	62	39.99	2021	9
512	50	34.99	2021	9
513	74	33.79	2021	10
514	71	29.99	2021	10
515	11	29.99	2021	10
516	18	24.99	2021	10
517	22	23.99	2021	10
518	1	23.99	2021	10
519	17	11.99	2021	10
520	30	11.79	2021	10

-- Ranking Functions: ROW_NUMBER, RANK, DENSE_RANK (GIVES SEQUENCE NUMBER)

- `row_number()` - unique number for each row within partition, with different numbers for tied values
- `rank()` - ranking within partition, with gaps and same ranking for tied values. **WHEN GIVING THE INDEX NUMBER OF THE FIRST VALUE TO ALL OF THE SAME VALUES; COUNTER WORKS BEHIND.**
- `dense_rank()` - ranking within partition, with no gaps and same ranking for tied values. **COUNTER CONTINUES FROM WHERE IT LEFT.**
- RANK and DENSE_RANK will assign the grades the same rank depending on how they fall compared to the other values. However, RANK will then skip the next available ranking value whereas DENSE_RANK would still use the next chronological ranking value.
- `rank()` and `dense_rank()` require `ORDER BY`, but `row_number()` does not require `ORDER BY`.

```
select product_id, model_year, list_price,
row_number() over(order by product_id) as row_1,
row_number() over(partition by model_year order by product_id) as row_2
from product.product
```

	product_id	model_year	list_price	row_1	row_2
169	251	2018	99.99	251	169
170	252	2018	299.99	252	170
171	253	2018	25.99	253	171
172	254	2018	1499.98	254	172
173	255	2018	199.99	255	173
174	256	2018	50.50	256	174
175	257	2018	65.00	257	175
176	258	2018	49.93	258	176
177	259	2018	2197.99	259	177
178	260	2019	161.99	260	1
179	261	2019	53.05	261	2
180	262	2019	66.17	262	3
181	263	2019	349.95	263	4
182	264	2019	199.99	264	5
183	265	2019	299.99	265	6
184	266	2019	116.99	266	7
185	267	2019	66.99	267	8

```
select model_year, list_price,
rank() over(order by list_price) as ranked,
dense_rank() over(order by list_price) as dense_ranked
from product.product
```

	model_year	list_price	ranked	dense_ranked
1	2019	1.00	1	1
2	2019	1.00	1	1
3	2019	1.00	1	1
4	2019	2.00	4	2
5	2018	2.00	4	2
6	2019	3.00	6	3
7	2018	7.99	7	4
8	2020	10.99	8	5
9	2021	11.79	9	6
10	2021	11.99	10	7

-- Distribution Functions: CUME_DIST, PERCENT_RANK

- `percent_rank()` - the percentile ranking number of a row—a value in [0, 1] interval: (rank-1) / (total number of rows - 1).
- `cume_dist()` - the cumulative distribution of a value within a group of values, i.e., the number of rows with values less than or equal to the current row's value divided by the total number of rows; a value in (0, 1] interval.
- Distribution functions require `ORDER BY`. They do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

```
select product_id,
cume_dist() over(order by product_id) as cume_disted,
percent_rank() over(order by product_id) as percent_ranked
from product.product
```

product_id	cume_disted	percent_ranked
1	0.00192307692307692	0
2	0.00384615384615385	0.00192678227360308
3	0.00576923076923077	0.00385356454720617
4	0.00769230769230769	0.00578034682080925
5	0.00961538461538462	0.00770712909441233
6	0.0115384615384615	0.00963391136801541
7	0.0134615384615385	0.0115606936416185
8	0.0153846153846154	0.0134874759152216
9	0.0173076923076923	0.0154142581888247
10	0.0192307692307692	0.0173410404624277

REFERENCES

https://learnsql.com/blog/sql-window-functions-cheat-sheet/

https://mode.com/sql-tutorial/sql-window-functions/

Marija Ilic, “How SQL Window Functions Can Help Managers Decide Who Gets a Raise”, https://learnsql.com/blog/sql-window-functions-for-managers/

Kateryna Koidan, “SQL Window Functions vs. SQL Aggregate Functions: Similarities and Differences”, https://learnsql.com/blog/window-functions-vs-aggregate-functions/