Cracking SQL

Window Functions

for advanced queries

Arindam Chakraborty

linkedin.com/in/arindam430

Let’s Swipe

01

What are SQL Window Functions?

Window functions are special functions that perform **calculations** across a set of table rows related to the current row. These rows are defined by something called a **window** (or subset of data). The key advantage? Window functions let us calculate **rankings**, **totals**, **running averages**, and more, without grouping or losing individual row details.

Let’s Swipe

02

Why Use Window Functions?

- **No Loss of Row Details:** Unlike GROUP BY, window functions retain all the rows in our query while still calculating things like sums or ranks.
- **Flexible Calculations:** They enable operations like running totals, rankings or finding a moving average.
- **Cleaner Queries:** Simplifies complex tasks like ranking and partitioned calculations.

Let’s Swipe

03

How Do Window Functions Work?

Window functions use the OVER clause to define a window (subset of data).

The OVER clause specifies two things:

- **Partitioning:** Defines subsets of data (similar to GROUP BY) that the function will work on.
- **Ordering:** Specifies the order of rows within each subset.

Basic Syntax -

```
<function>() OVER (PARTITION BY
<column> ORDER BY <column>)
```

Let’s Swipe →

04

Scenario - 1:

Understanding Row-level & Category level Contribution with OVER & PARTITION BY

Let’s Swipe

05

Imagine we have an expenses table that records different expense categories and amounts.

	date date	description text	category text	amount integer
1	2022-10-25	A2B restaurant	Food	6000
2	2022-10-02	Amazon	Shopping	3000
3	2022-10-18	Banglore muni water bill	Utilities	600
4	2022-10-02	Croma store	Shopping	13000
5	2022-10-10	D Mart grocery bill	Shopping	4300
6	2022-10-12	Macdonalds	Food	2700
7	2022-10-10	Pani puri on street	Food	400
8	2022-10-05	PSEG electricity bill	Utilities	7000
9	2022-10-15	Reliance geo phone bill	Utilities	800
10	2022-10-01	Saravana bhavan	Food	2700
11	2022-10-11	Thakur saris	Shopping	23000
12	2022-10-17	Verizon wireless	Utilities	2300

We want to calculate the percentage contribution of each row and category to the total expenses. For example:

- **Row level contribution:** How much does each individual row contribute to the total expenses?
- **Category-level contribution:** How much does each row contribute to the total expenses within its category?

Let’s Swipe

- **Row-level contribution using OVER**

  Window function can calculate the total once and use it for every row, avoiding the need for a subquery.

Calculating the percentage contribution of each row to the total expenses -

```sql
SELECT
    *,
    ROUND(amount*100.0/SUM(amount)
OVER(), 2) AS pct_contribution
FROM expenses
ORDER BY pct_contribution DESC;
```

Output -

	date date	description text	category text	amount integer	pct_contribution numeric
1	2022-10-11	Thakur saris	Shopping	23000	34.95
2	2022-10-02	Croma store	Shopping	13000	19.76
3	2022-10-05	PSEG electricity bill	Utilities	7000	10.64
4	2022-10-25	A2B restaurant	Food	6000	9.12
5	2022-10-10	D Mart grocery bill	Shopping	4300	6.53
6	2022-10-02	Amazon	Shopping	3000	4.56
7	2022-10-12	Macdonalds	Food	2700	4.10
8	2022-10-01	Saravana bhavan	Food	2700	4.10
9	2022-10-17	Verizon wireless	Utilities	2300	3.50
10	2022-10-15	Reliance geo phone bill	Utilities	800	1.22
11	2022-10-18	Banglore muni water bill	Utilities	600	0.91
12	2022-10-10	Pani puri on street	Food	400	0.61

Let’s Swipe

07

- **Category-Level Contribution Using PARTITION BY**

Calculating the percentage contribution within each category by creating windows for each -

```
SELECT
    *,
    ROUND(amount*100.0/SUM(amount)
OVER(PARTITION BY category), 2) AS
pct_contrib
FROM expenses
ORDER BY category;
```

Output -

	date date	description text	category text	amount integer	pct_contrib numeric
1	2022-10-25	A2B restaurant	Food	6000	50.85
2	2022-10-12	Macdonalds	Food	2700	22.88
3	2022-10-10	Pani puri on street	Food	400	3.39
4	2022-10-01	Saravana bhavan	Food	2700	22.88
5	2022-10-02	Amazon	Shopping	3000	6.93
6	2022-10-02	Croma store	Shopping	13000	30.02
7	2022-10-10	D Mart grocery bill	Shopping	4300	9.93
8	2022-10-11	Thakur saris	Shopping	23000	53.12
9	2022-10-18	Banglore muni water bill	Utilities	600	5.61
10	2022-10-05	PSEG electricity bill	Utilities	7000	65.42
11	2022-10-15	Reliance geo phone bill	Utilities	800	7.48
12	2022-10-17	Verizon wireless	Utilities	2300	21.50

Let’s Swipe

08

Scenario - 2: Understanding ROW_NUMBER, RANK & DENSE_RANK

Let’s Swipe

09

Imagine we have a table student_marks with student IDs, names, and their marks.

	student_id [PK] integer	name text	marks integer
1	111	Akhil	7
2	114	Prashant	60
3	116	Suresh	14
4	120	Ravi	95
5	138	Mahesh	72
6	140	Kiran	60
7	145	Raju	95
8	150	Gayatri	60
9	157	Arjun	33
10	180	Vipul	60
11	190	Ramesh	85
12	200	Aditya	14
13	214	Parth	28
14	220	Ishika	98

We want to rank students based on their marks, but different ranking methods may be needed depending on the situation:

- Assign unique numbers to each student, even if their marks are the same.
- Rank students with ties (same marks) while skipping ranks after ties.
- Rank students with ties but ensure the next rank continues sequentially.

Let’s Swipe

- **ROW_NUMBER**

Assigns a unique sequential number to each row within the specified window. This is useful when we need a unique identifier for each row, even when multiple rows have the same rank.

Assigning a unique sequential number to each row, even if marks are tied -

```
SELECT
    *,
ROW_NUMBER() OVER(ORDER BY marks
DESC) AS rn
FROM student_marks LIMIT 5;
```

Output -

	student_id [PK] integer	name text	marks integer	rn bigint
1	220	Ishika	98	1
2	145	Raju	95	2
3	120	Ravi	95	3
4	190	Ramesh	85	4
5	138	Mahesh	72	5

Let’s Swipe

- **RANK**

Assigns a rank to each row within the specified window, but skips ranks when there are ties.

Ranking students with ties (same marks) while skipping ranks after ties -

```
SELECT
    *,
RANK() OVER(ORDER BY marks DESC) AS rnk
FROM student_marks LIMIT 10;
```

Output -

	student_id [PK] integer	name text	marks integer	rnk bigint
1	220	Ishika	98	1
2	145	Raju	95	2
3	120	Ravi	95	2
4	190	Ramesh	85	4
5	138	Mahesh	72	5
6	150	Gayatri	60	6
7	114	Prashant	60	6
8	140	Kiran	60	6
9	180	Vipul	60	6
10	157	Arjun	33	10

In this example, two students tied for 2nd place, which means the next rank is 4th, and four students tied for 6th place, meaning the next rank is 10th.

Let’s Swipe

- **DENSE_RANK**

Similar to RANK(), but ensures ranks remain sequential within the same window, with no skipped ranks.

Ranking students with ties while ensuring that the next rank continues sequentially -

```
SELECT
    *,
DENSE_RANK()OVER(ORDER BY marks DESC) AS rnk
FROM student_marks LIMIT 10;
```

Output -

	student_id [PK] integer	name text	marks integer	rnk bigint
1	220	Ishika	98	1
2	145	Raju	95	2
3	120	Ravi	95	2
4	190	Ramesh	85	3
5	138	Mahesh	72	4
6	150	Gayatri	60	5
7	114	Prashant	60	5
8	140	Kiran	60	5
9	180	Vipul	60	5
10	157	Arjun	33	6

Here, rows with the same marks get the same rank, but the next rank continues sequentially.

Let’s Swipe

Key Differences Between the Functions:

Function	Ties	Skipped Ranks	Sequential Order
ROW_NUMBER	No ties Allowed	No	Always Unique
RANK	Yes	Yes	Skips after ties
DENSE_RANK	Yes	No	Continues sequentially

Key Takeaways:

- **ROW_NUMBER:** Use when distinct numbering for all rows is needed, regardless of ties (e.g., unique row IDs).
- **RANK:** Use when ties matter and skipped ranks are acceptable (e.g., sports rankings).
- **DENSE_RANK:** Use when ties matter but ranks should remain sequential (e.g., grading systems or loyalty tiers).

Let’s Swipe →

14

Scenario - 3: Understanding LEAD, LAG & NTILE

Let’s Swipe

Let’s focus on the same student_marks table with three columns: student_id, name, and marks.

	student_id [PK] integer	name text	marks integer
1	111	Akhil	7
2	114	Prashant	60
3	116	Suresh	14
4	120	Ravi	95
5	138	Mahesh	72
6	140	Kiran	60
7	145	Raju	95
8	150	Gayatri	60
9	157	Arjun	33
10	180	Vipul	60
11	190	Ramesh	85
12	200	Aditya	14
13	214	Parth	28
14	220	Ishika	98

We want to analyze the marks of students by comparing them to others in the table or grouping them into performance bands. For example:

- Comparing each student’s marks with the next to spot gaps or trends.
- Comparing with the previous student to track improvements or consistency.
- Grouping students into bands (e.g., top, middle, bottom) for relative performance analysis.

Let’s Swipe

- **LEAD**

Fetching the marks of the next student for each row -

```
SELECT
    *,
LEAD(marks, 1, 0) OVER(ORDER BY marks) AS
next_marks
FROM student_marks LIMIT 10;
```

Output -

	student_id [PK] integer	name text	marks integer	next_marks integer
1	111	Akhil	7	14
2	116	Suresh	14	14
3	200	Aditya	14	28
4	214	Parth	28	33
5	157	Arjun	33	60
6	150	Gayatri	60	60
7	114	Prashant	60	60
8	140	Kiran	60	60
9	180	Vipul	60	72
10	138	Mahesh	72	85

In this example, the LEAD() function retrieves the marks of the next student, in the order of ascending marks. If no next row exists (such as for the last student), it returns the default value of 0.

Let's Swipe

- **LAG**

Fetching the marks of the previous student for each row -

```
SELECT
    *,
LAG(marks, 1, 0) OVER(ORDER BY marks DESC)
AS previous_marks
FROM student_marks LIMIT 10;
```

Output -

	student_id [PK] integer	name text	marks integer	previous_marks integer
1	220	Ishika	98	0
2	145	Raju	95	98
3	120	Ravi	95	95
4	190	Ramesh	85	95
5	138	Mahesh	72	85
6	150	Gayatri	60	72
7	114	Prashant	60	60
8	140	Kiran	60	60
9	180	Vipul	60	60
10	157	Arjun	33	60

In this example, the LAG() function retrieves the marks of the previous student in the order of descending marks. If there's no previous row, the default value 0 is returned.

Let's Swipe

- **NTILE**

Dividing the students into 3 equal groups (tiles) based on their marks in descending order -

```
SELECT
    *,
NTILE(3) OVER(ORDER BY marks DESC) AS
tile_number
FROM student_marks;
```

Output -

	student_id [PK] integer	name text	marks integer	tile_number integer
1	220	Ishika	98	1
2	145	Raju	95	1
3	120	Ravi	95	1
4	190	Ramesh	85	1
5	138	Mahesh	72	1
6	150	Gayatri	60	2
7	114	Prashant	60	2
8	140	Kiran	60	2
9	180	Vipul	60	2
10	157	Arjun	33	2
11	214	Parth	28	3
12	116	Suresh	14	3
13	200	Aditya	14	3
14	111	Akhil	7	3

Here, the NTILE() function divides rows into 3 tiles based on the ORDER BY clause. Higher marks go into earlier tiles.

Let’s Swipe

Key Differences Between the Functions:

Function	Role	Output Description
LEAD	Look forward	Fetches marks of the next student based on marks.
LAG	Look backward	Fetches marks of the previous student based on marks
NTILE	Divide into groups	Splits rows into specified groups (e.g., quartiles).

Key Takeaways:

- **LEAD:** Compares each row with the next one, useful for identifying gaps or trends (e.g., comparing the next student's marks).
- **LAG:** Compares each row with the previous one, helpful for tracking progress (e.g., comparing with the prior student).
- **NTILE:** Divides rows into a set number of groups (tiles), like creating performance tiers (e.g., top, middle, bottom students).

Let’s Swipe →

Found this

Insightful ?

Share your thoughts in the comments & don't forget to LIKE, SAVE & SHARE!!

Arindam Chakraborty

linkedin.com/in/arindam430