Of course. Here is a detailed tutorial on SQL window functions.

Window functions are a powerful SQL feature that perform calculations across a set of rows related to the current row. Unlike aggregate functions (like SUM(), COUNT()) which collapse multiple rows into a single result, window functions return a value for **each** row based on a "window" of related rows.

The magic of a window function lies in the OVER() clause, which defines this window.

The OVER() Clause: The Heart of Window Functions

Every window function is followed by an OVER() clause, which has three optional parts that define the window of data the function operates on.

FUNCTION_NAME(column) OVER (PARTITION BY ... ORDER BY ... ROWS/RANGE BETWEEN ...)

1. **PARTITION BY**: This clause divides the rows into partitions, or groups. The window function is then applied independently to each partition. It's like a GROUP BY clause but for a window function. If you omit it, the entire result set is treated as a single partition.
2. **ORDER BY**: This clause sorts the rows within each partition. This is crucial for functions that rely on order, like ranking or calculating running totals.
3. **ROWS/RANGE (Frame Clause)**: This clause specifies a subset of rows within the current partition (the "frame"). For example, "the preceding row and the current row" or "all rows from the start of the partition to the current row." If omitted, the default is usually the entire partition.

Let's use a sample table called sales for our examples:

employee_id	department	sale_date	sale_amount
101	Hardware	2025-08-01	1500
102	Software	2025-08-01	3200
101	Hardware	2025-08-02	2200
103	Hardware	2025-08-03	2200

102	Software	2025-08-04	2800
104	Services	2025-08-05	5000
102	Software	2025-08-05	1500

Aggregate Window Functions

These are familiar aggregate functions (SUM, AVG, COUNT, MAX, MIN) used in a new way. They calculate an aggregate for the window but display the result on every row.

Example: Department Total Sales

Let's show each individual sale next to the total sales for its respective department.

SQL

```sql
SELECT
  employee_id,
  department,
  sale_amount,
  SUM(sale_amount) OVER (PARTITION BY department) AS department_total
FROM
  sales;
```

How it works:

- PARTITION BY department tells the SUM() function to only consider rows within the same department.
- The SUM() is calculated for 'Hardware', 'Software', and 'Services' independently.
- This total is then displayed on each row belonging to that department.

Result:

employee_id	department	sale_amount	department_total
101	Hardware	1500	5900
103	Hardware	2200	5900
101	Hardware	2200	5900
104	Services	5000	5000
102	Software	3200	7500
102	Software	2800	7500
102	Software	1500	7500

Example: Running Total

To calculate a running total, you need to define the order of operations using ORDER BY.

SQL

```
SELECT
    department,
    sale_date,
    sale_amount,
    SUM(sale_amount) OVER (PARTITION BY department ORDER BY sale_date) AS running_total
FROM
    sales;
```

How it works:

- PARTITION BY department: The running total restarts for each new department.
- ORDER BY sale_date: Within each department, the rows are sorted by sale_date. The SUM() adds the current row's sale_amount to the sum of all preceding rows in that partition.

Result (showing the 'Software' partition for clarity):

department	sale_date	sale_amount	running_total
...	...	...	...
Software	2025-08-01	3200	3200
Software	2025-08-04	2800	6000
Software	2025-08-05	1500	7500
...	...	...	...

Ranking Window Functions 🏆

These functions assign a rank to each row within a partition based on a specified ordering.

- ROW_NUMBER(): Assigns a unique, sequential number to each row (e.g., 1, 2, 3, 4).
- RANK(): Assigns a rank to each row. Rows with the same value receive the same rank. It skips the next rank(s) after a tie (e.g., 1, 2, 2, 4).
- DENSE_RANK(): Similar to RANK(), but it does not skip ranks after a tie (e.g., 1, 2, 2, 3).
- NTILE(n): Divides the rows into n roughly equal-sized groups and assigns a bucket number from 1 to n.

Example: Ranking Sales within Each Department

Let's rank the sales from highest to lowest within each department.

SQL

```
SELECT
  employee_id,
  department,
  sale_amount,
  ROW_NUMBER() OVER(PARTITION BY department ORDER BY sale_amount DESC) as row_num,
  RANK() OVER(PARTITION BY department ORDER BY sale_amount DESC) as rnk,
  DENSE_RANK() OVER(PARTITION BY department ORDER BY sale_amount DESC) as dense_rnk
FROM
  sales;
```

How it works:

- PARTITION BY department: Ranking is reset for each department.
- ORDER BY sale_amount DESC: We are ranking based on sale amount, from largest to smallest.
- Notice the different ways the functions handle the two Hardware sales of 2200.

Result (showing the 'Hardware' partition for clarity):

employee_id	department	sale_amount	row_num	rnk	dense_rnk
103	Hardware	2200	1	1	1
101	Hardware	2200	2	1	1
101	Hardware	1500	3	3	2
...	...	...	...	...	...

- row_num is always unique.
- rnk gives both sales a rank of 1, then skips rank 2 and goes to 3.
- dense_rnk gives both sales a rank of 1, then continues with rank 2.

Value Window Functions (Positional)

These functions allow you to access data from other rows within your window frame, such as the preceding row or the next row.

- LAG(column, offset, default): Accesses data from a previous row. offset is how many rows to go back (default is 1).
- LEAD(column, offset, default): Accesses data from a subsequent row.
- FIRST_VALUE(column): Returns the value of the specified column from the first row of the window frame.
- LAST_VALUE(column): Returns the value from the last row of the window frame.

Example: Comparing a Sale to the Previous Sale

Let's find the amount of the previous sale made by each employee.

SQL

```
SELECT
  employee_id,
  sale_date,
  sale_amount,
  LAG(sale_amount, 1, 0) OVER (PARTITION BY employee_id ORDER BY sale_date) as
previous_sale_amount
FROM
  sales;
```

How it works:

- PARTITION BY employee_id: We only want to look at the sales history for one employee at a time.
- ORDER BY sale_date: This is crucial. It puts the sales in chronological order so LAG knows which sale was "previous".
- LAG(sale_amount, 1, 0): This looks back 1 row (offset = 1) to get the sale_amount. If there is no previous row (i.e., it's the first sale for that employee), it will return 0 (default = 0).

Result (showing employee 102):

employee_id	sale_date	sale_amount	previous_sale_amo

			unt
…	…	…	…
102	2025-08-01	3200	0
102	2025-08-04	2800	3200
102	2025-08-05	1500	2800
…	…	…	…