



Window Functions



What is a Window Function?

A window function(AKA **Analytical Functions**) performs a **calculation across a set of rows related to the current row**, without collapsing rows like GROUP BY.

Key idea

- GROUP BY → reduces rows
- Window function → **keeps** all rows and adds calculated values



Why Use Window Functions?

They are useful when you want:

- Running totals
- Ranking (1st, 2nd, 3rd...)
- Comparing **current row** with **previous/next row**
- Aggregates **without losing row-level detail**



Basic Syntax (Very Important)




```
function_name(...) OVER (  
    PARTITION BY column_name  
    ORDER BY column_name  
)
```

Part

```
function_name  
  
OVER  
  
PARTITION BY  
  
ORDER BY
```

Meaning

```
Aggregate or analytic function  
  
Defines the window  
  
Splits data into groups  
  
Defines order within each group
```



```
CREATE TABLE employees (  
    employee_id INTEGER NOT NULL PRIMARY KEY,  
    name VARCHAR(100),  
    department VARCHAR(50),  
    salary INTEGER,  
    joining_date DATE  
);
```

```
INSERT INTO employees (employee_id, name,  
department, salary, joining_date) VALUES  
(1, 'Judy', 'HR', 60000, '2020-01-15'),  
(2, 'Khan', 'IT', 75000, '2019-11-23'),  
(3, 'Sameer', 'IT', 72000, '2021-03-01'),  
(4, 'Carlos', 'Finance', 68000, '2018-07-12'),  
(5, 'Eve', 'Finance', 70000, '2020-06-30'),  
(6, 'Happy', 'HR', 62000, '2019-05-17'),  
(7, 'Grace', 'IT', 77000, '2020-10-10'),  
(8, 'Heidi', 'Finance', 69000, '2022-02-14'),  
(9, 'Ivan', 'HR', 64000, '2021-08-19'),  
(10, 'Alice', 'IT', 73000, '2022-01-01');
```



	employee_id	name	department	salary	joining_date
1	1	Judy	HR	60000	2020-01-15
2	2	Khan	IT	75000	2019-11-23
3	3	Sameer	IT	72000	2021-03-01
4	4	Carlos	Finance	68000	2018-07-12
5	5	Eve	Finance	70000	2020-06-30
6	6	Happy	HR	62000	2019-05-17
7	7	Grace	IT	77000	2020-10-10
8	8	Heidi	Finance	69000	2022-02-14
9	9	Ivan	HR	64000	2021-08-19
10	10	Alice	IT	73000	2022-01-01



Running totals

With Group by clause, you **cannot** get running total:

```
SELECT
  employee_id,
  name,
  SUM(salary) AS total_salary
FROM employees
GROUP BY employee_id, name;
```

	employee_id	name	total_salary
1	1	Judy	60000
2	2	Khan	75000
3	3	Sameer	72000
4	4	Carlos	68000
5	5	Eve	70000
6	6	Happy	62000
7	7	Grace	77000
8	8	Heidi	69000
9	9	Ivan	64000
10	10	Alice	73000

With Window function **you can** get **Running Total** Example (No partition, no extra clauses). This is same as getting **cumulative sum**

```
SELECT
  employee_id,
  name,
  salary,
  SUM(salary) OVER (ORDER BY employee_id) AS running_total_salary
FROM employees;
```

Here the order is by employee_id

	employee_id	name	salary	running_total_salary
1	1	Judy	60000	60000
2	2	Khan	75000	135000
3	3	Sameer	72000	207000
4	4	Carlos	68000	275000
5	5	Eve	70000	345000
6	6	Happy	62000	407000
7	7	Grace	77000	484000
8	8	Heidi	69000	553000
9	9	Ivan	64000	617000
10	10	Alice	73000	690000

Running total of salary by joining date (company-wide)



Concept taught:

How SUM() with OVER() works and how ORDER BY defines the running sequence.

SELECT

employee_id,

name,

joining_date,

salary,

SUM(salary) OVER (

ORDER BY joining_date

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS running_total_salary

FROM employees

ORDER BY joining_date;

Here the order is by joining_date

	employee_id	name	joining_date	salary	running_total_salary
1	4	Carlos	2018-07-12	68000	68000
2	6	Happy	2019-05-17	62000	130000
3	2	Khan	2019-11-23	75000	205000
4	1	Judy	2020-01-15	60000	265000
5	5	Eve	2020-06-30	70000	335000
6	7	Grace	2020-10-10	77000	412000
7	3	Sameer	2021-03-01	72000	484000
8	9	Ivan	2021-08-19	64000	548000
9	10	Alice	2022-01-01	73000	621000
10	8	Heidi	2022-02-14	69000	690000



Ranking (1st, 2nd, 3rd...)

Hard way: Using subquery to rank employees by **salary** (highest to lowest)



```
SELECT
e1.employee_id,
e1.name,
e1.salary,
( SELECT COUNT(DISTINCT e2.salary)
  FROM employees e2
  WHERE e2.salary > e1.salary
) + 1 AS salary_rank

FROM employees e1

ORDER BY salary_rank;
```

	employee_id	name	salary	salary_rank
1	7	Grace	77000	1
2	2	Khan	75000	2
3	10	Alice	73000	3
4	3	Sameer	72000	4
5	5	Eve	70000	5
6	8	Heidi	69000	6
7	4	Carlos	68000	7
8	9	Ivan	64000	8
9	6	Happy	62000	9
10	1	Judy	60000	10



Easy way: Using Window function to rank employees by **salary**
(highest to lowest)


```
SELECT
    employee_id,
    name,
    salary,
    RANK() OVER (ORDER BY salary DESC) AS salary_rank
FROM employees;
```

	employee_id	name	salary	salary_rank
1	7	Grace	77000	1
2	2	Khan	75000	2
3	10	Alice	73000	3
4	3	Sameer	72000	4
5	5	Eve	70000	5
6	8	Heidi	69000	6
7	4	Carlos	68000	7
8	9	Ivan	64000	8
9	6	Happy	62000	9
10	1	Judy	60000	10



Using Window function to rank employees by **joining_date**

```
SELECT  
employee_id,  
name,  
joining_date,  
RANK() OVER (ORDER BY joining_date DESC) AS joining_date_rank  
  
FROM employees;
```



	employee_id	name	joining_date	joining_date_rank
1	8	Heidi	2022-02-14	1
2	10	Alice	2022-01-01	2
3	9	Ivan	2021-08-19	3
4	3	Sameer	2021-03-01	4
5	7	Grace	2020-10-10	5
6	5	Eve	2020-06-30	6
7	1	Judy	2020-01-15	7
8	2	Khan	2019-11-23	8
9	6	Happy	2019-05-17	9
10	4	Carlos	2018-07-12	10



Comparing current row with
previous / next row

Compare each employee's salary with the previous employee's salary (order by joining_date)



WITHOUT window functions (correlated subquery)

Example: Compare current salary with previous joining employee

```
SELECT
  e1.employee_id,
  e1.name,
  e1.joining_date,
  e1.salary,
  (
    SELECT e2.salary
    FROM employees e2
    WHERE e2.joining_date < e1.joining_date
    ORDER BY e2.joining_date DESC
    LIMIT 1
  ) AS previous_salary,
  e1.salary -
  (
    SELECT e2.salary
    FROM employees e2
    WHERE e2.joining_date < e1.joining_date
    ORDER BY e2.joining_date DESC
    LIMIT 1
  ) AS salary_difference
FROM employees e1
ORDER BY e1.joining_date;
```

	employee_id	name	joining_date	salary	previous_salary	salary_difference
1	4	Carlos	2018-07-12	68000	<i>NULL</i>	<i>NULL</i>
2	6	Happy	2019-05-17	62000	68000	-6000
3	2	Khan	2019-11-23	75000	62000	13000
4	1	Judy	2020-01-15	60000	75000	-15000
5	5	Eve	2020-06-30	70000	60000	10000
6	7	Grace	2020-10-10	77000	70000	7000
7	3	Sameer	2021-03-01	72000	77000	-5000
8	9	Ivan	2021-08-19	64000	72000	-8000
9	10	Alice	2022-01-01	73000	64000	9000
10	8	Heidi	2022-02-14	69000	73000	-4000

What's happening (painfully)

For **every row**:

1. Search the table again
2. Find the immediately previous joining employee
3. Fetch their salary
4. Subtract salaries

Compare each employee's salary with the previous employee's salary (order by joining_date)



WITH window functions (LAG)

Same logic as previous but it is **clean and readable**

```
SELECT
  employee_id,
  name,
  joining_date,
  salary,
  LAG(salary)          OVER (ORDER BY joining_date) AS previous_salary,
  salary - LAG(salary) OVER (ORDER BY joining_date) AS salary_difference
```

```
FROM employees
ORDER BY joining_date;
```

	employee_id	name	joining_date	salary	previous_salary	salary_difference
1	4	Carlos	2018-07-12	68000	<i>NULL</i>	<i>NULL</i>
2	6	Happy	2019-05-17	62000	68000	-6000
3	2	Khan	2019-11-23	75000	62000	13000
4	1	Judy	2020-01-15	60000	75000	-15000
5	5	Eve	2020-06-30	70000	60000	10000
6	7	Grace	2020-10-10	77000	70000	7000
7	3	Sameer	2021-03-01	72000	77000	-5000
8	9	Ivan	2021-08-19	64000	72000	-8000
9	10	Alice	2022-01-01	73000	64000	9000
10	8	Heidi	2022-02-14	69000	73000	-4000



What is IMPOSSIBLE (or impractical) without window functions



Compare with both previous and next row using window function

```
SELECT
  employee_id,
  name,
  salary,
  LAG(salary) OVER (ORDER BY joining_date) AS prev_salary,
  LEAD(salary) OVER (ORDER BY joining_date) AS next_salary
FROM employees;
```

	employee_id	name	salary	prev_salary	next_salary
1	4	Carlos	68000	<i>NULL</i>	62000
2	6	Happy	62000	68000	75000
3	2	Khan	75000	62000	60000
4	1	Judy	60000	75000	70000
5	5	Eve	70000	60000	77000
6	7	Grace	77000	70000	72000
7	3	Sameer	72000	77000	64000
8	9	Ivan	64000	72000	73000
9	10	Alice	73000	64000	69000
10	8	Heidi	69000	73000	<i>NULL</i>



Aggregates without losing row-level detail

Example tasks:

1. Show **department average salary** alongside each employee
2. Show **difference between employee salary and department average**



WITHOUT window functions (JOIN + GROUP BY)

```
SELECT
  e.employee_id,
  e.name,
  e.department,
  e.salary,
  d.dept_avg_salary,
  e.salary - d.dept_avg_salary AS diff_from_avg
FROM employees e
JOIN (
  SELECT
    department,
    AVG(salary) AS dept_avg_salary
  FROM employees
  GROUP BY department
) d
ON e.department = d.department;
```

	employee_id	name	department	salary	dept_avg_salary	diff_from_avg
1	1	Judy	HR	60000	62000	-2000
2	2	Khan	IT	75000	74250	750
3	3	Sameer	IT	72000	74250	-2250
4	4	Carlos	Finance	68000	69000	-1000
5	5	Eve	Finance	70000	69000	1000
6	6	Happy	HR	62000	62000	0
7	7	Grace	IT	77000	74250	2750
8	8	Heidi	Finance	69000	69000	0
9	9	Ivan	HR	64000	62000	2000
10	10	Alice	IT	73000	74250	-1250

<u>Issue with this method</u>	<u>Why it hurts</u>
Two queries	Harder to read
Join required	Extra mental overhead
Easy to break	Wrong joins = wrong results
Scaling logic	Becomes unreadable

Example tasks:

1. Show **department average salary** alongside each employee
2. Show **difference between employee salary and department average**



WITH window functions

```
SELECT  
employee_id,  
name,  
department,  
salary,  
AVG(salary) OVER (PARTITION BY department) AS dept_avg_salary,  
salary - AVG(salary) OVER (PARTITION BY department) AS diff_from_avg
```

```
FROM employees  
ORDER BY employee_id;
```

	employee_id	name	department	salary	dept_avg_salary	diff_from_avg
1	1	Judy	HR	60000	62000	-2000
2	2	Khan	IT	75000	74250	750
3	3	Sameer	IT	72000	74250	-2250
4	4	Carlos	Finance	68000	69000	-1000
5	5	Eve	Finance	70000	69000	1000
6	6	Happy	HR	62000	62000	0
7	7	Grace	IT	77000	74250	2750
8	8	Heidi	Finance	69000	69000	0
9	9	Ivan	HR	64000	62000	2000
10	10	Alice	IT	73000	74250	-1250

Example tasks:

- 1. Show department maximum salary alongside each employee
- 2. Show difference between employee salary and department maximum salary



WITH window functions

```
SELECT
employee_id,
name,
department,
salary,
MAX(salary) OVER (PARTITION BY department) AS dept_max_salary,
salary - MAX(salary) OVER (PARTITION BY department) AS diff_from_max

FROM employees
ORDER BY employee_id;
```

	empl	name	department	salary	dept_max_salary	diff_from_max
1	1	Judy	HR	60000	62000	-2000
2	2	Khan	IT	75000	77000	-2000
3	3	Sameer	IT	75000	77000	-2000
4	4	Carlos	Finance	68000	70000	-2000
5	5	Eve	Finance	70000	70000	0
6	6	Happy	HR	62000	62000	0
7	7	Grace	IT	77000	77000	0
8	8	Heidi	Finance	70000	70000	0
9	9	Ivan	HR	62000	62000	0
10	10	Alice	IT	73000	77000	-4000



Window Functions Supported in SQLite3

SQLite (≥ 3.25) supports:

- SUM, AVG, MIN, MAX
- ROW_NUMBER
- RANK, DENSE_RANK
- LAG, LEAD
- NTILE







Tect box