CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 9b: Window Functions

Amit Kumar Dhar

IIT Bhilai

Agenda

- Window functions — the big idea
- `PARTITION BY` & `ORDER BY`
- Ranking functions
- Window frames (`ROWS BETWEEN`)
- Offset functions (`LEAD`, `LAG`)

Window functions — the big idea

- Aggregates **collapse** groups into one row.
- **Window functions** compute an aggregate/rank **across a set of rows** but **keep every row**.

```sql
SELECT prod_name, price,
       AVG(price) OVER () AS overall_avg      -- same value on every row
FROM   products;
```

- The `OVER (...)` clause turns an aggregate into a window function.

Quick Check: OVER Clause

Question: If `MAX(salary) OVER ()` is used in a SELECT statement on an `employees` table, how many values will be returned if there are 100 employees?

Quick Check: OVER Clause

Question: If `MAX(salary) OVER ()` is used in a SELECT statement on an `employees` table, how many values will be returned if there are 100 employees?

Answer: 100 values.

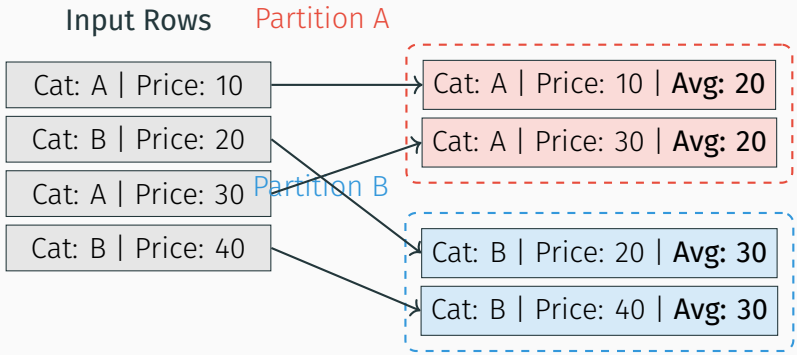
The `OVER ()` clause indicates a window function. It computes the maximum salary across all rows and repeats this single maximum value for every row in the result set, keeping all 100 rows intact.

PARTITION BY — per-group windows

```
SELECT prod_name, cat_id, price,
       AVG(price) OVER (PARTITION BY cat_id) AS cat_avg
FROM   products;
```

- `PARTITION BY` splits rows into windows (like `GROUP BY`)…
- …but **each row is preserved**, annotated with its category average.
- Compare each product to its own category's average in one query.

Understanding PARTITION BY



Quick Check: PARTITION BY

Question: Write a SELECT statement to get the `employee_name`, `department`, `salary`, and the total salary for their department.

Quick Check: PARTITION BY

Question: Write a SELECT statement to get the `employee_name`, `department`, `salary`, and the total salary for their department.

Answer:

```
SELECT employee_name, department, salary,
       SUM(salary) OVER (PARTITION BY department) AS dept_total
FROM employees;
```

Ranking functions

```sql
SELECT prod_name, cat_id, price,
       ROW_NUMBER() OVER (PARTITION BY cat_id ORDER BY price DESC) AS rn,
       RANK()       OVER (PARTITION BY cat_id ORDER BY price DESC) AS rnk,
       DENSE_RANK() OVER (PARTITION BY cat_id ORDER BY price DESC) AS drnk
FROM   products;
```

- `ROW_NUMBER` — unique 1,2,3…
- `RANK` — ties share a rank, leaves gaps (1,1,3).
- `DENSE_RANK` — ties share a rank, no gaps (1,1,2).

Visualizing Ranking Functions

Value	ROW_NUMBER	RANK	DENSE_RANK
100	1	1	1
100	2	1	1
90	3	3	2
80	4	4	3

Quick Check: Ranking

Question: If two rows tie for 2nd place, what will be the rank of the next row using `RANK()` and `DENSE_RANK()` respectively?

Quick Check: Ranking

Question: If two rows tie for 2nd place, what will be the rank of the next row using `RANK()` and `DENSE_RANK()` respectively?

Answer:

- `RANK()`: The next row will be rank **4**.
- `DENSE_RANK()`: The next row will be rank **3**.

Top-N per group (finally!)

```sql
WITH ranked AS (
  SELECT prod_name, cat_id, price,
         ROW_NUMBER() OVER (PARTITION BY cat_id ORDER BY price DESC) AS rn
  FROM products
)
SELECT * FROM ranked WHERE rn <= 2;      -- 2 priciest products per category
```

- The pattern plain `GROUP BY`/`LIMIT` couldn't do — window + filter.

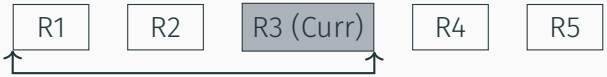
Running totals & moving windows

```
SELECT order_date, quantity,
       SUM(quantity) OVER (ORDER BY order_date
                           ROWS BETWEEN UNBOUNDED PRECEDING
                                    AND CURRENT ROW) AS running_total
FROM   daily_sales;
```

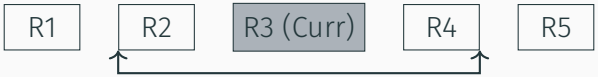
- `ORDER BY` inside `OVER` defines the running sequence.
- The **frame** (`ROWS BETWEEN ...`) sets which rows are included — enabling running totals, moving averages, etc.

Understanding Frame Clauses

Unbounded Preceding to Current Row (Running Total)



1 Preceding and 1 Following (3-Row Moving Window)



LAG and LEAD — look at neighbours

```sql
SELECT order_date, revenue,
       LAG(revenue)  OVER (ORDER BY order_date) AS prev_day,
       revenue - LAG(revenue) OVER (ORDER BY order_date) AS delta
FROM   daily_revenue;
```

- `LAG` / `LEAD` fetch a value from a previous/next row.
- Perfect for period-over-period comparisons (growth, differences).

Quick Check: LAG/LEAD

Question: What will `LAG(revenue, 2) OVER (ORDER BY date)` return for the 1st and 3rd rows in the dataset?

Quick Check: LAG/LEAD

Question: What will `LAG(revenue, 2) OVER (ORDER BY date)` return for the 1st and 3rd rows in the dataset?

Answer:

- 1st row: `NULL` (there are no rows 2 steps before the 1st row).
- 3rd row: The revenue value from the **1st row**.

Window vs GROUP BY — side by side

	GROUP BY	Window (OVER)
Rows out	one per group	one per input row
Detail kept?	no	yes
Use for	totals, summaries	ranks, running totals, per-row comparisons

- Both engines support window functions (SQLite $\geq$ 3.25).

NULLs in other places

- **`GROUP BY`** — all NULLs form **one** group together.
- **`DISTINCT`** — NULLs are considered equal (one NULL survives).
- **`UNION`** — likewise treats NULLs as equal for de-duplication.
- **`ORDER BY`** — NULLs sort first or last (engine-dependent; use **`NULLS LAST`**).
- **UNIQUE constraint** — most engines allow **multiple** NULLs (NULL $\neq$ NULL).

Aggregation/DISTINCT/GROUP BY treat NULLs as equal; **predicates** do not.

NTILE and percentiles

```sql
SELECT prod_name, price,
       NTILE(4) OVER (ORDER BY price) AS price_quartile
FROM   products;
```

- `NTILE(n)` splits ordered rows into `n` roughly equal buckets — quartiles, deciles, etc.
- Handy for segmenting customers/products into bands.

FIRST_VALUE / LAST_VALUE / NTH_VALUE

```sql
SELECT cat_id, prod_name, price,
       FIRST_VALUE(prod_name) OVER (PARTITION BY cat_id ORDER BY price DESC)
         AS priciest_in_cat
FROM   products;
```

- Pull a specific row's value into every row of the window.
- Mind the **frame**: `LAST_VALUE` needs an explicit frame to see the whole partition (`ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`).

The window frame, precisely

```
... OVER (PARTITION BY p ORDER BY o
          ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)     -- 3-row moving window
... ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW    -- running total
... RANGE BETWEEN ...                                    -- value-based frame
```

- **PARTITION BY** = which rows share a window.
- **ORDER BY** = order within the window.
- **frame** = which rows around the current one are included.

Views that use windows

```sql
CREATE VIEW product_rank AS
SELECT prod_name, cat_id, price,
       RANK() OVER (PARTITION BY cat_id ORDER BY price DESC) AS price_rank
FROM   products;

SELECT * FROM product_rank WHERE price_rank = 1;   -- priciest per category
```

- Wrap analytic logic in a view; downstream queries stay simple.

A combined mini-report

”For each customer: total spend, their rank by spend, and % of the max spender.”

```sql
WITH spend AS (
  SELECT o.cust_id, SUM(oi.quantity*oi.unit_price) AS total
  FROM orders o JOIN order_items oi ON oi.order_id=o.order_id
  GROUP BY o.cust_id)
SELECT cust_id, total,
       RANK() OVER (ORDER BY total DESC) AS rnk,
       ROUND(100.0*total / MAX(total) OVER (), 1) AS pct_of_top
FROM spend;
```

- CTE (aggregate) + window functions layered together.

Common mistakes

- `= NULL` instead of `IS NULL`.
- Expecting `x = 5 OR x <> 5` to include NULL rows.
- Forgetting `ELSE` in `CASE` (silent NULLs).
- Using `WHERE rn <= 3` in the **same** query as the window function — you must wrap it in a CTE/subquery (the window column doesn't exist yet in `WHERE`).
- Confusing `RANK` (gaps) with `DENSE_RANK` (no gaps).

Boolean aggregates

```sql
-- PostgreSQL
SELECT cust_id,
       bool_or(status = 'cancelled')  AS had_a_cancellation,
       bool_and(status = 'completed') AS all_completed
FROM   orders GROUP BY cust_id;
```

- `bool_or` = ”any row true?”; `bool_and` = ”every row true?”.
- Emulate portably with `MAX(CASE WHEN ... THEN 1 ELSE 0 END)`.

PARTITION BY multiple columns

```
SELECT cust_id, strftime('%Y', order_date) AS yr, order_id,
       COUNT(*) OVER (PARTITION BY cust_id, strftime('%Y', order_date)) AS orde
FROM   orders;
```

- The window is defined by the **combination** of partition columns.
- Each row still survives, annotated with its per-(customer, year) count.

Cumulative distribution & percent rank

```
SELECT prod_name, price,
       CUME_DIST()   OVER (ORDER BY price) AS cume_dist,
       PERCENT_RANK() OVER (ORDER BY price) AS pct_rank
FROM   products;
```

- `CUME_DIST` = fraction of rows $\leq$ this row's value.
- Useful for "this product is priced higher than X% of the catalog."

Gaps-and-islands (a teaser)

- Window functions solve classic ”find consecutive runs” problems: streaks of active days, contiguous ID ranges, sessionization.
- Technique: subtract `ROW_NUMBER()` from a sequence; equal differences mark an ”island.” A powerful pattern you’ll meet again in analytics work.

Summary — key takeaways

- **Window functions** aggregate/rank **while keeping every row** — enabling top-N per group, running totals, and neighbour comparisons.
- `PARTITION BY` splits data into independent windows.
- `ORDER BY` within `OVER( )` is critical for rankings and offsets.
- Frames (`ROWS BETWEEN`) bound the computation for running totals.

Exercises (on `store.sql`)

1. Rank products within each category by price; return the top 2 per category.
2. Running total of order counts by month.
3. Compare each order's total to the previous order using LAG.

Quick reference — window functions

```
<fn>() OVER (
    PARTITION BY <cols>      -- optional: split into windows
    ORDER BY <cols>          -- optional: order within window
    ROWS BETWEEN ... )       -- optional: the frame
```

Family	Functions
Ranking	ROW_NUMBER, RANK, DENSE_RANK, NTILE
Offset	LAG, LEAD, FIRST_VALUE, LAST_VALUE
Aggregate	SUM/AVG/COUNT/MIN/MAX ... OVER (...)
Distribution	CUME_DIST, PERCENT_RANK

Before next lecture

- Try the window-function exercises — they're new to most students.
- Optional reading: Silberschatz §5.5 (windows).

Next time — Lecture 10 (Week 4): DML in depth and **SQL from a program** (Python DB-API, parameterized queries, SQL injection).

Questions?