Module 10 of 16

Window functions

OVER and PARTITION BY for aggregates that keep every row, ROW_NUMBER, RANK, DENSE_RANK and NTILE with their tie rules, LAG, LEAD and FIRST_VALUE for comparing neighbours, frames with ROWS and RANGE for running totals and moving averages, and the top-N-per-group pattern.

GROUP BY answers questions about groups by collapsing them; window functions answer questions about rows *in the context of* their group, and keep every row. Each employee beside their department's average, each day beside the day before, each transaction with the balance after it, each product with its rank in its category — all are one window function, written as a function followed by `OVER (PARTITION BY … ORDER BY … frame)`. MySQL has supported them since 8.0, and they are the feature interviewers probe most once the basics are settled, because the tie rules, the frame defaults and the order of evaluation are easy to get silently wrong.

OVER and PARTITION BY covers aggregates as window functions, `OVER ()` for the whole result, partitions, what an ORDER BY inside the window changes, where windows sit in the order of evaluation (after WHERE, GROUP BY and HAVING — so filtering on one needs an outer query), windows over grouped rows and named windows. Ranking functions covers `ROW_NUMBER`, `RANK`, `DENSE_RANK`, their behaviour on ties, the second-highest-salary question, deterministic tie-breakers, `NTILE`, `PERCENT_RANK` and `CUME_DIST`. LAG and LEAD covers offsets and defaults, change and growth, gaps between events, partitioned comparisons and `FIRST_VALUE` / `LAST_VALUE` / `NTH_VALUE`. Running totals and frames covers frame syntax, `ROWS` versus `RANGE`, moving averages, the default frame and its ties trap. Top N per group puts it together: rank in a CTE, filter outside, and choose the ranking function by what ties should mean.

The exercises are queries judged on real tables: each salary against its department average, each rep's share of their region, three rankings side by side and a per-department rank, day-over-day change and the days between a customer's orders, a running balance per account and a 3-day moving average, the top two salary levels per department and each customer's latest order. The checkpoint adds the second-highest salary in every department, month-over-month growth, and each product's rank and gap to its category leader.

CONTENTS

1	OVER and PARTITION BY — aggregates that keep every row	15 min
2	Ranking functions — ROW_NUMBER, RANK, DENSE_RANK and NTILE	15 min
3	LAG and LEAD — comparing a row with its neighbours	14 min
4	Running totals and frames — ROWS, RANGE and moving averages	16 min
5	Top N per group — the most asked window-function question	14 min
✓	Checkpoint — Window functions	25 min

Exercises in this module compile on MySQL 8 syntax on SQLite 3.49. Reference solutions and the answer key follow each lesson.

LESSON 1 · 15 MIN

OVER and PARTITION BY — aggregates that keep every row

GROUP BY answers "the total per department", but it pays for the answer with the details: once rows are grouped, each employee's own name and salary are gone. Many questions need both at once — each employee's salary *and* their department's average; each sale *and* the running total so far; each player's score *and* their rank. **Window functions** give you exactly that. They compute over a set of related rows — the row's *window* — and attach the result to every row, without collapsing anything. They are standard SQL, supported by MySQL since 8.0, and the single most asked-about advanced SQL feature in interviews. This lesson covers the `OVER` clause, partitions, ordering within a window, where window functions sit in the order of evaluation, and the named-window shorthand.

Aggregate, but keep the rows

`employees`

id	name	dept	salary
1	Asha	Eng	90000
2	Ben	Eng	70000
3	Chen	Sales	50000
4	Dev	Sales	40000
5	Eva	HR	45000

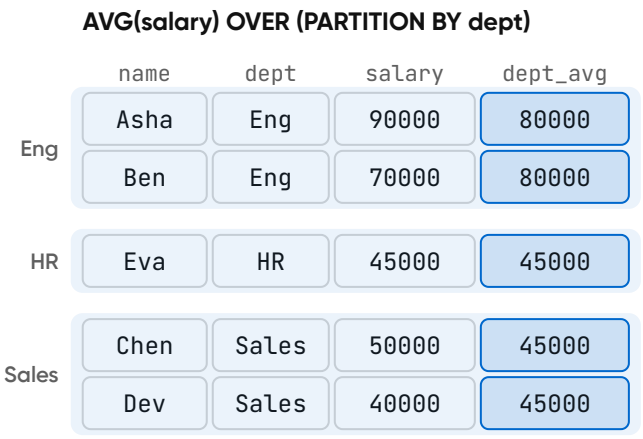
GROUP BY gives one row per department:

```
SELECT dept, AVG(salary) AS dept_avg FROM employees GROUP BY dept;
```

The same aggregate with `OVER` gives one row per *employee*, each carrying its department's average:

```sql
SELECT name, dept, salary,
       AVG(salary) OVER (PARTITION BY dept) AS dept_avg
FROM employees
ORDER BY dept, name;
```

PARTITION BY keeps every row; GROUP BY keeps one per group



1. The employees table: 5 people in 3 departments, each row with its own name and salary.
2. PARTITION BY dept sorts the rows into 3 partitions, one per department. Each row's window is its own partition; HR's partition holds a single row, and that is fine.
3. GROUP BY dept would collapse each partition into one row: 3 rows out, and the names and salaries are gone — Asha and Ben become a single Eng row.
4. With OVER the same AVG is computed per partition but written beside every row: all 5 rows survive, Asha and Ben both carry Eng's 80000, so salary minus dept_avg is one expression on one row.

The function is an ordinary aggregate — `SUM`, `AVG`, `COUNT`, `MIN`, `MAX` all work — and the `OVER (...)` turns it into a window function. Now `salary - dept_avg`, `salary / dept_total` and "is this the department maximum?" are each one expression on one row.

OVER () — the whole result as one window

Empty parentheses make every row's window the entire result:

```sql
SELECT name, salary,
       SUM(salary) OVER () AS payroll,
       ROUND(100 * salary / SUM(salary) OVER (), 1) AS pct_of_payroll,
       COUNT(*) OVER () AS headcount
FROM employees
ORDER BY salary DESC, name;
```

Before window functions, the same report needed a cross join to a one-row subquery (`CROSS JOIN (SELECT SUM(salary) AS payroll FROM employees) t`). Both still work; the window version says what it means in place.

PARTITION BY — one window per group

`PARTITION BY` splits the rows into groups, and each row's window is its own group. It takes any expressions, like GROUP BY:

```
SUM(amount) OVER (PARTITION BY region)                      -- per region
COUNT(*)    OVER (PARTITION BY customer_id, YEAR(order_date))  -- per customer
MAX(score)  OVER (PARTITION BY team)                        -- the team's best
```

Partitions do not change the number of rows and do not interact: one function may partition by department and the next by city in the same SELECT, each over the same rows. A partition of one row is perfectly fine — an HR department with one person has an average equal to that person's salary.

The comparison interviewers want, in one sentence: **GROUP BY collapses each group to one row; PARTITION BY defines groups for the window but keeps every row.**

ORDER BY inside OVER — order within the window

Adding `ORDER BY` inside `OVER` orders the rows of each partition for the function's calculation. Ranking functions (next lesson) and `LAG` / `LEAD` require it. For an aggregate, it changes the meaning: the window becomes "from the start of the partition up to this row", which turns `SUM` into a running total:

```
SELECT day, amount,
       SUM(amount) OVER (ORDER BY day) AS running_total
FROM daily_sales
ORDER BY day;
```

Exactly which rows count as "up to this row" — and what happens when two rows tie on `day` — is the subject of the frames lesson. For now remember two facts:

1. `AVG(x) OVER (PARTITION BY d)` — the whole partition.

2. `AVG(x) OVER (PARTITION BY d ORDER BY t)` — a *growing* window, from the partition's first row to the current one.

Adding an ORDER BY to a window "just to be tidy" therefore changes the numbers. Leave it out when you want the whole partition.

The window's ORDER BY does not order the output

This is the most common mistake with window functions. The ORDER BY inside `OVER` decides how the function sees the rows; it says nothing about the order the query *returns* them in. MySQL often happens to return rows in window order, because it sorted them to compute the window — and then a second window with a different ORDER BY, or a join, or a different plan reorders them. Always end with the query's own ORDER BY, and make it complete with tie-breakers:

```
SELECT name, dept, salary,
       RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS dept_rank
FROM employees
ORDER BY dept, dept_rank, name;     -- the output order, stated
```

When window functions are evaluated

The logical order of a SELECT, with windows added:

1. FROM and joins
2. WHERE
3. GROUP BY and aggregates
4. HAVING
5. **window functions**
6. SELECT list aliases, DISTINCT
7. ORDER BY
8. LIMIT

Three consequences follow.

WHERE cannot use a window function — the windows have not been computed yet. Filtering on a rank or a running total needs a CTE or derived table:

```sql
WITH ranked AS (
  SELECT name, dept, salary,
         RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS dept_rank
  FROM employees
)
SELECT name, dept, salary FROM ranked WHERE dept_rank = 1 ORDER BY dept;
```

Windows see only the rows that survived WHERE. `WHERE dept <> 'HR'` removes HR before any window is computed, so a `COUNT(*) OVER ()` counts the rest.

Windows can run over grouped rows. After GROUP BY, each group is a row, and a window function can work across groups — including over an aggregate:

```sql
SELECT dept,
       SUM(salary) AS dept_payroll,
       ROUND(100 * SUM(salary) / SUM(SUM(salary)) OVER (), 1) AS pct
FROM employees
GROUP BY dept
ORDER BY dept_payroll DESC, dept;
```

`SUM(SUM(salary)) OVER ()` reads oddly but is exact: the inner SUM is the group's aggregate, the outer SUM is a window over the grouped rows.

Named windows

When several functions share one window, name it once with a `WINDOW` clause (after HAVING, before ORDER BY):

```sql
SELECT name, dept, salary,
       AVG(salary) OVER w AS dept_avg,
       MAX(salary) OVER w AS dept_max,
       COUNT(*)    OVER w AS dept_size
FROM employees
WINDOW w AS (PARTITION BY dept)
ORDER BY dept, name;
```

A named window can also be extended: `OVER (w ORDER BY salary)` adds an ORDER BY to `w`'s partitioning. MySQL, PostgreSQL and SQLite support `WINDOW`; SQL Server added it only in 2022.

Rounding window averages

An average over integers is rarely an integer, and different engines print different numbers of decimals. Wrap window averages in `ROUND(…, 2)` (or the precision the report needs) whenever the value is shown — the exercises always do.

In other databases

PostgreSQL, SQL Server, Oracle and SQLite all support window functions with the same `OVER (PARTITION BY … ORDER BY …)` syntax; PostgreSQL and Oracle had them a decade before MySQL. Code written for MySQL 5.7 emulates them with correlated subqueries (`(SELECT AVG(salary) FROM employees e2 WHERE e2.dept = e.dept)`) or with user variables; the window version is clearer and usually much faster.

Pitfalls

- Expecting the window's ORDER BY to order the output.
- Adding ORDER BY to an aggregate's window and getting running values instead of partition totals.
- Filtering on a window function in WHERE or HAVING.
- Forgetting that WHERE has already removed rows before the window sees them.
- Printing unrounded window averages.

Key takeaways

- `function OVER (...)` computes over a window of related rows and keeps every row.
- `PARTITION BY` sets the groups; `OVER ()` is one window over the whole result.
- `ORDER BY` inside OVER orders rows for the function; for aggregates it makes a growing window.
- Windows run after WHERE, GROUP BY and HAVING — filter on them in an outer query.
- The query's own ORDER BY decides the output order; always write it.

EXERCISE 1

Salary against the department average

Show every employee beside their department's average salary, and how far above or below it they are.

employees: id	name	dept	salary
1	Asha	Eng	90000
2	Ben	Eng	70000
3	Chen	Sales	50000
4	Dev	Sales	40000
5	Eva	HR	45000
6	Gita	Eng	65000

Return `name`, `dept`, `salary`, `dept_avg` (the department's average, `ROUND`ed to 2 places) and `diff` (salary minus the department average, `ROUND`ed to 2 places). Order by `dept`, then `name`.

```
name | dept | salary | dept_avg | diff
Asha | Eng | 90000 | 75000 | 15000
Ben | Eng | 70000 | 75000 | -5000
Gita | Eng | 65000 | 75000 | -10000
Eva | HR | 45000 | 45000 | 0
Chen | Sales | 50000 | 45000 | 5000
Dev | Sales | 40000 | 45000 | -5000
```

Every employee keeps their own row — use a window, not GROUP BY.

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(30), dept VARCHAR(20),
salary INT);
INSERT INTO employees VALUES
  (1, 'Asha', 'Eng', 90000), (2, 'Ben',
'Eng', 70000), (3, 'Chen', 'Sales',
50000),
  (4, 'Dev', 'Sales', 40000), (5, 'Eva',
'HR', 45000), (6, 'Gita', 'Eng', 65000);
```

OUTPUT 1

```
name | dept | salary | dept_avg | diff
Asha | Eng | 90000 | 75000 | 15000
Ben | Eng | 70000 | 75000 | -5000
Gita | Eng | 65000 | 75000 | -10000
Eva | HR | 45000 | 45000 | 0
Chen | Sales | 50000 | 45000 | 5000
Dev | Sales | 40000 | 45000 | -5000
```

REFERENCE SOLUTION

```sql
SELECT name, dept, salary,
       ROUND(AVG(salary) OVER (PARTITION BY dept), 2) AS dept_avg,
       ROUND(salary - AVG(salary) OVER (PARTITION BY dept), 2) AS diff
FROM employees
ORDER BY dept, name;
```

EXERCISE 2

Each rep's share of the region

`sales` has one row per sale. For each rep, show their total, their region's total, and their share of it.

sales: id	region	rep	amount
1	North	Asha	300
2	North	Ben	200
3	North	Asha	100
4	South	Chen	250
5	South	Dev	250
6	South	Eva	500
7	West	Farah	80

Return `region`, `rep`, `total` (the rep's sum), `region_total` (the region's sum) and `pct_of_region` (`100 * total / region_total`, `ROUND`ed to 1 place). Order by `region`, then `total` descending, then `rep`.

```
region | rep | total | region_total | pct_of_region
North | Asha | 400 | 600 | 66.7
North | Ben | 200 | 600 | 33.3
South | Eva | 500 | 1000 | 50
South | Chen | 250 | 1000 | 25
South | Dev | 250 | 1000 | 25
West | Farah | 80 | 80 | 100
```

INPUT 1

```
CREATE TABLE sales (id INT PRIMARY KEY,
region VARCHAR(20), rep VARCHAR(20),
amount DECIMAL(10,2));
INSERT INTO sales VALUES
  (1, 'North', 'Asha', 300), (2,
'North', 'Ben', 200), (3, 'North',
'Asha', 100),
  (4, 'South', 'Chen', 250), (5,
'South', 'Dev', 250), (6, 'South',
'Eva', 500),
  (7, 'West', 'Farah', 80);
```

OUTPUT 1

```
region | rep | total | region_total |
pct_of_region
North | Asha | 400 | 600 | 66.7
North | Ben | 200 | 600 | 33.3
South | Eva | 500 | 1000 | 50
South | Chen | 250 | 1000 | 25
South | Dev | 250 | 1000 | 25
West | Farah | 80 | 80 | 100
```

REFERENCE SOLUTION

```sql
SELECT region, rep,
       SUM(amount) AS total,
       SUM(SUM(amount)) OVER (PARTITION BY region) AS region_total,
       ROUND(100 * SUM(amount) / SUM(SUM(amount)) OVER (PARTITION BY region), 1) AS
FROM sales
GROUP BY region, rep
ORDER BY region, total DESC, rep;
```

QUIZ

1. `employees` has 5 rows in 3 departments. How many rows does this return?

```sql
SELECT dept, AVG(salary) OVER (PARTITION BY dept) FROM employees;
```

A. 3

B. 5

C. 1

D. 15

2. Why is this query refused?

```sql
SELECT name, RANK() OVER (ORDER BY salary DESC) AS r
FROM employees
WHERE RANK() OVER (ORDER BY salary DESC) ⩽ 3;
```

A. RANK needs PARTITION BY

B. Window functions are evaluated after WHERE, so WHERE cannot use them

C. The alias r must be used instead

D. RANK cannot be used with DESC

3. What changes when you add `ORDER BY day` inside `SUM(amount) OVER (PARTITION BY store)`?

A. Nothing — it only sorts the output

B. The sum becomes a running total from the partition's first row to the current row

C. The query's rows are returned sorted by day

D. It is an error: aggregates cannot be ordered

4. What does `COUNT(*) OVER ()` put on each row?

A. 1

B. The row's position

C. The total number of rows in the result (after WHERE)

D. The number of rows in the table, ignoring WHERE

5. Which query puts each department's payroll as a percentage of the company's beside the department?

A. `SELECT dept, SUM(salary) * 100 / SUM(salary) FROM employees GROUP BY dept`

B. `SELECT dept, 100 * SUM(salary) / SUM(SUM(salary)) OVER () FROM employees GROUP BY dept`

C. `SELECT dept, 100 * salary / SUM(salary) OVER () FROM employees GROUP BY dept`

D. `SELECT dept, 100 * SUM(salary) OVER (PARTITION BY dept) FROM employees GROUP BY dept`

ANSWER KEY · OVER AND PARTITION BY — AGGREGATES THAT KEEP EVERY ROW

1. B A window function keeps every input row: 5 rows, each with its department's average. `GROUP BY dept` would return 3. Add DISTINCT if you want one row per department from the window version.

2. B WHERE runs before window functions are computed. Compute the rank in a CTE or derived table and filter on it outside. Using the alias in WHERE fails for the same reason; PARTITION BY is optional.

3. B With an ORDER BY, the default frame becomes "from the start of the partition up to the current row", so SUM turns into a running total. The output order is still decided only by the query's own ORDER BY.

4. C Empty `OVER ()` is one window over all rows that reached the window stage — after WHERE, GROUP BY and HAVING. Rows filtered out by WHERE are not counted.

5. B After GROUP BY each department is a row; `SUM(SUM(salary)) OVER ()` sums those group totals across all rows. The first option divides each group by itself (always 100), and the others mix ungrouped columns or never divide by the company total.

LESSON 2 · 15 MIN

Ranking functions — ROW_NUMBER, RANK, DENSE_RANK and NTILE

"Who came first?" is one of the most common questions asked of data — the top sellers, the highest-paid in each department, the latest order per customer, the second highest salary. The ranking window functions answer it. `ROW_NUMBER`, `RANK` and `DENSE_RANK` all number rows in a chosen order; they differ only in how they treat ties, and choosing the right one is a favourite interview question precisely because it is easy to get wrong silently. This lesson covers the three, how ties behave, ranking within partitions, making a ranking deterministic, and the bucketing functions `NTILE`, `PERCENT_RANK` and `CUME_DIST`.

Three ways to number rows

`scores`

player	score
Asha	90
Ben	90
Chen	85
Dev	80
Eva	80
Farah	70

```
SELECT player, score,
       ROW_NUMBER() OVER (ORDER BY score DESC, player) AS row_num,
       RANK()       OVER (ORDER BY score DESC) AS rnk,
       DENSE_RANK() OVER (ORDER BY score DESC) AS dense_rnk
FROM scores
ORDER BY score DESC, player;
```

ROW_NUMBER, RANK and DENSE_RANK on the same ties

ORDER BY score DESC

player	score	row_num	rnk	dense_rnk	
Asha	90	1	1	1	
Ben	90	2	1	1	
Chen	85	3	3	2	next distinct score → 2
Dev	80	4	4	3	
Eva	80	5	4	3	
Farah	70	6	6	4	

the window: OVER (ORDER BY score DESC)

1. All three functions number the rows in the window's order, highest score first. Two ties matter: Asha and Ben share 90, Dev and Eva share 80.
2. ROW_NUMBER gives every row a different number, 1 to 6. Inside a tie the ORDER BY decides — here the player name puts Asha before Ben; without it the engine would choose.
3. RANK is one plus the rows strictly ahead. Tied rows share a rank, then the count jumps: Asha and Ben are both 1, and Chen, with 2 rows ahead, is 3. No one is ranked 2.
4. DENSE_RANK is the value's position among the 4 distinct scores, so ties share and nothing is skipped: Chen is 2, Farah 4. This is the one for "the second highest score".

- **`ROW_NUMBER()`** gives 1, 2, 3, … — every row a different number, ties broken however the ORDER BY (or the engine) decides.
- **`RANK()`** gives tied rows the same number and then *skips*: two players share 1, so the next is 3. A row's rank is one plus the number of rows strictly ahead of it — competition ranking, as in sport.
- **`DENSE_RANK()`** gives ties the same number and does *not* skip: the next distinct score is 2. A row's dense rank is the position of its value among the distinct values.

Without ties, all three are identical. Note the aliases: `rank` is a reserved word in MySQL 8 (it names the function), so `AS rank` fails — use `rnk`, `position` or `` `rank` `` in backticks.

Choosing between them

Question	Function
Number rows uniquely, pick exactly one per group, paginate	`ROW_NUMBER`
Standings where ties share a place and push the next one down	`RANK`
"The nth highest value", "top three price levels"	`DENSE_RANK`

The classic: **the second highest salary**. With salaries 100, 100, 90, `RANK` gives 1, 1, 3 — no row has rank 2, and the answer comes out empty. `ROW_NUMBER` gives 1, 2, 3 and returns 100 — the same as the highest. Only `DENSE_RANK` gives 1, 1, 2 and returns 90:

```
WITH ranked AS (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS dr
  FROM employees
)
SELECT DISTINCT salary FROM ranked WHERE dr = 2;
```

(The pre-window answer is still worth knowing: `SELECT MAX(salary) FROM employees WHERE salary < (SELECT MAX(salary) FROM employees)`. It also handles "no second salary" by returning NULL.)

Ranking within partitions

`PARTITION BY` restarts the numbering in every group:

```
SELECT dept, name, salary,
       DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS dept_rank
FROM employees
ORDER BY dept, dept_rank, name;
```

dept	name	salary	dept_rank
Eng	Asha	90000	1
Eng	Farah	90000	1
Eng	Ben	70000	2
Sales	Chen	50000	1
Sales	Dev	40000	2

Each department is ranked on its own. This is the building block of "top N per group", which has its own lesson.

Making the ranking deterministic

`ROW_NUMBER` must give tied rows *different* numbers, and if the ORDER BY does not decide which gets which, the engine does — arbitrarily, and not necessarily the same way twice. A query that picks "row number 1" per customer can then return a different row tomorrow from the same data. Always finish a `ROW_NUMBER` ordering with a unique column:

```
ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date DESC, id DESC)
```

`RANK` and `DENSE_RANK` are deterministic by definition — ties get the same number — so they need no tie-breaker for the numbers themselves. But the *output order* of tied rows still needs one in the final ORDER BY.

Bucketing: NTILE

`NTILE(n)` deals the ordered rows of each partition into `n` buckets as evenly as possible and returns the bucket number:

```sql
SELECT name, salary,
       NTILE(4) OVER (ORDER BY salary DESC, name) AS quartile
FROM employees
ORDER BY salary DESC, name;
```

With 10 rows and 4 buckets, the sizes are 3, 3, 2, 2 — the remainder goes to the first buckets. NTILE does not respect ties: two equal salaries can land in different quartiles if the boundary falls between them, so give it a tie-breaker like ROW_NUMBER. Uses: quartiles of customers by spend, deciles of response time, splitting work into n batches.

Relative rank: PERCENT_RANK and CUME_DIST

Two functions express position as a fraction:

- `PERCENT_RANK()` = `(rank - 1) / (rows in partition - 1)`, from 0 for the first row to 1 for the last; ties share a value. A single-row partition gives 0.
- `CUME_DIST()` = (rows at or before this one, ties included) / (rows in partition), from just above 0 up to 1. "What fraction of employees earn at most this much?"

```
SELECT name, salary,
       ROUND(PERCENT_RANK() OVER (ORDER BY salary), 2) AS pct_rank,
       ROUND(CUME_DIST()   OVER (ORDER BY salary), 2) AS cume
FROM employees
ORDER BY salary, name;
```

Round both when you print them — they are fractions.

Ranking after grouping

Like every window function, rankings run after GROUP BY, so you can rank aggregates directly:

```
SELECT customer_id,
       SUM(amount) AS spend,
       RANK() OVER (ORDER BY SUM(amount) DESC) AS spend_rank
FROM orders
GROUP BY customer_id
ORDER BY spend_rank, customer_id;
```

And, because they run *before* the final ORDER BY and LIMIT, you can rank everything and then show only a page — though for a simple "top 10 overall", `ORDER BY … LIMIT 10` is simpler and faster. Ranking functions earn their place when there are partitions or ties to respect.

In other databases

The four functions and their tie rules are standard SQL and identical in PostgreSQL, SQL Server, Oracle and SQLite. SQL Server and Oracle also allow `ROW_NUMBER` to drive pagination in older code (`WHERE rn BETWEEN 11 AND 20`); MySQL and PostgreSQL would use `LIMIT … OFFSET`. In MySQL 5.7 rankings were faked with user variables (`@rn := @rn + 1`), a trick whose evaluation order MySQL never guaranteed and which 8.0 deprecates.

Pitfalls

- `RANK` where `DENSE_RANK` was meant, so "rank 2" does not exist after a tie.
- `ROW_NUMBER` without a unique tie-breaker, picking a different row each run.
- Aliasing a column `rank` without backticks.
- Filtering on a rank in WHERE instead of an outer query.
- Expecting NTILE buckets to keep ties together.

Key takeaways

- `ROW_NUMBER`: unique numbers. `RANK`: ties share, then gaps. `DENSE_RANK`: ties share, no gaps.
- The nth highest value is a `DENSE_RANK` question.
- `PARTITION BY` restarts ranking per group.
- Give `ROW_NUMBER` a unique tie-breaker so it is deterministic.
- `NTILE(n)` makes n near-equal buckets; `PERCENT_RANK` and `CUME_DIST` give position as a fraction.

EXERCISE 1

Three rankings side by side

Rank the players by score, highest first, in the three standard ways.

scores: player	score
Eva	80
Asha	90
Chen	85
Farah	70
Ben	90
Dev	80

Return `player`, `score`, `row_num` (`ROW_NUMBER` by score descending, ties broken by `player`), `rnk` (`RANK` by score descending) and `dense_rnk` (`DENSE_RANK` by score descending). Order by `score` descending, then `player`.

```
player | score | row_num | rnk | dense_rnk
Asha | 90 | 1 | 1 | 1
Ben | 90 | 2 | 1 | 1
Chen | 85 | 3 | 3 | 2
Dev | 80 | 4 | 4 | 3
Eva | 80 | 5 | 4 | 3
Farah | 70 | 6 | 6 | 4
```

INPUT 1

```
CREATE TABLE scores (player VARCHAR(20)
PRIMARY KEY, score INT);
INSERT INTO scores VALUES
  ('Eva', 80), ('Asha', 90), ('Chen',
85), ('Farah', 70), ('Ben', 90), ('Dev',
80);
```

OUTPUT 1

```
player | score | row_num | rnk |
dense_rnk
Asha | 90 | 1 | 1 | 1
Ben | 90 | 2 | 1 | 1
Chen | 85 | 3 | 3 | 2
Dev | 80 | 4 | 4 | 3
Eva | 80 | 5 | 4 | 3
Farah | 70 | 6 | 6 | 4
```

REFERENCE SOLUTION

```sql
SELECT player, score,
       ROW_NUMBER() OVER (ORDER BY score DESC, player) AS row_num,
       RANK() OVER (ORDER BY score DESC) AS rnk,
       DENSE_RANK() OVER (ORDER BY score DESC) AS dense_rnk
FROM scores
ORDER BY score DESC, player;
```

EXERCISE 2

Rank within each department

Rank employees by salary within their own department, highest first, so that equal salaries share a rank and the next salary gets the next number.

employees: id	name	dept	salary
1	Asha	Eng	90000
2	Ben	Eng	70000
3	Chen	Sales	50000
4	Dev	Sales	40000
5	Eva	HR	45000
6	Farah	Eng	90000
7	Gita	Eng	60000

Return `dept`, `name`, `salary` and `dept_rank`. Order by `dept`, `dept_rank`, then `name`.

```
dept | name | salary | dept_rank
Eng | Asha | 90000 | 1
Eng | Farah | 90000 | 1
Eng | Ben | 70000 | 2
Eng | Gita | 60000 | 3
HR | Eva | 45000 | 1
Sales | Chen | 50000 | 1
Sales | Dev | 40000 | 2
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(30), dept VARCHAR(20),
salary INT);
INSERT INTO employees VALUES
  (1, 'Asha', 'Eng', 90000), (2, 'Ben',
'Eng', 70000), (3, 'Chen', 'Sales',
50000),
  (4, 'Dev', 'Sales', 40000), (5, 'Eva',
'HR', 45000), (6, 'Farah', 'Eng',
90000), (7, 'Gita', 'Eng', 60000);
```

OUTPUT 1

```
dept | name | salary | dept_rank
Eng | Asha | 90000 | 1
Eng | Farah | 90000 | 1
Eng | Ben | 70000 | 2
Eng | Gita | 60000 | 3
HR | Eva | 45000 | 1
Sales | Chen | 50000 | 1
Sales | Dev | 40000 | 2
```

REFERENCE SOLUTION

```sql
SELECT dept, name, salary,
       DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS dept_rank
FROM employees
ORDER BY dept, dept_rank, name;
```

QUIZ

1. Salaries are 100, 100, 90, 80. What does `RANK() OVER (ORDER BY salary DESC)` give the 90?

A. 2

B. 3

C. 1

D. 4

2. Which function correctly finds the second highest *distinct* salary when the top salary is shared?

A. `ROW_NUMBER`

B. `RANK`

C. `DENSE_RANK`

D. `NTILE(2)`

3. Why add `id` to `ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date DESC, id DESC)`?

A. ROW_NUMBER requires at least two ORDER BY columns

B. Without a unique tie-breaker, two orders on the same date can be numbered either way, so row 1 is not deterministic

C. It makes the query use an index

D. It sorts the output by id

4. `NTILE(4)` over 10 ordered rows produces groups of which sizes?

A. 3, 3, 2, 2

B. 2, 2, 3, 3

C. 2, 2, 2, 4

D. 3, 3, 3, 1

ANSWER KEY · RANKING FUNCTIONS — ROW_NUMBER, RANK, DENSE_RANK AND NTILE

1. B RANK gives tied rows the same number and then skips: 1, 1, 3, 4. DENSE_RANK would give the 90 rank 2; ROW_NUMBER would give 3 but number the two 100s 1 and 2.

2. C With 100, 100, 90, DENSE_RANK gives 1, 1, 2, so rank 2 is 90. RANK gives 1, 1, 3 — no rank 2 at all — and ROW_NUMBER's row 2 is the second 100. NTILE splits rows into halves, not distinct values.

3. B ROW_NUMBER must give tied rows different numbers; if the ORDER BY does not decide, the engine does, arbitrarily. A unique column makes the choice repeatable. It does not order the output — the final ORDER BY does.

4. A NTILE makes the groups as equal as possible and gives the remainder to the first groups: 10 = 4 × 2 + 2, so the first two groups get one extra row.

LESSON 3 · 14 MIN

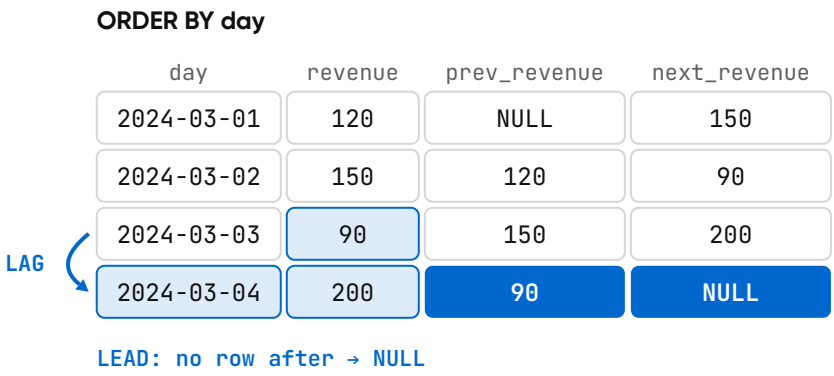
LAG and LEAD — comparing a row with its neighbours

Some questions are about change: how much did revenue move since yesterday, how many days passed between a customer's orders, did this reading go up or down. Each compares a row with the row *before* or *after* it in some order. Before window functions this took a self-join — join the table to itself on "the previous day" — which breaks the moment a day is missing. `LAG` and `LEAD` read a neighbouring row's value directly. This lesson covers both, their offset and default arguments, partitioning so comparisons stay inside one customer or product, the patterns they enable (change, growth, gaps, sessions) and their relatives `FIRST_VALUE`, `LAST_VALUE` and `NTH_VALUE`.

The previous and the next row

```
SELECT day, revenue,
       LAG(revenue)  OVER (ORDER BY day) AS prev_revenue,
       LEAD(revenue) OVER (ORDER BY day) AS next_revenue
FROM daily_sales
ORDER BY day;
```

LAG looks one row back, LEAD one row ahead



1. The window orders the rows by day; that order is what previous and next mean. The two new columns are filled one row at a time.
2. On 2024-03-01 (revenue 120): there is no row before the first one, so LAG gives NULL; LEAD reads the row after and copies 150.
3. On 2024-03-02 (revenue 150): LAG reads the row before, 2024-03-01, and copies its revenue 120; LEAD reads the row after and copies 90.
4. On 2024-03-03 (revenue 90): LAG reads the row before, 2024-03-02, and copies its revenue 150; LEAD reads the row after and copies 200.
5. On 2024-03-04 (revenue 200): LAG reads the row before, 2024-03-03, and copies its revenue 90; there is no row after the last one, so LEAD gives NULL.

`LAG(expr)` is `expr` evaluated on the previous row of the window's order; `LEAD(expr)` on the next. The first row has no previous row and the last no next one, so those cells are NULL. The window's `ORDER BY` is what "previous" means — it is required, and it should be unique (a tie on `day` would leave which row is "previous" up to the engine).

Offset and default

The full form is `LAG(expr, offset, default)`:

```
LAG(revenue, 7)      OVER (ORDER BY day)   -- the value a week earlier (7 rows
LAG(revenue, 1, 0)   OVER (ORDER BY day)   -- 0 instead of NULL on the first r
LEAD(status, 2, 'none') OVER (ORDER BY step)
```

`offset` counts *rows*, not days. `LAG(revenue, 7)` is "a week ago" only if there is exactly one row per day with no gaps — fill the gaps first (the recursive date series from the last module) or compare on dates with a join.

The default replaces the NULL produced by a missing row only. If the previous row exists and its value is NULL, LAG returns that NULL; the default does not apply.

Change and growth

```sql
SELECT day, revenue,
       revenue - LAG(revenue) OVER (ORDER BY day) AS change,
       ROUND(100 * (revenue - LAG(revenue) OVER (ORDER BY day))
                 / LAG(revenue) OVER (ORDER BY day), 1) AS growth_pct
FROM daily_sales
ORDER BY day;
```

Repeating the window three times is noisy; a named window or a CTE that computes `prev_revenue` once reads better:

```sql
WITH s AS (
  SELECT day, revenue, LAG(revenue) OVER (ORDER BY day) AS prev
  FROM daily_sales
)
SELECT day, revenue,
       revenue - prev AS change,
       ROUND(100 * (revenue - prev) / NULLIF(prev, 0), 1) AS growth_pct
FROM s
ORDER BY day;
```

`NULLIF(prev, 0)` turns a zero previous value into NULL, so the division deliberately gives NULL. MySQL's SELECT would return NULL with a warning anyway, but PostgreSQL and SQL Server raise a division-by-zero error, so the habit travels. The first row's change is NULL, which is correct — there is nothing to compare with.

Month-over-month growth is the same query over a monthly aggregate: group by `DATE_FORMAT(order_date, '%Y-%m')` in a CTE, then LAG over the months. Window functions run after GROUP BY, so you can even write `LAG(SUM(amount)) OVER (ORDER BY …)` in the grouped query itself.

Partitioning: neighbours within a group

Without `PARTITION BY`, the "previous row" of a customer's first order is someone else's last order. Partition to keep each comparison inside its group:

```
SELECT customer, order_date,
       LAG(order_date) OVER (PARTITION BY customer ORDER BY order_date, id) AS
       DATEDIFF(order_date,
                LAG(order_date) OVER (PARTITION BY customer ORDER BY order_dat
FROM orders
ORDER BY customer, order_date, id;
```

Each customer's first order has `prev_order` NULL and `days_since_prev` NULL. `DATEDIFF(a, b)` in MySQL is `a - b` in days, so the later date goes first.

Patterns built on LAG and LEAD

- **Gaps between events.** `DATEDIFF(day, LAG(day) …)` per user — churn analysis, "customers who came back after 30 days".
- **Streak and session boundaries.** Mark a row as the start of a new session when the gap since the previous event exceeds 30 minutes: `CASE WHEN TIMESTAMPDIFF(MINUTE, LAG(ts) OVER w, ts) > 30 OR LAG(ts) OVER w IS NULL THEN 1 ELSE 0 END`, then a running SUM of those flags numbers the sessions (frames lesson, and gaps-and-islands in the patterns module).
- **Status changes.** Keep the rows where `status <> LAG(status) OVER (PARTITION BY ticket ORDER BY changed_at)` — each transition once.
- **Duration of each state.** `LEAD(changed_at) OVER (PARTITION BY ticket ORDER BY changed_at)` is when this state ended; subtract to get how long it lasted. The last state's end is NULL — still open.
- **Increasing values.** `WHERE temperature > prev_temperature` in an outer query finds the days warmer than the day before — a classic interview exercise, best answered by comparing *dates* (`DATEDIFF(day, prev_day) = 1`) as well as values when days can be missing.

FIRST_VALUE, LAST_VALUE and NTH_VALUE

These return a value from a fixed position *in the window frame* rather than relative to the current row:

```sql
SELECT dept, name, salary,
       FIRST_VALUE(name) OVER (PARTITION BY dept ORDER BY salary DESC, name) A
       salary - FIRST_VALUE(salary) OVER (PARTITION BY dept ORDER BY salary DE
FROM employees
ORDER BY dept, salary DESC, name;
```

`FIRST_VALUE` with the default frame is the first row of the partition — what you usually want. `LAST_VALUE` is the trap: with an ORDER BY, the default frame ends at the *current row* (and its ties), so `LAST_VALUE` returns the current row's own value, not the partition's last. Either reverse the order and use `FIRST_VALUE`, or extend the frame: `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`. The frames lesson explains why. `NTH_VALUE(expr, n)` returns the nth row's value in the frame — NULL until the frame has n rows.

In other databases

`LAG`, `LEAD`, `FIRST_VALUE`, `LAST_VALUE` and `NTH_VALUE` are standard and behave identically in PostgreSQL, SQL Server (2012+), Oracle and SQLite. SQL Server's `DATEDIFF(day, start, end)` takes the unit first and subtracts the other way round from MySQL's `DATEDIFF(end, start)` — a frequent porting bug. Oracle and recent SQL Server versions also support `IGNORE NULLS` on LAG/LEAD (skip NULL values when looking back); MySQL parses `RESPECT NULLS` but does not implement `IGNORE NULLS`.

Pitfalls

- A non-unique window ORDER BY, making "previous" ambiguous.
- Forgetting PARTITION BY, so one customer's first row compares with another's last.
- Using a row offset as a time offset when days can be missing.
- Dividing by a previous value of 0 — use `NULLIF`.
- `LAST_VALUE` with the default frame returning the current row.

Key takeaways

- `LAG(expr, n, default)` reads n rows back, `LEAD` n rows ahead, in the window's order.
- Missing neighbours give NULL or the default; the first change in a series is NULL.

- Partition to keep comparisons inside one entity; order uniquely.
- Change, growth, gaps, sessions and state durations are all LAG/LEAD patterns.
- `FIRST_VALUE` reads the frame's first row; `LAST_VALUE` needs a full frame to mean the partition's last.

EXERCISE 1

Day-over-day change

Show each day's revenue beside the previous day's and the change between them.

daily_sales: day	revenue
2024-03-01	120
2024-03-02	150
2024-03-03	90
2024-03-04	200

Return `day`, `revenue`, `prev_revenue` (the revenue of the previous row by day; NULL for the first) and `change` (`revenue - prev_revenue`). Order by `day`.

```
day | revenue | prev_revenue | change
2024-03-01 | 120 | NULL | NULL
2024-03-02 | 150 | 120 | 30
2024-03-03 | 90 | 150 | -60
2024-03-04 | 200 | 90 | 110
```

A day's revenue can be NULL (not yet reported); then its change is NULL, and so is the next day's change.

INPUT 1

```
CREATE TABLE daily_sales (day DATE
PRIMARY KEY, revenue INT);
INSERT INTO daily_sales VALUES
  ('2024-03-01', 120), ('2024-03-02',
150), ('2024-03-03', 90), ('2024-03-04',
200);
```

OUTPUT 1

```
day | revenue | prev_revenue | change
2024-03-01 | 120 | NULL | NULL
2024-03-02 | 150 | 120 | 30
2024-03-03 | 90 | 150 | -60
2024-03-04 | 200 | 90 | 110
```

REFERENCE SOLUTION

```sql
SELECT day, revenue,
       LAG(revenue) OVER (ORDER BY day) AS prev_revenue,
       revenue - LAG(revenue) OVER (ORDER BY day) AS change
FROM daily_sales
ORDER BY day;
```

EXERCISE 2

Days between a customer's orders

For each order, show how many days passed since the same customer's previous order, and the date of their next one.

orders: id	customer	order_date
1	Asha	2024-01-05
2	Chen	2024-01-01
3	Asha	2024-03-01
4	Ben	2024-02-10
5	Asha	2024-01-20
6	Chen	2024-01-02

Return `customer`, `order_date`, `days_since_prev` (days since that customer's previous order, NULL for their first) and `next_order` (the date of their next order, NULL for their last). A customer's orders are in `order_date` order, then `id`. Order the result by `customer`, `order_date`, then `id`.

```
customer | order_date | days_since_prev | next_order
Asha | 2024-01-05 | NULL | 2024-01-20
Asha | 2024-01-20 | 15 | 2024-03-01
Asha | 2024-03-01 | 41 | NULL
Ben | 2024-02-10 | NULL | NULL
Chen | 2024-01-01 | NULL | 2024-01-02
Chen | 2024-01-02 | 1 | NULL
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
customer VARCHAR(20), order_date DATE);
INSERT INTO orders VALUES
  (1, 'Asha', '2024-01-05'), (2, 'Chen',
'2024-01-01'), (3, 'Asha', '2024-03-
01'),
  (4, 'Ben', '2024-02-10'), (5, 'Asha',
'2024-01-20'), (6, 'Chen', '2024-01-
02');
```

OUTPUT 1

```
customer | order_date | days_since_prev
| next_order
Asha | 2024-01-05 | NULL | 2024-01-20
Asha | 2024-01-20 | 15 | 2024-03-01
Asha | 2024-03-01 | 41 | NULL
Ben | 2024-02-10 | NULL | NULL
Chen | 2024-01-01 | NULL | 2024-01-02
Chen | 2024-01-02 | 1 | NULL
```

REFERENCE SOLUTION

```sql
SELECT customer, order_date,
       DATEDIFF(order_date, LAG(order_date) OVER w) AS days_since_prev,
       LEAD(order_date) OVER w AS next_order
FROM orders
WINDOW w AS (PARTITION BY customer ORDER BY order_date, id)
ORDER BY customer, order_date, id;
```

QUIZ

1. What does `LAG(revenue, 1, 0) OVER (ORDER BY day)` return on a row whose previous row's revenue is NULL?

A. 0

B. NULL

C. The current row's revenue

D. An error

2. Days are stored one row per day, but some days are missing. What does `LAG(visits, 7) OVER (ORDER BY day)` return?

A. The visits exactly seven calendar days earlier

B. The visits seven rows earlier, which may be more than seven days ago

C. NULL whenever a day is missing

D. The average of the last seven days

3. You compute `days_since_prev` with `LAG(order_date) OVER (ORDER BY order_date)` and no PARTITION BY. What goes wrong?

A. Nothing

B. Each customer's first order is compared with another customer's previous order

C. LAG returns only NULLs

D. The query is refused

4. With `OVER (PARTITION BY dept ORDER BY salary DESC)` and the default frame, what does `LAST_VALUE(name)` return?

A. The lowest-paid name in the department

B. The current row's name (or that of its last salary tie)

C. The highest-paid name

D. NULL

ANSWER KEY · LAG AND LEAD — COMPARING A ROW WITH ITS NEIGHBOURS

1. B The default (0) replaces only a *missing* row, such as the first row's. Here the previous row exists, and its value is NULL, so LAG returns NULL.

2. B The offset counts rows, not days. Fill the missing days first, or compare by date with a join or a RANGE frame.

3. B Without a partition, "previous" means the previous row in the whole table. `PARTITION BY customer` keeps each comparison inside one customer, and their first order gets NULL.

4. B With an ORDER BY, the default frame ends at the current row and its peers, so the frame's last row is (a tie of) the current row. Use `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`, or reverse the order and use FIRST_VALUE.

LESSON 4 · 16 MIN

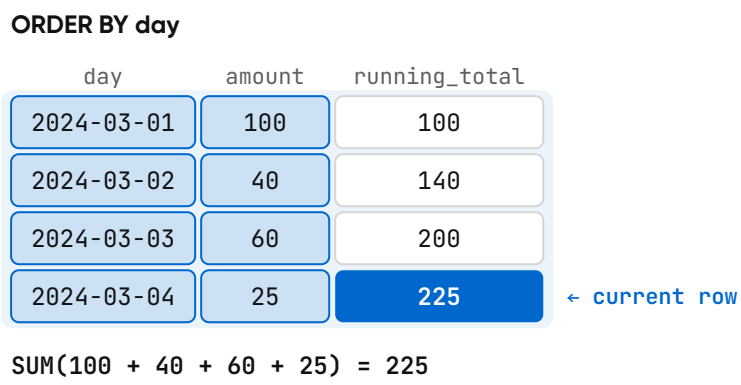
Running totals and frames — ROWS, RANGE and moving averages

A running balance, a cumulative count of sign-ups, a 7-day moving average of visits, the share of the year's target reached so far — each is an aggregate over a *sliding* set of rows: everything up to this one, or the last few. Window functions express these with a **frame**: a clause inside `OVER` that says which rows of the partition, relative to the current row, the aggregate sees. Frames are where window functions get subtle, because the default frame is not what most people assume and it behaves differently when the ORDER BY has ties. This lesson covers running totals, frame syntax, `ROWS` versus `RANGE`, moving averages, the default frame and its ties trap, and which functions use frames at all.

The running total

```
SELECT day, amount,
       SUM(amount) OVER (ORDER BY day ROWS BETWEEN UNBOUNDED PRECEDING AND CU
FROM daily_sales
ORDER BY day;
```

A running total: the frame grows by one row each step



1. The window orders the rows by day. The frame is every row from the partition's first through the current one, so it starts at one row and grows.
2. On 2024-03-01 the frame is that row alone, so the running total is its own amount, 100.
3. On 2024-03-02 the frame holds 2 rows, the first through this one; their sum, 140, is the running total so far.
4. On 2024-03-03 the frame holds 3 rows, the first through this one; their sum, 200, is the running total so far.
5. On 2024-03-04 the frame holds 4 rows, the first through this one; their sum, 225, is the running total so far.

The window is ordered by `day`, and the frame `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` means "from the first row of the partition through this row". Each row's SUM covers a frame one row larger than the last. Add `PARTITION BY account` and the total restarts for every account — a running balance per account in one pass, where the pre-window answer was a correlated subquery summing every earlier row (quadratic work).

Frame syntax

```
{ROWS | RANGE} BETWEEN start AND end

start / end:
  UNBOUNDED PRECEDING     -- the partition's first row
  n PRECEDING             -- n rows (or n in value) before
  CURRENT ROW
  n FOLLOWING             -- n rows (or n in value) after
  UNBOUNDED FOLLOWING     -- the partition's last row
```

The short form `ROWS UNBOUNDED PRECEDING` means `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Some frames worth knowing by heart:

Frame	Meaning
`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`	running total
`ROWS BETWEEN 2 PRECEDING AND CURRENT ROW`	this row and the two before (3-row moving window)
`ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`	centred 3-row window
`ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`	from here to the end (remaining total)
`ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`	the whole partition

The frame is clipped at the partition's edges: on the first row, "2 preceding" has no rows to include, so the frame is just the current row. A moving average's first values are therefore averages of fewer rows — be aware of it, and blank them out with `CASE WHEN ROW_NUMBER() OVER (…) ⩾ 3 THEN … END` if the report wants only full windows.

Moving averages

```
SELECT day, visits,
       ROUND(AVG(visits) OVER (ORDER BY day ROWS BETWEEN 2 PRECEDING AND CURRE
FROM daily_visits
ORDER BY day;
```

A moving average: a three-row frame slides down

ORDER BY day

day	visits	moving_avg_3	
2024-03-01	30	30	
2024-03-02	60	45	
2024-03-03	90	60	
2024-03-04	30	60	
2024-03-05	0	40	frame: 3 rows

AVG(90, 30, 0) = 40

1. On 2024-03-01 there is no row before it, so the frame is clipped at the partition's start: 1 row, average 30. Early values average fewer rows.
2. On 2024-03-02 there is only one row before it, so the frame is clipped at the partition's start: 2 rows, average 45. Early values average fewer rows.
3. On 2024-03-03 the frame is the full three rows, this one and the two before it; their average is 60. The frame moves down one row each step.
4. On 2024-03-04 the frame is the full three rows, this one and the two before it; their average is 60. The frame moves down one row each step.
5. On 2024-03-05 the frame is the full three rows, this one and the two before it; their average is 40. The frame moves down one row each step.

On 03-01 the frame is one row (30); on 03-02 two rows (30, 60 → 45); from 03-03 on, always three. `ROUND` matters: averages are rarely whole, and engines print different numbers of decimals. A moving SUM, MIN or MAX works the same way — "the highest price in the last five trades" is `MAX(price) OVER (ORDER BY traded_at, id ROWS BETWEEN 4 PRECEDING AND CURRENT ROW)`.

ROWS versus RANGE

`ROWS` counts physical rows in the window's order. `RANGE` works on the *value* of the ORDER BY expression:

- `RANGE … CURRENT ROW` includes every **peer** — every row with the same ORDER BY value as the current row.
- `RANGE BETWEEN 5 PRECEDING AND CURRENT ROW` with `ORDER BY price` includes rows whose price is within 5 of the current row's price, however many rows that is.
- On a date column, MySQL accepts intervals: `RANGE BETWEEN INTERVAL 6 DAY PRECEDING AND CURRENT ROW` is a true 7-*day* window even when some days have no rows (where `ROWS BETWEEN 6 PRECEDING` would reach back seven *rows*, possibly two weeks).

`RANGE` with a numeric or interval offset needs exactly one ORDER BY expression of a numeric or date type. PostgreSQL 11+ supports offsets too; SQL Server's RANGE accepts only `UNBOUNDED` and `CURRENT ROW`. There is also `GROUPS` (PostgreSQL, SQLite; not MySQL), which counts groups of peers.

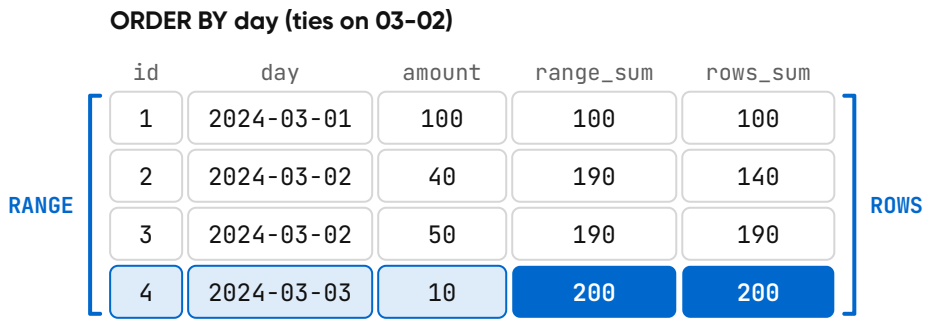
The default frame and the ties trap

If you write an ORDER BY in the window and no frame, the frame is:

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

That is a running total — *including all peers of the current row*. With a unique ORDER BY it is the same as the ROWS version. With ties, it is not:

ROWS counts rows; RANGE takes every peer of the current row



1. Rows 2 and 3 share the day 2024-03-02. With ORDER BY day and no frame, the default is RANGE up to the current row — which means up to its last peer.
2. On row 1 there are no ties: both frames hold just that row, and both sums are 100.
3. On row 2 the ROWS frame stops at row 2 itself: 140. The RANGE frame takes every row with the same day, so it reaches row 3 too: 190.
4. On row 3 both frames hold rows 1 to 3 and agree on 190 — so under RANGE the two tied rows show the same total, the total as of the end of that day.
5. On row 4 the tie is behind it and both frames are the whole table: 200. For a balance after each transaction, add a unique tie-breaker and say ROWS.

Rows 2 and 3 share a day, so under the default RANGE frame each one's frame includes the other: both show 190. Sometimes that is exactly right — "total as of the end of each day". When you want a balance after each *transaction*, add a unique tie-breaker and use ROWS. Write the frame explicitly whenever you care; readers should not have to remember the default.

Without an ORDER BY in the window, the default frame is the entire partition, which is why `SUM(x) OVER (PARTITION BY g)` is the group total.

The same default explains the `LAST_VALUE` surprise from the previous lesson: with an ORDER BY, the frame ends at the current row's last peer, so the "last value" in the frame is (a peer of) the current row. Use `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` for the partition's real last value.

Which functions use the frame

- **Use it:** aggregates as window functions (`SUM`, `AVG`, `COUNT`, `MIN`, `MAX`, …), `FIRST_VALUE`, `LAST_VALUE`, `NTH_VALUE`.
- **Ignore it:** `ROW_NUMBER`, `RANK`, `DENSE_RANK`, `NTILE`, `PERCENT_RANK`, `CUME_DIST`, `LAG`, `LEAD` — they always work on the whole ordered partition. MySQL accepts a frame clause on them and silently ignores it.

Running totals on top of aggregates

Frames apply after GROUP BY like every window, so a cumulative monthly total is a running SUM over a grouped query:

```sql
SELECT DATE_FORMAT(order_date, '%Y-%m') AS month,
       SUM(amount) AS revenue,
       SUM(SUM(amount)) OVER (ORDER BY DATE_FORMAT(order_date, '%Y-%m')
                              ROWS UNBOUNDED PRECEDING) AS cumulative
FROM orders
GROUP BY DATE_FORMAT(order_date, '%Y-%m')
ORDER BY month;
```

And a running *count* of flags numbers groups: `SUM(is_new_session) OVER (PARTITION BY user_id ORDER BY ts, id ROWS UNBOUNDED PRECEDING)` gives each event its session number — the core of the gaps-and-islands patterns later in the course.

Percent of running total

Combine a running sum with a whole-partition sum to show progress towards a total:

```sql
SELECT day, amount,
       ROUND(100 * SUM(amount) OVER (ORDER BY day ROWS UNBOUNDED PRECEDING)
                 / SUM(amount) OVER (), 1) AS pct_done
FROM daily_sales
ORDER BY day;
```

Two windows, two different frames, one row each — exactly the kind of report that once took three subqueries.

Pitfalls

- Relying on the default frame and getting equal running totals on tied rows.
- `ROWS n PRECEDING` used as "n days" when days can be missing.
- Forgetting that early rows of a moving average cover fewer rows.
- Unrounded moving averages.
- Expecting a frame clause to change ROW_NUMBER, RANK or LAG.

Key takeaways

- A frame chooses which rows of the partition an aggregate sees, relative to the current row.
- Running total: `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` with a unique ORDER BY.
- Moving window: `ROWS BETWEEN n PRECEDING AND CURRENT ROW`; round averages.
- `ROWS` counts rows; `RANGE` uses values and includes peers; `RANGE INTERVAL n DAY` is a true day window.
- With ORDER BY the default frame is RANGE up to the current row (ties included); without it, the whole partition.

EXERCISE 1

A running balance per account

Show the balance of each account after every transaction. Transactions on the same date are applied in `id` order, so each gets its own balance.

transactions: id	account	txn_date	amount
1	A	2024-03-01	100
2	A	2024-03-02	-40
3	B	2024-03-01	500
4	A	2024-03-02	25
5	B	2024-03-03	-200
6	A	2024-03-05	-10

Return `account`, `txn_date`, `amount` and `balance` (the sum of the account's amounts up to and including this transaction). Order by `account`, `txn_date`, then `id`.

```
account | txn_date | amount | balance
A | 2024-03-01 | 100 | 100
A | 2024-03-02 | -40 | 60
A | 2024-03-02 | 25 | 85
A | 2024-03-05 | -10 | 75
B | 2024-03-01 | 500 | 500
B | 2024-03-03 | -200 | 300
```

Note the two transactions on 2024-03-02: each has its own balance.

INPUT 1

```
CREATE TABLE transactions (id INT
PRIMARY KEY, account VARCHAR(10),
txn_date DATE, amount INT);
INSERT INTO transactions VALUES
  (1, 'A', '2024-03-01', 100), (2, 'A',
'2024-03-02', -40), (3, 'B', '2024-03-
01', 500),
  (4, 'A', '2024-03-02', 25), (5, 'B',
'2024-03-03', -200), (6, 'A', '2024-03-
05', -10);
```

OUTPUT 1

```
account | txn_date | amount | balance
A | 2024-03-01 | 100 | 100
A | 2024-03-02 | -40 | 60
A | 2024-03-02 | 25 | 85
A | 2024-03-05 | -10 | 75
B | 2024-03-01 | 500 | 500
B | 2024-03-03 | -200 | 300
```

REFERENCE SOLUTION

```sql
SELECT account, txn_date, amount,
       SUM(amount) OVER (PARTITION BY account ORDER BY txn_date, id
                         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS balanc
FROM transactions
ORDER BY account, txn_date, id;
```

EXERCISE 2

A 3-day moving average

Smooth the daily visit counts with a moving average over the current day and the two before it.

There is one row per day; the first two days average over the rows that exist.

daily_visits: day	visits
2024-03-01	30
2024-03-02	60
2024-03-03	90
2024-03-04	30
2024-03-05	0

Return `day`, `visits` and `moving_avg_3` (`ROUND`ed to 2 places). Order by `day`.

```
day | visits | moving_avg_3
2024-03-01 | 30 | 30
2024-03-02 | 60 | 45
2024-03-03 | 90 | 60
2024-03-04 | 30 | 60
2024-03-05 | 0 | 40
```

INPUT 1

```
CREATE TABLE daily_visits (day DATE
PRIMARY KEY, visits INT);
INSERT INTO daily_visits VALUES
  ('2024-03-01', 30), ('2024-03-02',
60), ('2024-03-03', 90), ('2024-03-04',
30), ('2024-03-05', 0);
```

OUTPUT 1

```
day | visits | moving_avg_3
2024-03-01 | 30 | 30
2024-03-02 | 60 | 45
2024-03-03 | 90 | 60
2024-03-04 | 30 | 60
2024-03-05 | 0 | 40
```

REFERENCE SOLUTION

```
SELECT day, visits,
       ROUND(AVG(visits) OVER (ORDER BY day ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
FROM daily_visits
ORDER BY day;
```

QUIZ

1. What is the default frame of `SUM(x) OVER (ORDER BY d)` ?

A. `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`

B. `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`

C. `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`

D. `RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`

2. Rows: (d=1, x=10), (d=2, x=5), (d=2, x=7). What does `SUM(x) OVER (ORDER BY d)` give the two d=2 rows?

A. 15 and 22

B. 22 and 22

C. 15 and 15

D. 5 and 7

3. Which frame is a true 7-day window on a `day DATE` column even when days are missing?

A. `ROWS BETWEEN 6 PRECEDING AND CURRENT ROW`

B. `RANGE BETWEEN INTERVAL 6 DAY PRECEDING AND CURRENT ROW`

C. `ROWS BETWEEN 7 PRECEDING AND CURRENT ROW`

D. `RANGE BETWEEN 6 PRECEDING AND 6 FOLLOWING`

4. Which function is *not* affected by a frame clause?

A. `SUM`

B. `FIRST_VALUE`

C. `ROW_NUMBER`

D. `AVG`

5. A 3-row moving average uses `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW`. What does the partition's first row get?

A. NULL

B. Its own value — the frame is clipped to one row

C. 0

D. The average of the first three rows

ANSWER KEY · RUNNING TOTALS AND FRAMES — ROWS, RANGE AND MOVING AVERAGES

1. B With an ORDER BY and no frame, the frame is RANGE from the partition's start to the current row *including its peers*. Without an ORDER BY it is the whole partition.

2. B The default RANGE frame includes every peer of the current row, so both d=2 rows see 10 + 5 + 7 = 22. A ROWS frame with a tie-breaker would give 15 and 22.

3. B RANGE measures by value, so an interval offset on a date covers calendar days whatever the number of rows. ROWS counts rows, which spans more than a week when days are missing.

4. C Ranking functions and LAG/LEAD always work on the whole ordered partition; MySQL accepts a frame on them and ignores it. Aggregates and FIRST_VALUE/LAST_VALUE/NTH_VALUE use the frame.

5. B A frame never extends past the partition's edges, so the first row averages only itself and the second averages two rows. Blank them out explicitly if the report wants full windows only.

LESSON 5 · 14 MIN

Top N per group — the most asked window-function question

"The top three products in each category", "the highest-paid employee in each department", "each customer's most recent order", "the two best scores per player": this shape — the best N rows *within each group* — is the single most common window-function question in interviews and one of the most common in real reporting. GROUP BY cannot answer it, because it returns one row per group and loses the other columns; `ORDER BY … LIMIT` cannot, because the limit applies to the whole result, not per group. The answer combines the last few lessons: rank within a partition, then filter the rank in an outer query. This lesson covers the pattern, choosing the ranking function for ties, the "latest row per group" variant, and the older answers you will still meet.

The pattern

```
WITH ranked AS (
  SELECT dept, name, salary,
         ROW_NUMBER() OVER (PARTITION BY dept ORDER BY salary DESC, id) AS rn
  FROM employees
)
SELECT dept, name, salary
FROM ranked
WHERE rn ⩽ 2
ORDER BY dept, salary DESC, name;
```

Three parts:

1. **Partition** by the group (`dept`).
2. **Order** within the group by "best first" (`salary DESC`), with a tie-breaker.
3. **Filter** the rank in the outer query (`WHERE rn ⩽ 2`).

The CTE is required, not a style choice: window functions are computed after WHERE, so `WHERE ROW_NUMBER() OVER (…) ⩽ 2` in the same SELECT is an error. A derived table works equally well (`SELECT … FROM (SELECT …, ROW_NUMBER() … AS rn FROM employees) AS r WHERE rn ⩽ 2`). Some databases have a shortcut — Snowflake, BigQuery and Teradata

support `QUALIFY rn ⩽ 2`, a WHERE for window functions — but MySQL, PostgreSQL and SQL Server do not.

Ties decide the function

`employees`

dept	name	salary
Eng	Asha	90000
Eng	Farah	90000
Eng	Ben	70000
Eng	Gita	60000

"Top 2 in Eng" has three defensible answers:

Function	rn values	`⩽ 2` returns
`ROW_NUMBER() … ORDER BY salary DESC, id`	1, 2, 3, 4	Asha, Farah — exactly two
`RANK()`	1, 1, 3, 4	Asha, Farah — two, but a third tie at 90000 would make three
`DENSE_RANK()`	1, 1, 2, 3	Asha, Farah, Ben — the top two *salary levels*

- Exactly N rows per group (a "top 3" widget with three slots): **ROW_NUMBER** with a meaningful tie-breaker — and know the tie-breaker is choosing who is left out.
- Everyone who placed in the top N, ties included (prize winners): **RANK**. The result may have more than N rows.
- The top N distinct values (the two highest salaries and everyone earning them): **DENSE_RANK**.

In an interview, say which one you chose and why; the question is often deliberately ambiguous to see whether you notice ties.

The latest row per group

The most common variant in production code: each customer's most recent order, each sensor's latest reading, each ticket's current status.

```
WITH latest AS (
  SELECT customer_id, id AS order_id, order_date, amount,
         ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date DESC,
  FROM orders
)
SELECT customer_id, order_id, order_date, amount
FROM latest
WHERE rn = 1
ORDER BY customer_id;
```

`ROW_NUMBER` is right here — "the" latest order is one row — and `id DESC` decides between two orders on the same date (the later-inserted one). The first row per group is the same query with `ASC`: each customer's *first* order, the classic start of a cohort analysis.

Top N per group with the group's other facts

Because the window keeps every row, the ranked CTE can carry anything else you need — the group's size, its total, the gap to the leader — computed in the same pass:

```
WITH ranked AS (
  SELECT category, product, revenue,
         DENSE_RANK() OVER (PARTITION BY category ORDER BY revenue DESC) AS rn
         SUM(revenue) OVER (PARTITION BY category) AS category_revenue
  FROM product_sales
)
SELECT category, product, revenue,
       ROUND(100 * revenue / category_revenue, 1) AS pct_of_category
FROM ranked
WHERE rnk ⩽ 3
ORDER BY category, rnk, product;
```

Compute everything that needs the whole partition *before* filtering; after `WHERE rnk ⩽ 3`, the other rows are gone.

If the ranking is over an aggregate — the top two customers by spend in each city — group first, then rank in the same SELECT or the next CTE:

```sql
WITH spend AS (
  SELECT city, customer_id, SUM(amount) AS total
  FROM orders
  GROUP BY city, customer_id
),
ranked AS (
  SELECT city, customer_id, total,
         RANK() OVER (PARTITION BY city ORDER BY total DESC) AS rnk
  FROM spend
)
SELECT city, customer_id, total FROM ranked WHERE rnk ⩽ 2 ORDER BY city, rnk;
```

Before window functions

You will meet these in older code and in "solve it without window functions" follow-ups.

N = 1 with a join to the group maximum:

```sql
SELECT e.dept, e.name, e.salary
FROM employees e
JOIN (SELECT dept, MAX(salary) AS top FROM employees GROUP BY dept) m
  ON m.dept = e.dept AND m.top = e.salary
ORDER BY e.dept, e.name;
```

This returns every employee tied at the top — RANK semantics.

Any N with a correlated count:

```sql
SELECT e.dept, e.name, e.salary
FROM employees e
WHERE (SELECT COUNT(*)
       FROM employees e2
       WHERE e2.dept = e.dept AND e2.salary > e.salary) < 2
ORDER BY e.dept, e.salary DESC, e.name;
```

"Fewer than two people in my department earn more than me" — again RANK semantics (ties pass together). Counting `COUNT(DISTINCT e2.salary)` instead gives DENSE_RANK semantics.

The subquery runs once per row, so it is quadratic per group without a good index; the window version sorts once.

MySQL 5.7 user variables (`@rn := IF(@dept = dept, @rn + 1, 1)`) also appear in old answers. Their evaluation order was never guaranteed, and MySQL 8 deprecates setting variables inside expressions — replace them with ROW_NUMBER whenever you touch such code.

Performance notes

The window approach reads the rows once and sorts them by `(group, order)` — an index on exactly those columns, such as `(dept, salary)` or `(customer_id, order_date, id)`, lets MySQL skip the sort. For "latest row per group" on a huge table with few groups, a per-group lookup (`LATERAL` join in MySQL 8.0.14+ and PostgreSQL, `CROSS APPLY` in SQL Server) that reads one index entry per group can beat ranking every row. Measure before reaching for it.

Pitfalls

- Filtering the rank in the same SELECT's WHERE.
- `ROW_NUMBER` without a tie-breaker — a different "top" row each run.
- Choosing ROW_NUMBER when ties should all be included, or RANK when exactly N are needed.
- Computing a group total after filtering to the top N.
- Forgetting the final ORDER BY — the CTE's window order is not the output order.

Key takeaways

- Top N per group = rank in a partition (CTE or derived table), then `WHERE rank ⩽ N` outside.
- ROW_NUMBER for exactly N, RANK for ties included, DENSE_RANK for the top N values.
- Latest or first row per group = `ROW_NUMBER() … ORDER BY date DESC, id DESC` and `rn = 1`.
- Carry partition totals in the CTE before filtering.
- The pre-window answers are a join to a grouped MAX (N = 1) and a correlated count.

EXERCISE 1

The top two salary levels per department

For each department, list every employee whose salary is one of the department's **two highest distinct salaries** — so everyone tied at a level is included.

employees: id	name	dept	salary
1	Asha	Eng	90000
2	Ben	Eng	70000
3	Chen	Sales	50000
4	Dev	Sales	40000
5	Eva	HR	45000
6	Farah	Eng	90000
7	Gita	Eng	60000
8	Hari	Sales	30000

Return `dept`, `name` and `salary`. Order by `dept`, `salary` descending, then `name`.

```
dept | name | salary
Eng | Asha | 90000
Eng | Farah | 90000
Eng | Ben | 70000
HR | Eva | 45000
Sales | Chen | 50000
Sales | Dev | 40000
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(30), dept VARCHAR(20),
salary INT);
INSERT INTO employees VALUES
  (1, 'Asha', 'Eng', 90000), (2, 'Ben',
'Eng', 70000), (3, 'Chen', 'Sales',
50000),
  (4, 'Dev', 'Sales', 40000), (5, 'Eva',
'HR', 45000), (6, 'Farah', 'Eng',
90000),
  (7, 'Gita', 'Eng', 60000), (8, 'Hari',
'Sales', 30000);
```

OUTPUT 1

```
dept | name | salary
Eng | Asha | 90000
Eng | Farah | 90000
Eng | Ben | 70000
HR | Eva | 45000
Sales | Chen | 50000
Sales | Dev | 40000
```

REFERENCE SOLUTION

```sql
WITH ranked AS (
  SELECT dept, name, salary,
         DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS lvl
  FROM employees
)
SELECT dept, name, salary
FROM ranked
WHERE lvl ⩽ 2
ORDER BY dept, salary DESC, name;
```

EXERCISE 2

Each customer's latest order

Show each customer's most recent order. When a customer has two orders on the same latest date, the one with the higher `id` counts as the latest.

orders: id	customer	order_date	amount
1	Asha	2024-01-05	300
2	Ben	2024-02-01	120
3	Asha	2024-03-01	80
4	Chen	2024-02-15	50
5	Ben	2024-02-01	200
6	Asha	2024-02-20	40

Return `customer`, `order_id`, `order_date` and `amount` — exactly one row per customer. Order by `customer`.

```
customer | order_id | order_date | amount
Asha | 3 | 2024-03-01 | 80
Ben | 5 | 2024-02-01 | 200
Chen | 4 | 2024-02-15 | 50
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
customer VARCHAR(20), order_date DATE,
amount INT);
INSERT INTO orders VALUES
  (1, 'Asha', '2024-01-05', 300), (2,
'Ben', '2024-02-01', 120), (3, 'Asha',
'2024-03-01', 80),
  (4, 'Chen', '2024-02-15', 50), (5,
'Ben', '2024-02-01', 200), (6, 'Asha',
'2024-02-20', 40);
```

OUTPUT 1

```
customer | order_id | order_date |
amount
Asha | 3 | 2024-03-01 | 80
Ben | 5 | 2024-02-01 | 200
Chen | 4 | 2024-02-15 | 50
```

REFERENCE SOLUTION

```sql
WITH latest AS (
  SELECT customer, id AS order_id, order_date, amount,
         ROW_NUMBER() OVER (PARTITION BY customer ORDER BY order_date DESC, id DESC) AS rn
  FROM orders
)
SELECT customer, order_id, order_date, amount
FROM latest
WHERE rn = 1
ORDER BY customer;
```

QUIZ

1. Eng salaries are 90, 90, 90, 70. Which ranking makes `WHERE r ⩽ 2` return exactly two Eng employees?

A. `RANK()`

B. `DENSE_RANK()`

C. `ROW_NUMBER()` with a tie-breaker

D. Any of them

2. Why does the top-N-per-group pattern need a CTE or derived table?

A. PARTITION BY only works inside a CTE

B. The rank must be filtered, and WHERE runs before window functions are computed

C. CTEs are faster

D. ROW_NUMBER cannot appear in a SELECT list

3. This pre-window query returns the highest-paid employees per department. What does it do when two people share the top salary?

```
SELECT e.dept, e.name, e.salary
FROM employees e
JOIN (SELECT dept, MAX(salary) AS top FROM employees GROUP BY dept) m
  ON m.dept = e.dept AND m.top = e.salary;
```

A. Returns one of them at random

B. Returns both

C. Returns neither

D. Raises an error

4. You need each category's top 3 products *and* each product's share of its category's revenue. Where should the category total be computed?

A. After filtering to the top 3, with SUM over the remaining rows

B. In the same CTE as the rank, before the filter, with `SUM(revenue) OVER (PARTITION BY category)`

C. In the final ORDER BY

D. It cannot be done in one query

ANSWER KEY · TOP N PER GROUP — THE MOST ASKED WINDOW-FUNCTION QUESTION

1. C ROW_NUMBER numbers 1, 2, 3, 4, so exactly two pass (the tie-breaker decides which). RANK gives 1, 1, 1, 4 — three pass. DENSE_RANK gives 1, 1, 1, 2 — all four pass.

2. B The window is computed after WHERE, so the filter on its value has to happen one level up. Some warehouses offer `QUALIFY` for this; MySQL does not.

3. B Both rows match the department's MAX, so both are returned — the same result as RANK() = 1. For exactly one per department you need a tie-breaker, which ROW_NUMBER expresses directly.

4. B Once the outer WHERE keeps only the top 3, the other products' revenue is gone, so a total computed afterwards would be the top 3's total. Compute every whole-partition value beside the rank first.

CHECKPOINT · 25 MIN

Checkpoint — Window functions

This checkpoint covers the whole module: window functions with `OVER`, partitions and the whole-result window `OVER ()`, where windows sit in the order of evaluation, `ROW_NUMBER`, `RANK`, `DENSE_RANK` and `NTILE` and how they treat ties, `LAG`, `LEAD` and `FIRST_VALUE`, frames with `ROWS` and `RANGE`, running totals and moving averages, and the top-N-per-group pattern.

How it works. Twelve questions and three queries. You need 70% on the questions and every query accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What is the difference between `GROUP BY dept` and `PARTITION BY dept`?
- For scores 90, 90, 80, what do `ROW_NUMBER`, `RANK` and `DENSE_RANK` give?
- Why can a window function not appear in WHERE, and how do you filter on one?
- What does `LAG(x, 1, 0)` return on a partition's first row, and on a row whose previous `x` is NULL?
- What is the default frame when a window has an ORDER BY, and what does it do with ties?
- Which frame gives a 3-row moving average, and why must its result be rounded?
- When does top N per group need `RANK` rather than `ROW_NUMBER`?

The three queries are the second-highest salary in every department (ties included), month-over-month revenue growth computed with `LAG` over a monthly aggregate, and each product's rank and gap to the leader of its category.

EXERCISE 1

The second-highest salary in every department

For each department, list the employees paid the department's **second-highest distinct salary**. Departments with only one distinct salary do not appear.

employees: id	name	dept	salary
1	Asha	Eng	90000
2	Farah	Eng	90000
3	Ben	Eng	70000
4	Gita	Eng	70000
5	Ivan	Eng	60000
6	Chen	Sales	50000
7	Dev	Sales	40000
8	Eva	HR	45000
9	Kai	Ops	30000
10	Lin	Ops	30000

Return `dept`, `name` and `salary`. Order by `dept`, then `name`.

```
dept | name | salary
Eng | Ben | 70000
Eng | Gita | 70000
Sales | Dev | 40000
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(30), dept VARCHAR(20),
salary INT);
INSERT INTO employees VALUES
  (1, 'Asha', 'Eng', 90000), (2,
'Farah', 'Eng', 90000), (3, 'Ben',
'Eng', 70000),
  (4, 'Gita', 'Eng', 70000), (5, 'Ivan',
'Eng', 60000), (6, 'Chen', 'Sales',
50000),
  (7, 'Dev', 'Sales', 40000), (8, 'Eva',
'HR', 45000), (9, 'Kai', 'Ops', 30000),
(10, 'Lin', 'Ops', 30000);
```

OUTPUT 1

```
dept | name | salary
Eng | Ben | 70000
Eng | Gita | 70000
Sales | Dev | 40000
```

REFERENCE SOLUTION

```sql
WITH ranked AS (
  SELECT dept, name, salary,
         DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS lvl
  FROM employees
)
SELECT dept, name, salary
FROM ranked
WHERE lvl = 2
ORDER BY dept, name;
```

EXERCISE 2

Month-over-month growth

Report revenue per month and its growth over the previous month that had orders.

orders: id	order_date	amount
1	2024-01-04	100
2	2024-01-19	50
3	2024-02-02	300
4	2024-03-11	200
5	2024-03-30	40
6	2024-04-15	240

Return `month` (`DATE_FORMAT(order_date, '%Y-%m')`), `revenue` (the month's sum), `prev_revenue` (the previous listed month's revenue, NULL for the first) and `growth_pct` (`100 * (revenue - prev_revenue) / prev_revenue`, `ROUND`ed to 1 place). Order by `month`.

```
month | revenue | prev_revenue | growth_pct
2024-01 | 150 | NULL | NULL
2024-02 | 300 | 150 | 100
2024-03 | 240 | 300 | -20
2024-04 | 240 | 240 | 0
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
order_date DATE, amount INT);
INSERT INTO orders VALUES
  (1, '2024-01-04', 100), (2, '2024-01-
19', 50), (3, '2024-02-02', 300), (4,
'2024-03-11', 200),
  (5, '2024-03-30', 40), (6, '2024-04-
15', 240);
```

OUTPUT 1

```
month | revenue | prev_revenue |
growth_pct
2024-01 | 150 | NULL | NULL
2024-02 | 300 | 150 | 100
2024-03 | 240 | 300 | -20
2024-04 | 240 | 240 | 0
```

REFERENCE SOLUTION

```sql
WITH monthly AS (
  SELECT DATE_FORMAT(order_date, '%Y-%m') AS month, SUM(amount) AS revenue
  FROM orders
  GROUP BY DATE_FORMAT(order_date, '%Y-%m')
),
compared AS (
  SELECT month, revenue, LAG(revenue) OVER (ORDER BY month) AS prev_revenue
  FROM monthly
)
SELECT month, revenue, prev_revenue,
       ROUND(100 * (revenue - prev_revenue) / prev_revenue, 1) AS growth_pct
FROM compared
ORDER BY month;
```

EXERCISE 3

Rank and gap to the category leader

Rank each product by price within its category (most expensive first, ties sharing a rank and leaving a gap after), and show how far below the category's most expensive product it is.

products: id	category	name	price
1	Audio	Headphones	4999
2	Audio	Speaker	2999
3	Audio	Earbuds	2999
4	Audio	Cable	299
5	Video	TV	39999
6	Video	Projector	25999
7	Gaming	Console	49999

Return `category`, `name`, `price`, `price_rank` and `gap_to_top` (the category's highest price minus this price). Order by `category`, `price_rank`, then `name`.

```
category | name | price | price_rank | gap_to_top
Audio | Headphones | 4999 | 1 | 0
Audio | Earbuds | 2999 | 2 | 2000
Audio | Speaker | 2999 | 2 | 2000
Audio | Cable | 299 | 4 | 4700
Gaming | Console | 49999 | 1 | 0
Video | TV | 39999 | 1 | 0
Video | Projector | 25999 | 2 | 14000
```

INPUT 1

```
CREATE TABLE products (id INT PRIMARY
KEY, category VARCHAR(20), name
VARCHAR(30), price DECIMAL(10,2));
INSERT INTO products VALUES
  (1, 'Audio', 'Headphones', 4999), (2,
'Audio', 'Speaker', 2999), (3, 'Audio',
'Earbuds', 2999),
  (4, 'Audio', 'Cable', 299), (5,
'Video', 'TV', 39999), (6, 'Video',
'Projector', 25999),
  (7, 'Gaming', 'Console', 49999);
```

OUTPUT 1

```
category | name | price | price_rank |
gap_to_top
Audio | Headphones | 4999 | 1 | 0
Audio | Earbuds | 2999 | 2 | 2000
Audio | Speaker | 2999 | 2 | 2000
Audio | Cable | 299 | 4 | 4700
Gaming | Console | 49999 | 1 | 0
Video | TV | 39999 | 1 | 0
Video | Projector | 25999 | 2 | 14000
```

REFERENCE SOLUTION

```sql
SELECT category, name, price,
       RANK() OVER w AS price_rank,
       FIRST_VALUE(price) OVER w - price AS gap_to_top
FROM products
WINDOW w AS (PARTITION BY category ORDER BY price DESC)
ORDER BY category, price_rank, name;
```

CHECKPOINT QUESTIONS

1. Which statement about `PARTITION BY` is true?

A. It collapses each partition to one row, like GROUP BY

B. It divides rows into groups for the window function but keeps every row

C. It sorts the output by the partition columns

D. It can only be used with ranking functions

2. Scores 70, 80, 80, 90 ranked `ORDER BY score DESC`. Which gives 1, 2, 2, 3?

A. `ROW_NUMBER`

B. `RANK`

C. `DENSE_RANK`

D. `NTILE(3)`

3. In which order are these evaluated?

A. WHERE → window functions → GROUP BY → ORDER BY

B. WHERE → GROUP BY → HAVING → window functions → ORDER BY

C. Window functions → WHERE → GROUP BY → ORDER BY

D. GROUP BY → WHERE → window functions → ORDER BY

4. Does the ORDER BY inside OVER decide the order of the query's output?

A. Yes, always

B. Only when there is no PARTITION BY

C. No — only the query's own ORDER BY guarantees an output order

D. Only for ROW_NUMBER

5. What does `LEAD(order_date) OVER (PARTITION BY customer ORDER BY order_date, id)` return on a customer's last order?

A. The same date

B. The next customer's first order date

C. NULL

D. An error

6. How do you compute month-over-month change on a table of individual orders?

A. `LAG(amount) OVER (ORDER BY order_date)` on the orders

B. Group by month in a CTE, then `LAG(revenue) OVER (ORDER BY month)` over the monthly rows

C. `SUM(amount) OVER (PARTITION BY month)`

D. A self-join on `order_date - 30`

7. Which frame turns `SUM(amount)` into a per-row running total that is correct even when dates tie (ordered by `day, id`)?

A. No frame (the default)

B. `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`

C. `RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`

D. `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`

8. `AVG(visits) OVER (ORDER BY day ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)` on visits 10, 20, 30, 40 — what is the value on the last row?

A. 25

B. 30

C. 40

D. 35

9. Which pattern returns each user's latest login, exactly one row per user?

A. `RANK() OVER (PARTITION BY user_id ORDER BY login_at DESC)` and keep rank 1

B. `ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY login_at DESC, id DESC)` and keep 1

C. `MAX(login_at) OVER ()`

D. `ORDER BY login_at DESC LIMIT 1`

10. Why must a window average be wrapped in `ROUND` before it is printed in a report?

A. AVG over a window returns text

B. Averages are rarely whole numbers, and engines print different numbers of decimal places

C. MySQL refuses unrounded window averages

D. ROUND makes the window faster

11. What does `FIRST_VALUE(price) OVER (PARTITION BY category ORDER BY price DESC)` give each row?

A. The row's own price

B. The highest price in the row's category

C. The lowest price in the category

D. The previous row's price

12. Which needs window functions rather than GROUP BY alone?

A. Total revenue per region

B. The number of orders per customer

C. Each order listed with its customer's total spend beside it

D. The highest salary per department (the value only)

ANSWER KEY · CHECKPOINT — WINDOW FUNCTIONS

1. B PARTITION BY defines each row's window and never removes rows. It does not order the output, and it works with aggregates, ranking and offset functions alike.

2. C DENSE_RANK gives ties the same number without gaps. RANK would give 1, 2, 2, 4; ROW_NUMBER 1, 2, 3, 4; NTILE(3) buckets 1, 1, 2, 3.

3. B Windows are computed on the rows left after WHERE, GROUP BY and HAVING, before the final ORDER BY and LIMIT. That is why WHERE cannot use them and why they can run over aggregates.

4. C The window's ORDER BY orders rows for the function. Rows may come out in that order by coincidence of the plan, but another window or a join can change it; write the final ORDER BY.

5. C There is no next row inside the partition, so LEAD returns its default, which is NULL unless a third argument gives one.

6. B LAG over raw orders compares individual orders, not months. Aggregate to one row per month first; then the previous row is the previous month.

7. B A ROWS frame counts physical rows, so each row's total stops at that row. The default RANGE frame would include tied peers, and the other two are a remaining total and a centred 3-row sum.

8. B The frame is the last row and the two before it: (20 + 30 + 40) / 3 = 30.

9. B ROW_NUMBER with a unique tie-breaker gives exactly one row 1 per user. RANK can return two rows for a user with two logins at the same time; `OVER ()` is a single global maximum; LIMIT 1 returns one row overall.

10. B An average of integers is usually a fraction, and MySQL, PostgreSQL and others display a different scale. Rounding to the precision the report needs makes the output exact and comparable.

11. B With a descending order, the first row of the partition is the most expensive one, and the default frame always starts at the partition's first row. LAG would give the previous row's price.

12. C Keeping each order row while showing a per-customer total is exactly what a window does: `SUM(amount) OVER (PARTITION BY customer_id)`. The others are one row per group, which GROUP BY returns directly.
