Module 15 of 16

Query patterns

The shapes real reports and interview questions take: gaps and islands, deduplication, pivots, rolling windows, the nth highest value, and date series that fill the missing days.

Fourteen modules taught the clauses one at a time. This one teaches the *patterns* — the recurring shapes that questions take once the clauses are known, each a few lines of SQL that an experienced developer recognises from the wording of the question. Interviews are built from them, and so is most real reporting.

Gaps and islands groups consecutive rows into runs with the row-number difference, numbers islands with LAG and a running SUM when "consecutive" means something more general, and finds the holes in a sequence with LEAD. Deduplication decides what makes two rows the same, finds duplicates with GROUP BY and a windowed COUNT, keeps one row per group with ROW_NUMBER, deletes the rest without tripping MySQL's error 1093, and adds the UNIQUE constraint that stops them returning. Pivoting turns rows into columns with conditional aggregation — MySQL has no PIVOT — and back with UNION ALL. Rolling windows are moving averages and seven-day sums with window frames, and the trap that a ROWS frame counts rows, not days. The nth highest value is four ways to write the most asked SQL question, with the NULL answer done right, per group too. Date series build calendars with recursive CTEs and LEFT JOIN data onto them so that empty days count as zero.

The exercises are the reports these patterns exist for: login streaks and missing ticket numbers, the latest contact per email and a duplicate purge, quarterly sales across columns and exam scores back into rows, a moving average and a true seven-day revenue window, the third-highest price and the second-best score per subject, daily orders with the empty days and a half-year of monthly signups. The checkpoint combines them: the longest attendance streak per student, a monthly status report with the empty months filled, and each department's top two pay levels side by side.

CONTENTS

1	Gaps and islands — finding runs and holes in ordered data	16 min
2	Deduplication — finding, keeping and deleting duplicate rows	15 min
3	Pivoting — turning rows into columns and back	15 min

4	Rolling windows — moving averages, sliding sums and period-over-period	16 min
5	The nth highest value — LIMIT OFFSET, DENSE_RANK and the NULL answer	15 min
6	Date series — calendars from recursive CTEs and filling missing days	15 min
✓	Checkpoint — Query patterns	30 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 · 16 MIN

Gaps and islands — finding runs and holes in ordered data

Ordered data has a shape that plain grouping cannot see. A user logged in on the 1st, 2nd and 3rd, then on the 5th: that is a three-day streak and a one-day streak, but nothing in the rows says so — the streak is a property of *neighbouring* rows. Ticket numbers 101 to 107 with 104 missing hold one hole. A sensor was on for six readings, off for two, on again. These questions are called **gaps and islands**: the islands are the unbroken runs, the gaps are the holes between them. They are among the most common "hard" SQL interview questions and they turn up constantly in real reporting — streaks, sessions, outages, sequence audits.

Two tools from the window-function module solve almost all of them: the **row-number difference** for islands, and **LAG**/**LEAD** for gaps and run boundaries.

The row-number difference

Take one user's login dates, already distinct and ordered, and number them:

```sql
SELECT login_date,
       ROW_NUMBER() OVER (ORDER BY login_date) AS rn
FROM logins
WHERE user_id = 1;
```

login_date	rn
2024-03-01	1
2024-03-02	2
2024-03-03	3
2024-03-05	4
2024-03-06	5

Now subtract `rn` days from each date:

```sql
SELECT login_date,
       DATE_SUB(login_date, INTERVAL ROW_NUMBER() OVER (ORDER BY login_date) D
FROM logins
WHERE user_id = 1;
```

login_date	grp
2024-03-01	2024-02-29
2024-03-02	2024-02-29
2024-03-03	2024-02-29
2024-03-05	2024-03-01
2024-03-06	2024-03-01

Inside a run, the date and the row number both rise by one per row, so their difference is constant. At the hole (the 4th is missing), the date jumps by two while the row number still rises by one, so the difference changes. `grp` means nothing on its own — it is just a label every row of one island shares. Group by it:

```sql
WITH numbered AS (
  SELECT user_id, login_date,
         DATE_SUB(login_date, INTERVAL ROW_NUMBER() OVER (
           PARTITION BY user_id ORDER BY login_date) DAY) AS grp
  FROM (SELECT DISTINCT user_id, login_date FROM logins) AS d
)
SELECT user_id,
       MIN(login_date) AS start_date,
       MAX(login_date) AS end_date,
       COUNT(*)        AS days
FROM numbered
GROUP BY user_id, grp
ORDER BY user_id, start_date;
```

Three details make this correct in general. `PARTITION BY user_id` restarts the numbering per user, so one user's streak never runs into another's. `SELECT DISTINCT` first: a user who logged in twice on the 2nd would otherwise get two row numbers for one date, and the difference would drop by one and split the streak. (`DENSE_RANK()` instead of `ROW_NUMBER()` handles duplicates too, because equal dates share a rank.) And the grouping includes `user_id` as well as `grp`, since two users can share a `grp` value by coincidence.

The same trick works on integers — `id - ROW_NUMBER() OVER (ORDER BY id)` — and on anything you can count in steps. For "consecutive rows where the status was `down`", number all rows and number the `down` rows separately; the difference of the two row numbers is constant within each run of `down`:

```
SELECT reading_no, status,
       ROW_NUMBER() OVER (ORDER BY reading_no)
     - ROW_NUMBER() OVER (PARTITION BY status ORDER BY reading_no) AS grp
FROM readings;
```

Group by `status, grp` and every run of equal statuses becomes one row, even when the reading numbers have no holes at all. That two-row-number version is the general form; the date version is the special case where the value itself counts in steps.

Flagging run starts with LAG

The other approach marks where each island *starts* and then counts the starts with a running sum. A row starts a new island when the previous row is not its immediate predecessor:

```
WITH flagged AS (
  SELECT user_id, login_date,
         CASE WHEN DATEDIFF(login_date,
                LAG(login_date) OVER (PARTITION BY user_id ORDER BY login_date
              THEN 0 ELSE 1 END AS is_start
  FROM (SELECT DISTINCT user_id, login_date FROM logins) AS d
),
islands AS (
  SELECT user_id, login_date,
         SUM(is_start) OVER (PARTITION BY user_id ORDER BY login_date) AS isla
  FROM flagged
)
SELECT user_id, island_no, MIN(login_date) AS start_date, COUNT(*) AS days
FROM islands
GROUP BY user_id, island_no
ORDER BY user_id, island_no;
```

The first row of each user has no previous row, `LAG` returns NULL, `DATEDIFF` returns NULL, the comparison is not true, and the `ELSE 1` makes it a start — exactly right. The running `SUM` turns the flags into island numbers 1, 2, 3…

This version is longer, but it generalises better: the "same island" rule can be anything you can write in a `CASE`. Sessions that break after 30 idle minutes, prices that stay within 5% of the previous one, a status that has not changed — all are "start a new group when the condition between this row and the last one fails". Interviewers like seeing both versions, and like hearing which one you would choose when the rule is not "exactly one more".

Finding the gaps

Gaps are the space between one island's end and the next island's start. With `LEAD`, compare each value with the next one:

```sql
SELECT ticket_no + 1 AS gap_start,
       next_no - 1   AS gap_end
FROM (
  SELECT ticket_no,
         LEAD(ticket_no) OVER (ORDER BY ticket_no) AS next_no
  FROM tickets
) AS t
WHERE next_no - ticket_no > 1
ORDER BY gap_start;
```

For tickets 101, 102, 103, 105, 106, 110 this returns `104 | 104` and `107 | 109`. The last row's `next_no` is NULL, the subtraction is NULL, and the `WHERE` drops it — no special case needed. If the sequence must also start at a known value (say every batch begins at 1), a hole before the first row is a separate question: compare `MIN(ticket_no)` with the expected start.

Before window functions, the classic answer was a self-join or `NOT EXISTS`: "values `n` where `n + 1` does not exist, and `n` is not the maximum" give the gap starts, and a correlated `MIN` of the larger values gives each gap's end. It still works and is worth recognising in older code, but it reads each row several times where `LEAD` reads it once.

The longest streak

A frequent follow-up asks for each user's *longest* streak. Build the islands as a CTE, then rank them:

```sql
WITH islands AS ( /* user_id, start_date, end_date, days — as above */ ),
ranked AS (
  SELECT islands.*,
         ROW_NUMBER() OVER (PARTITION BY user_id
                            ORDER BY days DESC, start_date) AS rk
  FROM islands
)
SELECT user_id, start_date, end_date, days
FROM ranked
WHERE rk = 1;
```

The `start_date` in the ordering is the tie-breaker the question should specify — "if two streaks are equally long, report the earliest". Saying it aloud is part of the answer.

In other databases

The pattern is portable; only the date arithmetic changes. PostgreSQL subtracts an integer from a date directly (`login_date - rn::int`, or `login_date - rn * INTERVAL '1 day'`), and SQL Server writes `DATEADD(day, -rn, login_date)`. MySQL's `DATE_SUB(d, INTERVAL n DAY)` accepts the window function inside the interval expression, as above. `LAG`, `LEAD`, `ROW_NUMBER` and `DENSE_RANK` behave the same in all of them (MySQL from 8.0; on 5.7 these problems were solved with user variables, a technique worth recognising in legacy code but not writing anew).

How to say it in an interview

Name the pattern first: "This is gaps and islands." Then say which version you are writing and why: the row-number difference when consecutive means "exactly one step apart", the LAG flag and running sum when the rule is more general. Mention duplicates before you are asked — they are the usual hidden test — and finish with the ordering and the tie-breaker. That is the whole problem, and it is one most candidates find hard only because they have not seen the trick.

EXERCISE 1

Streaks of three days or more

The `logins` table records each day a user signed in. A user can sign in more than once on the same day, so a date can appear twice.

user_id	login_date
1	2024-03-01
1	2024-03-02
1	2024-03-03
1	2024-03-05
…	…
2	2024-03-02
2	2024-03-02

List every **streak of at least three consecutive days** for each user. Return the columns `user_id`, `start_date`, `end_date` and `days` (the number of distinct days in the streak), ordered by `user_id`, then `start_date`.

For the setup shown in the first test case the result is:

```
user_id | start_date | end_date | days
1 | 2024-03-01 | 2024-03-03 | 3
1 | 2024-03-05 | 2024-03-07 | 3
2 | 2024-03-01 | 2024-03-04 | 4
```

INPUT 1

```
CREATE TABLE logins (user_id INT,
login_date DATE);
INSERT INTO logins VALUES
  (1, '2024-03-01'), (1, '2024-03-02'),
(1, '2024-03-03'),
  (1, '2024-03-05'), (1, '2024-03-06'),
(1, '2024-03-07'),
  (2, '2024-03-01'), (2, '2024-03-02'),
(2, '2024-03-02'),
  (2, '2024-03-03'), (2, '2024-03-04'),
  (3, '2024-03-10'), (3, '2024-03-12');
```

OUTPUT 1

```
user_id | start_date | end_date | days
1 | 2024-03-01 | 2024-03-03 | 3
1 | 2024-03-05 | 2024-03-07 | 3
2 | 2024-03-01 | 2024-03-04 | 4
```

REFERENCE SOLUTION

```sql
WITH days AS (
  SELECT DISTINCT user_id, login_date FROM logins
),
numbered AS (
  SELECT user_id, login_date,
         DATE_SUB(login_date, INTERVAL ROW_NUMBER() OVER (
           PARTITION BY user_id ORDER BY login_date) DAY) AS grp
  FROM days
)
SELECT user_id,
       MIN(login_date) AS start_date,
       MAX(login_date) AS end_date,
       COUNT(*) AS days
FROM numbered
GROUP BY user_id, grp
HAVING COUNT(*) >= 3
ORDER BY user_id, start_date;
```

EXERCISE 2

Missing ticket numbers

Support tickets are numbered in sequence, but some numbers were never used or were deleted.

ticket_no	subject
101	Login fails
102	Refund
103	Slow page
105	Typo
106	Card declined
110	Password reset

Report each **gap** between the lowest and the highest ticket number as a range: the columns `gap_start` and `gap_end` (equal when a single number is missing), ordered by `gap_start`.

```
gap_start | gap_end
104 | 104
107 | 109
```

INPUT 1

```
CREATE TABLE tickets (ticket_no INT
PRIMARY KEY, subject VARCHAR(40));
INSERT INTO tickets VALUES
  (101, 'Login fails'), (102, 'Refund'),
(103, 'Slow page'),
  (105, 'Typo'), (106, 'Card declined'),
(110, 'Password reset');
```

OUTPUT 1

```
gap_start | gap_end
104 | 104
107 | 109
```

REFERENCE SOLUTION

```sql
SELECT ticket_no + 1 AS gap_start,
       next_no - 1   AS gap_end
FROM (
  SELECT ticket_no,
         LEAD(ticket_no) OVER (ORDER BY ticket_no) AS next_no
  FROM tickets
) AS t
WHERE next_no - ticket_no > 1
ORDER BY gap_start;
```

QUIZ

1. For one user's distinct login dates 2024-03-01, 03-02, 03-03, 03-05, what does `DATE_SUB(login_date, INTERVAL ROW_NUMBER() OVER (ORDER BY login_date) DAY)` give?

A. 2024-02-29 for all four rows

B. 2024-02-29 for the first three rows and 2024-03-01 for the fourth

C. Four different dates

D. 2024-03-01 for all four rows

2. A user has two rows for 2024-03-02 inside a run of 03-01 to 03-04. What happens if you apply the row-number difference without removing duplicates?

A. Nothing — ROW_NUMBER skips equal dates

B. The run splits in two, because the duplicate gets its own row number while the date stays the same

C. The query raises an error

D. The run is counted with five days but stays one group

3. When is the LAG flag plus a running `SUM` better than the row-number difference?

A. Never — they are always interchangeable

B. When "same island" is a rule other than "exactly one step apart", such as a session that breaks after 30 idle minutes

C. Only on MySQL 5.7

D. When the table has no primary key

4. In the gap query `WHERE next_no - ticket_no > 1`, what happens to the last ticket, whose `LEAD` is NULL?

A. It produces a gap from the ticket to NULL

B. It is dropped, because NULL minus a number is NULL and the comparison is not true

C. The query fails

D. It is reported as a gap of one number

ANSWER KEY · GAPS AND ISLANDS — FINDING RUNS AND HOLES IN ORDERED DATA

1. B Along the run the date and the row number both rise by one, so the difference stays at 2024-02-29. On the 5th the date jumps by two while the row number rises by one, so the difference moves to 2024-03-01 — a new island.

2. B ROW_NUMBER numbers every row, so after the duplicate the row number is one ahead of the date and the difference drops by a day. Use `SELECT DISTINCT` first, or `DENSE_RANK()`, which gives equal dates the same number.

3. B The flag version starts a new group whenever a condition between this row and the previous one fails, and the condition can be anything a CASE can express. The difference trick only works when consecutive values differ by exactly one step.

4. B Arithmetic with NULL yields NULL, and `NULL > 1` is unknown, which WHERE treats as false. No special case is needed for the last row.

LESSON 2 · 15 MIN

Deduplication — finding, keeping and deleting duplicate rows

Duplicates arrive in every real table sooner or later: a form submitted twice, an import run twice, the same customer typed as `Asha@Mail.com` and `asha@mail.com`. Dealing with them is three separate questions, and interviews ask all three: *which rows are duplicated*, *show me the data with one row per thing*, and *delete the extras*. Before any of them comes the one that decides the answer — **what makes two rows the same?**

Decide what "the same" means

Two rows are rarely identical in every column; the `id` alone differs. Duplicates are rows equal on a chosen set of columns — the *business key*: the email, or the customer plus the product plus the day. Write that set down first; it becomes the `GROUP BY` or the `PARTITION BY` in every query below.

Then decide how dirty the key is. If emails differ only in case and stray spaces, compare a normalised form:

```
SELECT LOWER(TRIM(email)) AS email_key, COUNT(*) AS copies
FROM contacts
GROUP BY LOWER(TRIM(email))
HAVING COUNT(*) > 1;
```

In MySQL a plain `GROUP BY email` would already ignore case under the default case-insensitive collation (`utf8mb4_0900_ai_ci`) — but not the spaces, and not on a database whose collation is case-sensitive. `LOWER(TRIM(…))` says what you mean on every engine.

Finding the duplicates

`GROUP BY … HAVING COUNT(*) > 1` gives one line per duplicated key — the right answer to "which emails are duplicated, and how often?". To see the duplicate *rows* themselves, with their ids and other columns, count inside a window instead so that no row is collapsed:

```sql
SELECT *
FROM (
  SELECT c.*,
         COUNT(*) OVER (PARTITION BY LOWER(TRIM(email))) AS copies
  FROM contacts AS c
) AS t
WHERE copies > 1
ORDER BY LOWER(TRIM(email)), id;
```

The older equivalent joins the table to the grouped duplicate keys (`JOIN (SELECT email … HAVING COUNT(*) > 1) d USING (email)`), or uses `EXISTS` for "another row with the same email and a different id". All three are correct; the window version reads the table once.

Keeping one row per group

To *read* the data deduplicated, number the rows of each group in the order of preference and keep number 1:

```sql
WITH ranked AS (
  SELECT c.*,
         ROW_NUMBER() OVER (
           PARTITION BY LOWER(TRIM(email))
           ORDER BY updated_at DESC, id DESC) AS rn
  FROM contacts AS c
)
SELECT id, email, name
FROM ranked
WHERE rn = 1
ORDER BY email;
```

The `ORDER BY` inside `OVER` is the *keeper rule*: the most recently updated row, and if two were updated at the same moment, the higher id. Always make that rule total — end it with a unique column — or the row that survives can change between runs. This "latest row per key" query is one of the most used queries in practice; it is how a slowly-changing history becomes a current view.

`SELECT DISTINCT` is the right tool only when the rows are duplicates in *every* selected column. `SELECT DISTINCT email, name` keeps both `asha@mail.com | Asha` and `asha@mail.com | Asha K` — DISTINCT cannot choose between rows that differ.

There is also the aggregate shortcut, `SELECT email, MAX(updated_at) … GROUP BY email`, which gives the latest *timestamp* but not the rest of that row. Asking for `name` in the same query either fails under `ONLY_FULL_GROUP_BY` (the MySQL 8 default) or, with that mode off, returns a name from an arbitrary row. Join the maximum back to the table, or use `ROW_NUMBER`.

Deleting the extras

Deleting duplicates means deleting every row that is not the keeper. The portable way names the rows to delete by id:

```
DELETE FROM orders
WHERE id IN (
  SELECT id FROM (
    SELECT id,
           ROW_NUMBER() OVER (
             PARTITION BY customer_id, product, ordered_on
             ORDER BY id) AS rn
    FROM orders
  ) AS numbered
  WHERE rn > 1
);
```

The extra level — `SELECT id FROM (…) AS numbered` — is not decoration. MySQL refuses a `DELETE` or `UPDATE` whose subquery reads the table being changed, with error 1093: *You can't specify target table 'orders' for update in FROM clause*. Wrapping the subquery in a derived table makes MySQL materialise it first, and the error goes away. (PostgreSQL and SQL Server have no such restriction, which is why this wrapper looks odd to people coming from them.)

MySQL also has a multi-table `DELETE` that does the same with a self-join, and it is the classic short answer for "keep the lowest id":

```
DELETE o1
FROM orders AS o1
JOIN orders AS o2
  ON  o1.customer_id = o2.customer_id
  AND o1.product     = o2.product
  AND o1.ordered_on  = o2.ordered_on
  AND o1.id > o2.id;
```

Every row that has a twin with a smaller id is deleted, so exactly the smallest id of each group survives. The `DELETE o1 FROM …` form is MySQL's (SQL Server spells it `DELETE o1 FROM … JOIN …` too; PostgreSQL uses `DELETE FROM orders o1 USING orders o2 WHERE …`). The `ROW_NUMBER` version runs everywhere and states the keeper rule more plainly, so it is the one to write when the rule is anything other than "smallest id".

Run a destructive deduplication inside a transaction, check the row count against the `SELECT` that found the duplicates, and only then `COMMIT`.

NULLs in the key

`GROUP BY`, `DISTINCT` and `PARTITION BY` all treat NULLs as one group, so two rows with a NULL phone number count as duplicates of each other in every query above. Whether they *are* duplicates is a business question — two contacts with no phone number are usually different people. If they are not duplicates, exclude them (`WHERE phone IS NOT NULL`) or partition by a key that cannot be NULL.

Stopping them coming back

Cleaning a table once is half the job. A `UNIQUE` constraint on the business key makes the next duplicate fail at insert time:

```
ALTER TABLE orders
  ADD CONSTRAINT uq_order UNIQUE (customer_id, product, ordered_on);
```

The constraint can only be added after the existing duplicates are gone. From then on, an import can use `INSERT IGNORE` to skip rows that would collide, or `INSERT … ON DUPLICATE KEY UPDATE` to update the existing row instead. Remember that a MySQL unique index allows any number of NULLs, so a nullable key column does not prevent NULL-keyed duplicates.

How to say it in an interview

State the duplicate key and the keeper rule before writing anything: "rows are duplicates when customer, product and day match; we keep the smallest id." Then: GROUP BY with HAVING to find, `ROW_NUMBER` with `rn = 1` to read, `rn > 1` (wrapped for MySQL) or the self-

join to delete, and a UNIQUE constraint so it does not happen again. Mentioning error 1093 unprompted is a strong signal that you have done this on a real MySQL server.

EXERCISE 1

The latest contact per email

The `contacts` table was filled by several imports, so the same person appears more than once — with different capitalisation and stray spaces in the email.

id	email	name	updated_at
1	asha@mail.com	Asha	2024-01-10
2	␣Asha@Mail.com	Asha K	2024-03-02
3	ravi@mail.com	Ravi	2024-02-01
4	meena@mail.com	Meena	2024-02-01
5	MEENA@mail.com␣	Meena S	2024-02-01
6	ravi@mail.com	Ravi P	2024-01-15

(␣ marks a space.) Two contacts are the same person when their emails match after `LOWER(TRIM(…))`. Keep **one row per person**: the one with the latest `updated_at`, and when two share that date, the one with the higher `id`.

Return `email` (lower-cased and trimmed), `id` and `name`, ordered by `email`.

```
email | id | name
asha@mail.com | 2 | Asha K
meena@mail.com | 5 | Meena S
ravi@mail.com | 3 | Ravi
```

INPUT 1

```
CREATE TABLE contacts (id INT PRIMARY
KEY, email VARCHAR(60), name
VARCHAR(30), updated_at DATE);
INSERT INTO contacts VALUES
  (1, 'asha@mail.com',    'Asha',
'2024-01-10'),
  (2, ' Asha@Mail.com',   'Asha K',
'2024-03-02'),
  (3, 'ravi@mail.com',    'Ravi',
'2024-02-01'),
  (4, 'meena@mail.com',   'Meena',
'2024-02-01'),
  (5, 'MEENA@mail.com ',  'Meena S',
'2024-02-01'),
  (6, 'ravi@mail.com',    'Ravi P',
'2024-01-15');
```

OUTPUT 1

```
email | id | name
asha@mail.com | 2 | Asha K
meena@mail.com | 5 | Meena S
ravi@mail.com | 3 | Ravi
```

REFERENCE SOLUTION

```
WITH ranked AS (
  SELECT LOWER(TRIM(email)) AS email, id, name,
         ROW_NUMBER() OVER (
           PARTITION BY LOWER(TRIM(email))
           ORDER BY updated_at DESC, id DESC) AS rn
  FROM contacts
)
SELECT email, id, name
FROM ranked
WHERE rn = 1
ORDER BY email;
```

EXERCISE 2

Delete the duplicate orders

A checkout bug saved some orders twice. Two orders are duplicates when they have the same `customer_id`, `product` and `ordered_on`.

id	customer_id	product	ordered_on
1	10	pen	2024-04-01
2	10	pen	2024-04-01
3	10	pen	2024-04-02
4	11	pen	2024-04-01
5	10	ink	2024-04-01
6	10	pen	2024-04-01
7	11	pen	2024-04-01

Write a `DELETE` that removes every duplicate and **keeps the order with the smallest `id`** of each group. The starter already ends with the `SELECT` that shows the table afterwards; keep it.

```
id | customer_id | product | ordered_on
1 | 10 | pen | 2024-04-01
3 | 10 | pen | 2024-04-02
4 | 11 | pen | 2024-04-01
5 | 10 | ink | 2024-04-01
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, product VARCHAR(20),
ordered_on DATE);
INSERT INTO orders VALUES
  (1, 10, 'pen', '2024-04-01'),
  (2, 10, 'pen', '2024-04-01'),
  (3, 10, 'pen', '2024-04-02'),
  (4, 11, 'pen', '2024-04-01'),
  (5, 10, 'ink', '2024-04-01'),
  (6, 10, 'pen', '2024-04-01'),
  (7, 11, 'pen', '2024-04-01');
```

OUTPUT 1

```
id | customer_id | product | ordered_on
1 | 10 | pen | 2024-04-01
3 | 10 | pen | 2024-04-02
4 | 11 | pen | 2024-04-01
5 | 10 | ink | 2024-04-01
```

REFERENCE SOLUTION

```sql
DELETE FROM orders
WHERE id IN (
  SELECT id FROM (
    SELECT id,
           ROW_NUMBER() OVER (
             PARTITION BY customer_id, product, ordered_on
             ORDER BY id) AS rn
    FROM orders
  ) AS numbered
  WHERE rn > 1
);

SELECT id, customer_id, product, ordered_on FROM orders ORDER BY id;
```

QUIZ

1. Which query lists each duplicated email once with its number of copies?

A. `SELECT DISTINCT email FROM users`

B. `SELECT email, COUNT(*) FROM users GROUP BY email HAVING COUNT(*) > 1`

C. `SELECT email, COUNT(*) FROM users WHERE COUNT(*) > 1 GROUP BY email`

D. `SELECT email FROM users ORDER BY email`

2. MySQL rejects `DELETE FROM orders WHERE id IN (SELECT id FROM orders WHERE …)` with error 1093. What is the usual fix?

A. Add `LIMIT 1` to the subquery

B. Wrap the subquery in a derived table: `WHERE id IN (SELECT id FROM (SELECT … FROM orders …) AS t)`

C. Use `TRUNCATE` instead

D. Disable foreign-key checks

3. `SELECT DISTINCT email, name FROM contacts` is used to deduplicate contacts whose names were typed differently ('Asha', 'Asha K'). What does it return for that person?

A. One row, chosen at random

B. Two rows — DISTINCT removes only rows equal in every selected column

C. One row with the longer name

D. An error

4. Rows with a NULL `phone` are grouped with `GROUP BY phone`. How many groups do three NULL-phone rows form?

A. Three — NULL is never equal to NULL

B. One

C. None — NULLs are skipped

D. It depends on the collation

ANSWER KEY · DEDUPLICATION — FINDING, KEEPING AND DELETING DUPLICATE ROWS

1. B GROUP BY makes one row per email and HAVING filters groups after counting. An aggregate cannot appear in WHERE, which filters rows before grouping; DISTINCT lists every email, duplicated or not.

2. B MySQL will not read the table being modified in a subquery of the same statement, but it materialises a derived table first, which lifts the restriction. A multi-table `DELETE o1 FROM orders o1 JOIN orders o2 …` is the other common answer.

3. B DISTINCT cannot choose between rows that differ. To keep one row per key, number the rows with ROW_NUMBER over the key and a keeper rule, and keep number 1.

4. B GROUP BY, DISTINCT and PARTITION BY treat all NULLs as one group, even though `NULL = NULL` is not true. Whether those rows really are duplicates is a business question — usually they are not, so exclude them.

LESSON 3 · 15 MIN

Pivoting — turning rows into columns and back

A query result has a fixed set of columns and any number of rows. Reports often want the opposite shape: one row per region and a column per quarter, one row per student and a column per subject, one row per day and a column per status. Turning the values of one column into column *headers* is a **pivot**; turning columns back into rows is an **unpivot**. SQL Server and Oracle have a `PIVOT` keyword for it; MySQL does not, and does not need one, because conditional aggregation does the same job in plain SQL that runs everywhere.

The long shape and the wide shape

Data is normally stored *long*: one row per fact.

region	sale_date	amount
North	2024-01-15	200
North	2024-05-02	150
South	2024-02-20	300
South	2024-11-30	120

A report wants it *wide*:

region	q1	q2	q3	q4
North	200	150	0	0
South	300	0	0	120

Long is right for storage — a new quarter is a new row, not a schema change — and wide is right for reading. A pivot is the query that goes from one to the other.

Conditional aggregation

The pivot is a `GROUP BY` on the row label (region) with one aggregate per output column. Each aggregate looks at every row of the group but only *counts* the ones that belong in its column:

```sql
SELECT region,
       SUM(CASE WHEN QUARTER(sale_date) = 1 THEN amount ELSE 0 END) AS q1,
       SUM(CASE WHEN QUARTER(sale_date) = 2 THEN amount ELSE 0 END) AS q2,
       SUM(CASE WHEN QUARTER(sale_date) = 3 THEN amount ELSE 0 END) AS q3,
       SUM(CASE WHEN QUARTER(sale_date) = 4 THEN amount ELSE 0 END) AS q4,
       SUM(amount) AS total
FROM sales
GROUP BY region
ORDER BY region;
```

Read one column at a time: for `q1`, a row from January to March contributes its amount and every other row contributes 0. The `ELSE 0` matters. Without it, the CASE yields NULL for non-matching rows; `SUM` ignores NULLs, which is fine — until a region has *no* sale in a quarter, when every input is NULL and `SUM` returns NULL instead of 0. Either write `ELSE 0` or wrap the sum: `COALESCE(SUM(CASE … END), 0)`.

The `total` column is just another aggregate over the same group; it costs nothing extra. If you want a total *row* as well, MySQL has `GROUP BY region WITH ROLLUP`, which adds a row with `region` NULL holding the column totals. PostgreSQL and SQL Server write `GROUP BY ROLLUP (region)`. Alternatively, `UNION ALL` a second query without the `GROUP BY` — it works everywhere and lets you label the row `'Total'`.

The data does not have to be grouped by a column it holds directly — `QUARTER(sale_date)` is computed per row inside the CASE. That makes conditional aggregation a natural fit for date reports: by month (`MONTH(d) = 1`), by weekday, by hour.

Counting instead of summing

Many pivots count rows rather than add amounts — tickets per status per month:

```sql
SELECT DATE_FORMAT(opened_on, '%Y-%m') AS month,
       COUNT(CASE WHEN status = 'open'   THEN 1 END) AS open_count,
       COUNT(CASE WHEN status = 'closed' THEN 1 END) AS closed_count,
       COUNT(*) AS total
FROM tickets
GROUP BY DATE_FORMAT(opened_on, '%Y-%m')
ORDER BY month;
```

`COUNT(expr)` counts non-NULL values, and a CASE without ELSE is NULL when it does not match, so each `COUNT` counts only its rows. MySQL offers a shorter spelling: a comparison is `1` when true and `0` when false, so `SUM(status = 'open')` counts the open tickets. It is idiomatic MySQL and reads well, but it is not portable — PostgreSQL has a real boolean type and refuses to `SUM` it (there you write `COUNT(*) FILTER (WHERE status = 'open')`, PostgreSQL's own conditional-aggregate syntax).

Pivoting text

When the cell is a value rather than a number to add — each student's grade per subject — use `MAX` (or `MIN`) around a CASE with no ELSE:

```sql
SELECT student,
       MAX(CASE WHEN subject = 'maths'   THEN grade END) AS maths,
       MAX(CASE WHEN subject = 'physics' THEN grade END) AS physics
FROM grades
GROUP BY student
ORDER BY student;
```

There is one matching row per (student, subject), so `MAX` simply picks it out of the NULLs; a missing subject prints NULL, which is honest. If there could be several rows per cell, decide which one you want — the latest, the highest — before relying on `MAX`.

Unpivoting

The reverse direction appears when data arrives wide, as from a spreadsheet: one row per student with a column per subject. Normalising it back into rows is a stack of SELECTs, one per column:

```sql
SELECT student, 'maths' AS subject, maths AS score FROM scores
UNION ALL
SELECT student, 'physics', physics FROM scores
UNION ALL
SELECT student, 'chemistry', chemistry FROM scores
ORDER BY student, subject;
```

`UNION ALL`, not `UNION`: a student with the same score in two subjects must keep both rows, and `UNION` would also pay for a needless duplicate check. The column names come from the first SELECT. Add `WHERE maths IS NOT NULL` (and so on) to each part if an empty cell should not become a row. SQL Server and Oracle have an `UNPIVOT` operator; MySQL 8 can also unpivot with a `CROSS JOIN` to a derived table of the column names plus a CASE, but `UNION ALL` is the version everyone reads instantly.

When the columns are not known in advance

A SELECT's output columns are fixed when the query is written. A pivot over "every status that exists" — values you do not know in advance — cannot be written as one static query in any SQL dialect. The MySQL workaround builds the query text itself:

```sql
SELECT GROUP_CONCAT(DISTINCT
         CONCAT('SUM(status = ''', status, ''') AS `', status, '`'))
INTO @cols FROM tickets;
SET @sql = CONCAT('SELECT month, ', @cols, ' FROM … GROUP BY month');
PREPARE stmt FROM @sql; EXECUTE stmt; DEALLOCATE PREPARE stmt;
```

That is a stored-procedure technique, worth recognising in an interview; in practice most teams return the long, grouped rows and let the application or the reporting tool pivot them. PostgreSQL's `crosstab` function (from the `tablefunc` extension) has the same limitation — it still needs the output column list.

How to say it in an interview

"MySQL has no PIVOT, so I group by the row label and write one conditional aggregate per column — SUM of a CASE for amounts, COUNT of a CASE for counts, MAX of a CASE for text — with ELSE 0 so empty cells print zero." For the reverse, "UNION ALL one SELECT per column." If

asked about unknown columns, say the column list must be known when the query is written, and describe the prepared-statement workaround.

EXERCISE 1

Quarterly sales by region

The `sales` table holds one row per sale.

id	region	sale_date	amount
1	North	2024-01-15	200
2	North	2024-03-10	100
3	North	2024-05-02	150
4	South	2024-02-20	300
5	South	2024-11-30	120
6	East	2024-07-04	80
7	East	2024-08-19	90
8	East	2024-12-01	60

Pivot it into **one row per region** with the columns `region`, `q1`, `q2`, `q3`, `q4` (the sum of that quarter's amounts, **0** when there were none) and `total`, ordered by `region`. All sales are in one year.

```
region | q1 | q2 | q3 | q4 | total
East | 0 | 0 | 170 | 60 | 230
North | 300 | 150 | 0 | 0 | 450
South | 300 | 0 | 0 | 120 | 420
```

INPUT 1

```
CREATE TABLE sales (id INT PRIMARY KEY,
region VARCHAR(10), sale_date DATE,
amount INT);
INSERT INTO sales VALUES
  (1, 'North', '2024-01-15', 200),
  (2, 'North', '2024-03-10', 100),
  (3, 'North', '2024-05-02', 150),
  (4, 'South', '2024-02-20', 300),
  (5, 'South', '2024-11-30', 120),
  (6, 'East',  '2024-07-04',  80),
  (7, 'East',  '2024-08-19',  90),
  (8, 'East',  '2024-12-01',  60);
```

OUTPUT 1

```
region | q1 | q2 | q3 | q4 | total
East | 0 | 0 | 170 | 60 | 230
North | 300 | 150 | 0 | 0 | 450
South | 300 | 0 | 0 | 120 | 420
```

REFERENCE SOLUTION

```sql
SELECT region,
       SUM(CASE WHEN QUARTER(sale_date) = 1 THEN amount ELSE 0 END) AS q1,
       SUM(CASE WHEN QUARTER(sale_date) = 2 THEN amount ELSE 0 END) AS q2,
       SUM(CASE WHEN QUARTER(sale_date) = 3 THEN amount ELSE 0 END) AS q3,
       SUM(CASE WHEN QUARTER(sale_date) = 4 THEN amount ELSE 0 END) AS q4,
       SUM(amount) AS total
FROM sales
GROUP BY region
ORDER BY region;
```

EXERCISE 2

Unpivot the exam scores

Exam scores arrived from a spreadsheet, one column per subject; a NULL means the student did not sit that exam.

student	maths	physics	chemistry
Arjun	78	NULL	65
Bela	91	88	NULL
Chen	70	70	70

Turn it into **one row per student and subject sat**: the columns `student`, `subject` (the text `maths`, `physics` or `chemistry`) and `score`, leaving out the NULL cells, ordered by `student`, then `subject`.

```
student | subject | score
Arjun | chemistry | 65
Arjun | maths | 78
Bela | maths | 91
Bela | physics | 88
Chen | chemistry | 70
Chen | maths | 70
Chen | physics | 70
```

INPUT 1

```
CREATE TABLE scores (student VARCHAR(20)
PRIMARY KEY, maths INT, physics INT,
chemistry INT);
INSERT INTO scores VALUES
  ('Arjun', 78, NULL, 65),
  ('Bela',  91, 88,   NULL),
  ('Chen',  70, 70,   70);
```

OUTPUT 1

```
student | subject | score
Arjun | chemistry | 65
Arjun | maths | 78
Bela | maths | 91
Bela | physics | 88
Chen | chemistry | 70
Chen | maths | 70
Chen | physics | 70
```

REFERENCE SOLUTION

```sql
SELECT student, 'maths' AS subject, maths AS score FROM scores WHERE maths IS NOT NU
UNION ALL
SELECT student, 'physics', physics FROM scores WHERE physics IS NOT NULL
UNION ALL
SELECT student, 'chemistry', chemistry FROM scores WHERE chemistry IS NOT NULL
ORDER BY student, subject;
```

QUIZ

1. Region West has no sale in Q2. What does `SUM(CASE WHEN QUARTER(sale_date) = 2 THEN amount END) AS q2` print for West?

   A. 0

   B. NULL

   C. An error

   D. The total of all West's sales

2. Which aggregate is right for pivoting a text value, such as each student's grade per subject?

A. `SUM(CASE WHEN subject = 'maths' THEN grade END)`

B. `MAX(CASE WHEN subject = 'maths' THEN grade END)`

C. `COUNT(CASE WHEN subject = 'maths' THEN grade END)`

D. `AVG(CASE WHEN subject = 'maths' THEN grade END)`

3. Why `UNION ALL` rather than `UNION` when unpivoting columns into rows?

A. UNION is not supported in MySQL

B. UNION would remove rows that happen to be identical — a student with the same score in two subjects could lose one — and it pays for a duplicate check

C. UNION ALL sorts the result

D. There is no difference

4. A report needs one column per status, but new statuses are added every month. Which statement is true?

A. `SUM(CASE …)` adds the new columns automatically

B. A static SELECT's columns are fixed when it is written; a dynamic pivot builds the SQL text (for example with GROUP_CONCAT) and runs it as a prepared statement, or the application pivots the rows

C. MySQL's PIVOT operator handles it

D. `GROUP BY WITH ROLLUP` creates the columns

ANSWER KEY · PIVOTING — TURNING ROWS INTO COLUMNS AND BACK

1. B Without ELSE the CASE yields NULL for every West row, and SUM of only NULLs is NULL. Add `ELSE 0`, or wrap the sum in `COALESCE(…, 0)`, to print 0.

2. B With one matching row per cell, MAX (or MIN) simply picks that value out of the NULLs. SUM and AVG need numbers, and COUNT returns how many rows matched, not the value.

3. B Each unpivoted row is a distinct fact even when its values coincide with another's. In this exercise the subject label differs, so UNION would not drop rows here, but it would still sort or hash every row for nothing.

4. B No dialect can return columns whose names are not known when the query is written. MySQL has no PIVOT operator; the workaround generates the column list as text.

LESSON 4 · 16 MIN

Rolling windows — moving averages, sliding sums and period-over-period

A daily figure is noisy: sales spike on a Saturday, a holiday empties a Wednesday. A **rolling window** smooths it by summarising, for each row, the rows around it — the last seven days, the last three readings. The window-function module introduced frames; this lesson is about using them for the questions reports actually ask, and about the trap that catches most first attempts: a frame counts *rows*, and dates are not rows.

The basic moving average

Given one row per day, a three-day moving average is an `AVG` over a frame of the current row and the two before it:

```
SELECT day, signups,
       ROUND(AVG(signups) OVER (
         ORDER BY day
         ROWS BETWEEN 2 PRECEDING AND CURRENT ROW), 2) AS avg_3d
FROM daily_signups
ORDER BY day;
```

The frame is evaluated for every row: on the 5th it covers the 3rd, 4th and 5th; on the 6th, the 4th, 5th and 6th. `ROWS 2 PRECEDING` is shorthand for the same frame. `SUM` in place of `AVG` gives a rolling total; `MAX` a rolling peak. The `ORDER BY` inside `OVER` decides what "preceding" means and is independent of the query's final `ORDER BY` — write both.

Add `PARTITION BY` when there are several series, so one store's window never borrows rows from the next store:

```
SUM(amount) OVER (PARTITION BY store_id ORDER BY day
                  ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
```

Frames can also look forward (`BETWEEN 1 PRECEDING AND 1 FOLLOWING` is a centred three-point average) and can be unbounded (`UNBOUNDED PRECEDING AND CURRENT ROW` is the running total from the earlier module).

The incomplete first window

On the first day, the frame has only one row; on the second, two. The average is computed anyway, over the rows that exist, so the first values of a "3-day average" are really 1- and 2-day averages. A report should usually say so by printing NULL until the window is full. Count the rows in the same frame and test it:

```
SELECT day, signups,
       CASE WHEN COUNT(*) OVER w = 3
            THEN ROUND(AVG(signups) OVER w, 2) END AS avg_3d
FROM daily_signups
WINDOW w AS (ORDER BY day ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
ORDER BY day;
```

The `WINDOW` clause names a window once so two functions can share it — MySQL 8, PostgreSQL and SQLite all support it; SQL Server added it only in 2022. Writing the `OVER (…)` out twice is equally correct.

Rows are not days

Now the trap. Suppose a shop has no payments on some days, and several payments on others:

paid_on	amount
2024-05-01	100
2024-05-01	50
2024-05-03	70
2024-05-09	30

`SUM(amount) OVER (ORDER BY paid_on ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)` is not a seven-day sum here. On the 9th it adds the 9th's row and up to six rows before it — reaching back to the 1st, eight days earlier — and on the 1st the two payments get *different* sums, because each is a separate row. "Seven rows" equals "seven days" only when there is exactly one row per day. There are three fixes, and an interviewer will be pleased to hear all three named.

First, aggregate to one row per day. Always the first step for daily reports: `GROUP BY paid_on` with `SUM(amount)`. That solves the several-rows-per-day half.

Second, fill the missing days. Join the daily totals onto a calendar of every date (the date-series lesson builds one with a recursive CTE), with `COALESCE(total, 0)` for the empty days. Now there is exactly one row per day and the `ROWS` frame is a true seven-day window.

Third, frame on the date instead of the row. MySQL 8 supports a `RANGE` frame with an interval when the window is ordered by a date:

```
SUM(day_total) OVER (
  ORDER BY paid_on
  RANGE BETWEEN INTERVAL 6 DAY PRECEDING AND CURRENT ROW)
```

`RANGE` includes every row whose `paid_on` is within six days before the current one, however many rows that is, so gaps no longer matter. PostgreSQL writes the interval as `INTERVAL '6 days'`; SQL Server supports `RANGE` only with `UNBOUNDED` or `CURRENT ROW` bounds, so there you fill the calendar or use the subquery below. (The exercises on this site run on an engine that does not take the interval form, so they use the other approaches.)

The portable fallback is a correlated subquery or self-join that states the window as a date condition:

```
WITH daily AS (
  SELECT paid_on, SUM(amount) AS day_total
  FROM payments GROUP BY paid_on
)
SELECT d.paid_on, d.day_total,
       (SELECT SUM(x.day_total) FROM daily AS x
        WHERE x.paid_on BETWEEN DATE_SUB(d.paid_on, INTERVAL 6 DAY) AND d.paid
         AS week_total
FROM daily AS d
ORDER BY d.paid_on;
```

It reads naturally and runs everywhere. Its cost is one lookup per output row, which an index on the date makes cheap; the window version reads the data once.

ROWS versus RANGE with ties

`RANGE` matters even without dates. With `ORDER BY score` and no explicit frame, the default frame is `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, and under `RANGE` all rows with an equal `score` are *peers* that enter the frame together. A running `SUM(…) OVER (ORDER BY score)` therefore jumps by the whole tie at once and prints the same total on every tied row. If you want a strictly row-by-row running total, either say `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` or make the order unique by adding a tie-breaker column. This default surprises many people, which is why interviewers ask about it.

Period over period

"How did this week compare with last week?" is a rolling comparison of two buckets, not a frame. Aggregate to one row per period, then reach back one row with `LAG`:

```
WITH weekly AS (
  SELECT YEARWEEK(order_date, 3) AS yw, SUM(amount) AS total
  FROM orders GROUP BY YEARWEEK(order_date, 3)
)
SELECT yw, total,
       total - LAG(total) OVER (ORDER BY yw) AS change_abs,
       ROUND(100 * (total - LAG(total) OVER (ORDER BY yw))
             / NULLIF(LAG(total) OVER (ORDER BY yw), 0), 1) AS change_pct
FROM weekly
ORDER BY yw;
```

The first week has no previous week, so both columns are NULL. `NULLIF(prev, 0)` turns a zero denominator into NULL, so a week after an empty week gives NULL rather than a division error (MySQL would return NULL for division by zero anyway, with a warning; PostgreSQL and SQL Server raise an error). The same shape answers month-over-month and year-over-year, with `LAG(total, 12)` over monthly rows for "same month last year" — provided no month is missing, which is the calendar problem again.

How to say it in an interview

"A moving average is AVG over a ROWS frame — but a ROWS frame counts rows, so first I aggregate to one row per day and fill missing days from a calendar, or I use a RANGE frame on the date." Mention the incomplete first window and how you would report it, and use `LAG`

with `NULLIF` for period-over-period change. Those three sentences cover nearly every rolling-window question.

EXERCISE 1

Three-day moving average of signups

`daily_signups` has exactly one row per day, with no days missing.

day	signups
2024-06-01	10
2024-06-02	14
2024-06-03	9
2024-06-04	20
2024-06-05	17
2024-06-06	11

For each day, return `day`, `signups` and `avg_3d`: the average of that day and the two days before it, rounded to 2 decimal places — and **NULL** on any day that does not yet have two days before it. Order by `day`.

```
day | signups | avg_3d
2024-06-01 | 10 | NULL
2024-06-02 | 14 | NULL
2024-06-03 | 9 | 11
2024-06-04 | 20 | 14.33
2024-06-05 | 17 | 15.33
2024-06-06 | 11 | 16
```

INPUT 1

```
CREATE TABLE daily_signups (day DATE
PRIMARY KEY, signups INT);
INSERT INTO daily_signups VALUES
  ('2024-06-01', 10), ('2024-06-02',
14), ('2024-06-03', 9),
  ('2024-06-04', 20), ('2024-06-05',
17), ('2024-06-06', 11);
```

OUTPUT 1

```
day | signups | avg_3d
2024-06-01 | 10 | NULL
2024-06-02 | 14 | NULL
2024-06-03 | 9 | 11
2024-06-04 | 20 | 14.33
2024-06-05 | 17 | 15.33
2024-06-06 | 11 | 16
```

REFERENCE SOLUTION

```sql
SELECT day, signups,
       CASE WHEN COUNT(*) OVER (ORDER BY day ROWS BETWEEN 2 PRECEDING AND CURRENT RO
            THEN ROUND(AVG(signups) OVER (ORDER BY day ROWS BETWEEN 2 PRECEDING AND
       END AS avg_3d
FROM daily_signups
ORDER BY day;
```

EXERCISE 2

Seven calendar days of revenue

`payments` can have several rows on one day and none on others.

id	paid_on	amount
1	2024-05-01	100
2	2024-05-01	50
3	2024-05-03	70
4	2024-05-07	20
5	2024-05-08	40
6	2024-05-09	30
7	2024-05-15	10

The starter tries a seven-day total with `ROWS BETWEEN 6 PRECEDING`, which counts rows, not days. Fix it. For **each date that has payments**, return `paid_on`, `day_total` (that day's sum) and `week_total` — the sum of all payments from six days before that date up to and including it. Order by `paid_on`.

```
paid_on | day_total | week_total
2024-05-01 | 150 | 150
2024-05-03 | 70 | 220
2024-05-07 | 20 | 240
2024-05-08 | 40 | 130
2024-05-09 | 30 | 160
2024-05-15 | 10 | 40
```

INPUT 1

```
CREATE TABLE payments (id INT PRIMARY
KEY, paid_on DATE, amount INT);
INSERT INTO payments VALUES
  (1, '2024-05-01', 100), (2, '2024-05-
01', 50), (3, '2024-05-03', 70),
  (4, '2024-05-07', 20),  (5, '2024-05-
08', 40), (6, '2024-05-09', 30),
  (7, '2024-05-15', 10);
```

OUTPUT 1

```
paid_on | day_total | week_total
2024-05-01 | 150 | 150
2024-05-03 | 70 | 220
2024-05-07 | 20 | 240
2024-05-08 | 40 | 130
2024-05-09 | 30 | 160
2024-05-15 | 10 | 40
```

REFERENCE SOLUTION

```
WITH daily AS (
  SELECT paid_on, SUM(amount) AS day_total
  FROM payments
  GROUP BY paid_on
)
SELECT d.paid_on, d.day_total,
       (SELECT SUM(x.day_total)
        FROM daily AS x
        WHERE x.paid_on BETWEEN DATE_SUB(d.paid_on, INTERVAL 6 DAY) AND d.paid_on) A
FROM daily AS d
ORDER BY d.paid_on;
```

QUIZ

1. What does `ROWS BETWEEN 6 PRECEDING AND CURRENT ROW` cover on a table with one row per *transaction*?

A. The last seven calendar days

B. The current row and the six rows before it, however many days they span

C. The last seven distinct dates

D. The whole partition

2. With `SUM(points) OVER (ORDER BY score)` and no frame clause, two rows share the same `score`. What do they show?

A. Different running totals, one row at a time

B. The same total, because the default frame is RANGE … CURRENT ROW and tied rows are peers that enter together

C. NULL

D. An error about non-unique ordering

3. Which MySQL 8 frame is a true seven-day window over a date column with gaps?

A. `ROWS BETWEEN 7 PRECEDING AND CURRENT ROW`

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

C. `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`

D. `RANGE BETWEEN 6 PRECEDING AND 6 FOLLOWING`

4. Why write `NULLIF(LAG(total) OVER (ORDER BY week), 0)` in a week-over-week percentage?

A. To skip the first week

B. So a previous week of zero gives NULL rather than a division error (or MySQL's NULL with a warning)

C. To round the result

D. It makes LAG look two rows back

ANSWER KEY · ROLLING WINDOWS — MOVING AVERAGES, SLIDING SUMS AND PERIOD-OVER-PERIOD

1. B A ROWS frame counts physical rows. It equals seven days only when there is exactly one row per day — aggregate per day and fill missing days, or use a RANGE frame on the date.

2. B The default frame with an ORDER BY is `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Use `ROWS` or a unique tie-breaker for a strictly row-by-row total.

3. B A RANGE frame with an interval includes every row whose date lies within six days before the current one, whatever the number of rows. Today plus six days back is seven days.

4. B NULLIF(x, 0) returns NULL when x is 0, and dividing by NULL gives NULL. PostgreSQL and SQL Server raise an error on division by zero, so the guard is what makes the query portable.

LESSON 5 · 15 MIN

The nth highest value — LIMIT OFFSET, DENSE_RANK and the NULL answer

"Find the second highest salary" may be the most asked SQL interview question there is, and its general form — the nth highest value, overall or per group — is a good test of three things at once: sorting with `LIMIT` / `OFFSET`, ranking functions, and what a query returns when the answer does not exist. That last part is where most answers fail. This lesson shows four ways to write it, which one to reach for, and how to make it return `NULL` correctly.

Pin down the question

Before writing, settle two points, because the sample data in an interview is usually chosen to test them:

1. **Distinct values or rows?** With salaries 300, 300, 200, is the second highest 300 (the second row) or 200 (the second distinct value)? Almost always the second *distinct* value — 200. Say that you are assuming it.
2. **What if there is none?** With one employee, or everyone on the same salary, there is no second highest. The usual requirement is a single row holding `NULL`, not an empty result.

Way 1: LIMIT and OFFSET

Sort the distinct values descending and skip `n - 1` of them:

```sql
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;          -- second highest; MySQL also accepts LIMIT 1, 1
```

This returns the right value when it exists and *no rows* when it does not. To turn "no rows" into a row with `NULL`, make it a scalar subquery:

```sql
SELECT (
  SELECT DISTINCT salary
  FROM employees
  ORDER BY salary DESC
  LIMIT 1 OFFSET 1
) AS second_highest;
```

A scalar subquery that finds nothing evaluates to `NULL`, and the outer `SELECT` without a `FROM` always produces exactly one row. This wrapper is the detail interviewers look for.

`OFFSET` must be a literal in MySQL — `LIMIT 1 OFFSET N - 1` with a variable is not allowed in a plain query. Inside a stored function you compute `SET n = n - 1;` first and then write `LIMIT 1 OFFSET n`, which MySQL accepts for routine variables. SQL Server spells the clause `ORDER BY salary DESC OFFSET 1 ROWS FETCH NEXT 1 ROWS ONLY`; PostgreSQL accepts `LIMIT 1 OFFSET 1` like MySQL.

Way 2: DENSE_RANK

Rank the salaries, then keep rank `n`:

```sql
SELECT MAX(salary) AS third_highest
FROM (
  SELECT salary,
         DENSE_RANK() OVER (ORDER BY salary DESC) AS rk
  FROM employees
) AS ranked
WHERE rk = 3;
```

`DENSE_RANK` gives equal salaries the same rank and leaves no gaps: 300, 300, 200, 100 are ranked 1, 1, 2, 3. `RANK` would give 1, 1, 3, 4 — there would be no rank 2 at all, and asking for the second highest would wrongly return nothing. `ROW_NUMBER` would number the two 300s as 1 and 2, and "rank 2" would be 300, the row answer rather than the distinct one. The comparison of the three ranking functions on tied data is a theory question in its own right; this query is where it matters.

The outer `MAX` does two jobs: it collapses the tied rows of rank 3 into one value, and, like every aggregate without `GROUP BY`, it returns one row with `NULL` when no row has rank 3. So this version needs no scalar-subquery wrapper.

Way 3: a correlated count

Before window functions, the nth highest value was "the salary that exactly `n - 1` distinct salaries are greater than":

```sql
SELECT DISTINCT e.salary
FROM employees AS e
WHERE 2 = (                  -- n - 1 = 2, so the third highest
  SELECT COUNT(DISTINCT x.salary)
  FROM employees AS x
  WHERE x.salary > e.salary
);
```

It is correct and portable, and it is worth recognising, but the subquery runs once per row — quadratic on a large table without an index. Mention it as the classic alternative and prefer one of the others.

Way 4: MAX below MAX

For the second highest only, the shortest correct answer is:

```sql
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

The inner query finds the top; the outer finds the largest value strictly below it — automatically distinct, and `NULL` when nothing is below. It does not generalise well (the third highest needs another nesting level), but for "second highest" it is the answer many interviewers expect first.

Per group

"The second highest salary in each department" partitions the ranking:

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

`GROUP BY dept, salary` removes repeats when two people share the second salary (a `SELECT DISTINCT` does the same). A department with a single distinct salary simply does not appear. If the report must list *every* department, with `NULL` where there is no second salary, start from the departments and outer-join:

```sql
SELECT d.dept, r.salary AS second_salary
FROM (SELECT DISTINCT dept FROM employees) AS d
LEFT JOIN (
  SELECT DISTINCT dept, salary
  FROM (SELECT dept, salary,
               DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS r
        FROM employees) AS x
  WHERE rk = 2
) AS r ON r.dept = d.dept
ORDER BY d.dept;
```

This "start from the full list of groups, then LEFT JOIN the answers" shape is the per-group version of the scalar-subquery wrapper: it guarantees one row per group whether or not an answer exists. A correlated scalar subquery per department — `(SELECT DISTINCT salary … WHERE dept = d.dept ORDER BY salary DESC LIMIT 1 OFFSET 1)` — does the same in MySQL and PostgreSQL.

NULL salaries

`ORDER BY salary DESC` puts `NULL` last in MySQL (NULL sorts as the smallest value), and `DENSE_RANK` ranks NULLs too — last in descending order. If the column can be NULL, decide whether an unknown salary is a value at all; usually it is not, and a `WHERE salary IS NOT NULL` keeps it from becoming "the fourth highest". `MAX` and `COUNT(DISTINCT …)` ignore NULLs by themselves.

How to say it in an interview

"I'll take the nth highest *distinct* value, and return NULL if there isn't one." Then write `DENSE_RANK() … WHERE rk = n` under a `MAX`, or the `LIMIT 1 OFFSET n-1` inside a scalar subquery, and explain why `RANK` and `ROW_NUMBER` give the wrong answer on ties. For the per-group version, partition the rank, and LEFT JOIN from the list of groups if every group must appear.

EXERCISE 1

The third-highest distinct price

Find the **third-highest distinct price** in `products`. Return one row with one column, `third_price` — and `NULL` when there are fewer than three distinct prices (including when the table is empty).

id	name	price
1	Desk	500
2	Chair	750
3	Sofa	750
4	Lamp	300
5	Rug	200
6	Shelf	300

The distinct prices from the top are 750, 500, 300:

```
third_price
300
```

INPUT 1

```
CREATE TABLE products (id INT PRIMARY
KEY, name VARCHAR(30), price INT);
INSERT INTO products VALUES
  (1, 'Desk', 500), (2, 'Chair', 750),
(3, 'Sofa', 750),
  (4, 'Lamp', 300), (5, 'Rug', 200), (6,
'Shelf', 300);
```

OUTPUT 1

```
third_price
300
```

REFERENCE SOLUTION

```sql
SELECT (
  SELECT DISTINCT price
  FROM products
  ORDER BY price DESC
  LIMIT 1 OFFSET 2
) AS third_price;
```

EXERCISE 2

Second-best score per subject

`results` holds exam scores (never NULL).

student	subject	score
Ira	maths	90
Jai	maths	85
Kim	maths	90
Lev	maths	70
Ira	physics	60
Jai	physics	60
Ira	chemistry	88
Kim	chemistry	92
Lev	biology	75

For **every subject**, return `subject` and `second_score`: the second-highest *distinct* score in that subject, or `NULL` when the subject has only one distinct score. Order by `subject`.

```
subject | second_score
biology | NULL
chemistry | 88
maths | 85
physics | NULL
```

INPUT 1

```
CREATE TABLE results (student
VARCHAR(20), subject VARCHAR(20), score
INT);
INSERT INTO results VALUES
  ('Ira', 'maths', 90), ('Jai', 'maths',
85), ('Kim', 'maths', 90), ('Lev',
'maths', 70),
  ('Ira', 'physics', 60), ('Jai',
'physics', 60),
  ('Ira', 'chemistry', 88), ('Kim',
'chemistry', 92),
  ('Lev', 'biology', 75);
```

OUTPUT 1

```
subject | second_score
biology | NULL
chemistry | 88
maths | 85
physics | NULL
```

REFERENCE SOLUTION

```sql
WITH ranked AS (
  SELECT subject, score,
         DENSE_RANK() OVER (PARTITION BY subject ORDER BY score DESC) AS rk
  FROM results
)
SELECT subject,
       MAX(CASE WHEN rk = 2 THEN score END) AS second_score
FROM ranked
GROUP BY subject
ORDER BY subject;
```

QUIZ

1. Salaries are 300, 300, 200, 100. Which function makes `WHERE rk = 2` return 200?

A. `ROW_NUMBER()`

B. `RANK()`

C. `DENSE_RANK()`

D. Any of the three

2. There is only one employee. What does `SELECT DISTINCT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 1` return?

A. One row with NULL

B. No rows

C. The only salary

D. An error

3. Why does `SELECT MAX(salary) FROM employees WHERE salary < (SELECT MAX(salary) FROM employees)` need no NULL wrapper?

A. Because MAX skips NULL salaries

B. Because an aggregate without GROUP BY always returns exactly one row, NULL when no row qualifies

C. Because the subquery returns NULL

D. It does need one

4. A report must list every department with its second-highest salary, NULL where there is none. Which shape guarantees one row per department?

A. `… WHERE rk = 2` on the ranked rows

B. The distinct departments LEFT JOINed to the rank-2 rows, or a GROUP BY dept with `MAX(CASE WHEN rk = 2 …)`

C. `INNER JOIN` the departments to the rank-2 rows

D. `LIMIT 1 OFFSET 1` per department in the main query

ANSWER KEY · THE NTH HIGHEST VALUE — LIMIT OFFSET, DENSE_RANK AND THE NULL ANSWER

1. C DENSE_RANK gives 1, 1, 2, 3. RANK gives 1, 1, 3, 4 — no row has rank 2. ROW_NUMBER gives 1, 2, 3, 4, so rank 2 is the second 300.

2. B OFFSET 1 skips the only row, leaving nothing. Wrapping the query in a scalar subquery, `SELECT (…) AS second`, turns that empty result into one row holding NULL.

3. B MAX over an empty set is NULL, and an aggregate query with no GROUP BY returns one row even when no rows pass the WHERE.

4. B Filtering on rank 2 drops departments that have no second value. Starting from every department (or grouping all ranked rows by department) keeps them, with NULL in place of the missing value.

LESSON 6 · 15 MIN

Date series — calendars from recursive CTEs and filling missing days

A report built from `GROUP BY day` has a quiet flaw: days with no data are not *zero*, they are *missing*. Ask for orders per day and a Sunday with no orders simply is not in the result. A chart then draws a straight line across it, a moving average treats the next row as the next day, and a manager reading the table never learns that Sunday was empty. Every serious daily, weekly or monthly report starts from a **date series** — one row per period, whether or not anything happened — and joins the data onto it. MySQL has no built-in series function, so this lesson builds one with a recursive CTE and shows the patterns that follow from it.

A calendar from a recursive CTE

The recursive-CTE lesson counted from 1 to 10. A calendar is the same shape with dates:

```
WITH RECURSIVE calendar AS (
  SELECT DATE('2024-03-01') AS day          -- anchor: the first day
  UNION ALL
  SELECT DATE_ADD(day, INTERVAL 1 DAY)      -- recursive step: the next day
  FROM calendar
  WHERE day < '2024-03-07'                  -- stop condition
)
SELECT day FROM calendar ORDER BY day;
```

The anchor produces one row; each recursion takes the rows produced last time and adds one day, as long as the condition holds; the result is seven rows, the 1st to the 7th. The `WHERE` is checked against the *previous* row, so the last day produced is the first one that is not below the bound — here exactly `'2024-03-07'`.

Two MySQL specifics matter. First, the column's type comes from the anchor: `DATE('2024-03-01')` makes it a date. A bare string literal would make the column a short string type, and in strict mode the recursive part's longer values could be cut or rejected — the classic "Data too long for column" error in recursive CTEs. Cast the anchor (`DATE(…)` or `CAST(… AS DATE)`) to be safe. Second, MySQL stops any recursive CTE after **1,000 iterations** by default (`cte_max_recursion_depth`) and raises an error. Three years of days needs the variable

raised for the session (`SET SESSION cte_max_recursion_depth = 2000;`) or a different source of rows.

The bounds rarely need to be typed in. Take them from the data:

```
WITH RECURSIVE bounds AS (
  SELECT MIN(ordered_on) AS first_day, MAX(ordered_on) AS last_day FROM orders
),
calendar AS (
  SELECT first_day AS day FROM bounds
  UNION ALL
  SELECT DATE_ADD(c.day, INTERVAL 1 DAY)
  FROM calendar AS c
  JOIN bounds AS b ON c.day < b.last_day
)
SELECT day FROM calendar;
```

`WITH RECURSIVE` is written once at the top, and it applies to the whole list — the first CTE does not have to be the recursive one.

Filling the missing days

With a calendar, the report is a `LEFT JOIN` from the calendar to the data, grouped by the *calendar's* day:

```
WITH RECURSIVE calendar AS ( /* as above */ )
SELECT c.day,
       COUNT(o.id)                AS orders,
       COALESCE(SUM(o.amount), 0) AS revenue
FROM calendar AS c
LEFT JOIN orders AS o ON o.ordered_on = c.day
GROUP BY c.day
ORDER BY c.day;
```

Every detail here is load-bearing.

- The calendar is on the **left**. Every day survives; days with orders bring their rows along, days without bring one row of NULLs.
- **`COUNT(o.id)`, not `COUNT(*)`.** On an empty day the join still produces one row — the calendar row with NULLs on the order side. `COUNT(*)` counts that row and says 1;

`COUNT(o.id)` counts only real orders and says 0. This is the most common bug in gap-filling queries, and a standard interview follow-up.

- **`COALESCE(SUM(…), 0)`.** `SUM` of no values is NULL, not 0.
- **Group by `c.day`.** `o.ordered_on` is NULL on empty days, so grouping by it would merge every empty day into one NULL group.
- If `ordered_on` were a `DATETIME`, the join would need `o.ordered_on ⩾ c.day AND o.ordered_on < c.day + INTERVAL 1 DAY` — comparing a datetime to a date matches only midnight.

Any filter on the orders — only paid ones, only one store — belongs in the `ON` clause. Put it in `WHERE` and the NULL rows of empty days fail it, turning the left join back into an inner join and losing exactly the days you set out to keep.

Months, weeks and other steps

The step can be any interval. A month series starts on the first of a month and adds `INTERVAL 1 MONTH`:

```
WITH RECURSIVE months AS (
  SELECT DATE('2024-01-01') AS month_start
  UNION ALL
  SELECT DATE_ADD(month_start, INTERVAL 1 MONTH)
  FROM months WHERE month_start < '2024-06-01'
)
SELECT DATE_FORMAT(m.month_start, '%Y-%m') AS month,
       COUNT(s.user_id) AS signups
FROM months AS m
LEFT JOIN signups AS s
  ON s.signed_up ⩾ m.month_start
 AND s.signed_up <  DATE_ADD(m.month_start, INTERVAL 1 MONTH)
GROUP BY m.month_start
ORDER BY m.month_start;
```

Start month series on the 1st: adding a month to January 31st gives February 29th (in 2024), and the series then drifts to the 29th of every later month. Joining on a half-open range (`⩾` start, `<` next start) rather than `DATE_FORMAT(s.signed_up, '%Y-%m') = …` also keeps the join sargable — an index on `signed_up` can be used.

A plain number series (`SELECT 1 AS n UNION ALL SELECT n + 1 … WHERE n < 100`) is useful in its own right: generating slots, splitting a string by position, expanding a quantity into that

many rows.

Cumulative totals over the filled series

Once every day is present, running totals and rolling windows behave. A window function over the grouped result runs after the `GROUP BY`:

```
SELECT c.day,
       COUNT(o.id) AS orders,
       SUM(COUNT(o.id)) OVER (ORDER BY c.day) AS orders_to_date
FROM calendar AS c
LEFT JOIN orders AS o ON o.ordered_on = c.day
GROUP BY c.day
ORDER BY c.day;
```

`SUM(COUNT(o.id)) OVER (…)` looks strange the first time: the inner `COUNT` is the group's aggregate, and the outer `SUM … OVER` is a window across the grouped rows. On an empty day the cumulative total carries forward unchanged, as it should, and a `ROWS BETWEEN 6 PRECEDING AND CURRENT ROW` frame over this result is now a true seven-day window — the fix the rolling-windows lesson promised.

Calendar tables

Generating the series in every query is fine for reports over weeks or months. Teams that report by date constantly keep a permanent **calendar table** instead: one row per day for decades, with precomputed columns — year, quarter, month name, ISO week, weekday, `is_weekend`, `is_holiday`, fiscal period. It is filled once (by a recursive CTE, a script, or a spreadsheet import), is tiny by database standards (about 3,650 rows per decade), and turns questions like "working days between two dates" into a simple `COUNT(*) … WHERE NOT is_weekend AND NOT is_holiday`. Holidays cannot be computed by any function; a table is the only place to keep them.

Other databases

PostgreSQL has `generate_series('2024-03-01'::date, '2024-03-07', interval '1 day')`, which replaces the recursive CTE entirely. SQL Server uses a recursive CTE like MySQL

but stops at 100 levels by default; add `OPTION (MAXRECURSION 0)` (no limit) or a larger number to the statement. Oracle traditionally uses `CONNECT BY LEVEL ⩽ n`. The LEFT JOIN, `COUNT(column)` and `COALESCE` parts are identical everywhere.

How to say it in an interview

"Grouping by date drops days with no data, so I generate the dates with a recursive CTE — or use a calendar table — and LEFT JOIN the data onto it, counting a column from the right side and coalescing sums to zero." Mention the recursion limit and the filter-in-`ON` rule, and you have covered the follow-ups before they are asked.

EXERCISE 1

Daily orders with the empty days

`orders` has no rows on days nobody ordered.

id	ordered_on	amount
1	2024-03-01	100
2	2024-03-01	50
3	2024-03-03	70
4	2024-03-06	30

Report **every day** from the first order date to the last one, inclusive: the columns `day`, `orders` (the number of orders, 0 on an empty day) and `revenue` (the sum of amounts, 0 on an empty day), ordered by `day`. Take the first and last dates from the table — do not type them in.

```
day | orders | revenue
2024-03-01 | 2 | 150
2024-03-02 | 0 | 0
2024-03-03 | 1 | 70
2024-03-04 | 0 | 0
2024-03-05 | 0 | 0
2024-03-06 | 1 | 30
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
ordered_on DATE, amount INT);
INSERT INTO orders VALUES
  (1, '2024-03-01', 100), (2, '2024-03-
01', 50),
  (3, '2024-03-03', 70),  (4, '2024-03-
06', 30);
```

OUTPUT 1

```
day | orders | revenue
2024-03-01 | 2 | 150
2024-03-02 | 0 | 0
2024-03-03 | 1 | 70
2024-03-04 | 0 | 0
2024-03-05 | 0 | 0
2024-03-06 | 1 | 30
```

REFERENCE SOLUTION

```sql
WITH RECURSIVE bounds AS (
  SELECT MIN(ordered_on) AS first_day, MAX(ordered_on) AS last_day FROM orders
),
calendar AS (
  SELECT first_day AS day FROM bounds
  UNION ALL
  SELECT DATE_ADD(c.day, INTERVAL 1 DAY)
  FROM calendar AS c
  JOIN bounds AS b ON c.day < b.last_day
)
SELECT c.day,
       COUNT(o.id) AS orders,
       COALESCE(SUM(o.amount), 0) AS revenue
FROM calendar AS c
LEFT JOIN orders AS o ON o.ordered_on = c.day
GROUP BY c.day
ORDER BY c.day;
```

EXERCISE 2

Monthly signups for the first half of 2024

`signups` records the day each user signed up.

user_id	signed_up
1	2023-12-30
2	2024-01-05
3	2024-01-20
4	2024-03-31
5	2024-04-01
6	2024-04-15
7	2024-06-30
8	2024-07-01

Report **every month from January to June 2024** — six rows, even for months with no signups — with the columns `month` (as `YYYY-MM`), `signups` (that month's count) and `running_total` (signups from January up to and including that month; signups before 2024 do not count). Order by `month`.

```
month | signups | running_total
2024-01 | 2 | 2
2024-02 | 0 | 2
2024-03 | 1 | 3
2024-04 | 2 | 5
2024-05 | 0 | 5
2024-06 | 1 | 6
```

INPUT 1

```
CREATE TABLE signups (user_id INT
PRIMARY KEY, signed_up DATE);
INSERT INTO signups VALUES
  (1, '2023-12-30'), (2, '2024-01-05'),
(3, '2024-01-20'), (4, '2024-03-31'),
  (5, '2024-04-01'), (6, '2024-04-15'),
(7, '2024-06-30'), (8, '2024-07-01');
```

OUTPUT 1

```
month | signups | running_total
2024-01 | 2 | 2
2024-02 | 0 | 2
2024-03 | 1 | 3
2024-04 | 2 | 5
2024-05 | 0 | 5
2024-06 | 1 | 6
```

REFERENCE SOLUTION

```sql
WITH RECURSIVE months AS (
  SELECT DATE('2024-01-01') AS month_start
  UNION ALL
  SELECT DATE_ADD(month_start, INTERVAL 1 MONTH)
  FROM months
  WHERE month_start < '2024-06-01'
)
SELECT DATE_FORMAT(m.month_start, '%Y-%m') AS month,
       COUNT(s.user_id) AS signups,
       SUM(COUNT(s.user_id)) OVER (ORDER BY m.month_start) AS running_total
FROM months AS m
LEFT JOIN signups AS s
  ON s.signed_up ⩾ m.month_start
 AND s.signed_up <  DATE_ADD(m.month_start, INTERVAL 1 MONTH)
GROUP BY m.month_start
ORDER BY m.month_start;
```

QUIZ

1. A calendar is LEFT JOINed to `orders`. On a day with no orders, what do `COUNT(*)` and `COUNT(o.id)` return?

A. 0 and 0

B. 1 and 0

C. 1 and 1

D. NULL and 0

2. What happens to a MySQL recursive CTE that would need 1,500 iterations, with default settings?

A. It runs to completion

B. It stops with an error, because `cte_max_recursion_depth` defaults to 1,000

C. It silently returns the first 1,000 rows

D. It loops forever

3. A calendar is LEFT JOINed to `orders`, and `WHERE o.status = 'paid'` is added. What goes wrong?

A. Nothing

B. Empty days have a NULL status, fail the WHERE and disappear — the outer join behaves like an inner join

C. Paid orders are counted twice

D. The query fails

4. Why start a month series on the first of the month?

A. DATE_ADD only works on the 1st

B. Adding a month to the 31st clamps to a shorter month's last day (Jan 31 → Feb 29), and the series then drifts to that day for every later month

C. DATE_FORMAT needs it

D. It is required by RECURSIVE

ANSWER KEY · DATE SERIES — CALENDARS FROM RECURSIVE CTES AND FILLING MISSING DAYS

1. B The LEFT JOIN keeps the calendar row with NULLs on the order side, so the group holds one row. COUNT(*) counts it; COUNT(o.id) counts only non-NULL ids.

2. B MySQL aborts the statement once the recursion passes the limit. Raise the session variable, or use a calendar table or a cross join of smaller series.

3. B Conditions on the right-hand table of a LEFT JOIN belong in the ON clause: `LEFT JOIN orders o ON o.ordered_on = c.day AND o.status = 'paid'`.

4. B MySQL clamps an impossible date to the month's last day, so the next step starts from the 29th. Anchoring on the 1st avoids the drift, and half-open ranges (`⩾` start, `<` next start) cover each whole month.

CHECKPOINT · 30 MIN

Checkpoint — Query patterns

The checkpoint for query patterns: the shapes that reporting and interview questions take once the individual clauses are known — gaps and islands, deduplication, pivots, rolling windows, the nth highest value and date series.

How it works. Twelve questions and three queries; 70% on the questions and every query accepted clears the module.

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

- Why `date - ROW_NUMBER()` is constant along a run of consecutive dates, and what a duplicate date does to it.
- How LAG with a running SUM numbers islands when "consecutive" means something other than one step.
- How to find the gaps in a sequence with LEAD.
- How to keep one row per duplicate group, and why MySQL needs a derived table to delete the rest (error 1093).
- How SUM(CASE WHEN …) pivots rows into columns, why ELSE 0 matters, and how UNION ALL unpivots.
- The difference between a ROWS frame and a RANGE frame, and why a ROWS frame is not a 7-day window over missing dates.
- Why DENSE_RANK, not RANK, finds the nth highest distinct value, and how a scalar subquery makes "none" a NULL.
- How a recursive CTE builds a calendar, the 1,000-iteration limit, and why `COUNT(o.id)` and `COALESCE` follow the LEFT JOIN.

The three queries combine the patterns the way real reports do: each student's longest attendance streak (islands, then a maximum per student), a monthly ticket report with a column per status and the empty months filled in (a month series and a pivot together), and every department's top two pay levels side by side with the gap between them (the nth highest value per group, pivoted into columns).

EXERCISE 1

Longest attendance streak

`attendance` records each day a student attended; a day may have been recorded twice.

student	day
Ana	2024-09-02
Ana	2024-09-03
Ana	2024-09-04
Ana	2024-09-06
Ben	2024-09-02
Ben	2024-09-04
…	…

For each student, return `student` and `longest_streak`: the length in days of their longest run of consecutive attended days. Order by `longest_streak` descending, then `student`.

```
student | longest_streak
Cy | 4
Ana | 3
Dee | 2
Ben | 1
```

INPUT 1

```
CREATE TABLE attendance (student
VARCHAR(20), day DATE);
INSERT INTO attendance VALUES
  ('Ana', '2024-09-02'), ('Ana', '2024-
09-03'), ('Ana', '2024-09-04'), ('Ana',
'2024-09-06'),
  ('Ben', '2024-09-02'), ('Ben', '2024-
09-04'), ('Ben', '2024-09-06'),
  ('Cy',  '2024-09-03'), ('Cy',  '2024-
09-04'), ('Cy',  '2024-09-04'), ('Cy',
'2024-09-05'), ('Cy', '2024-09-06'),
  ('Dee', '2024-09-10'), ('Dee', '2024-
09-11');
```

OUTPUT 1

```
student | longest_streak
Cy | 4
Ana | 3
Dee | 2
Ben | 1
```

REFERENCE SOLUTION

```sql
WITH days AS (
  SELECT DISTINCT student, day FROM attendance
),
islands AS (
  SELECT student,
         DATE_SUB(day, INTERVAL ROW_NUMBER() OVER (PARTITION BY student ORDER BY day
  FROM days
),
streaks AS (
  SELECT student, COUNT(*) AS len
  FROM islands
  GROUP BY student, grp
)
SELECT student, MAX(len) AS longest_streak
FROM streaks
GROUP BY student
ORDER BY longest_streak DESC, student;
```

EXERCISE 2

Tickets per month by status

`tickets` has a status of `open`, `pending` or `closed`.

id	opened_on	status
1	2024-01-03	open
2	2024-01-15	closed
3	2024-01-20	closed
4	2024-03-02	pending
5	2024-03-09	open
6	2024-04-30	closed

Report **every month** from the month of the first ticket to the month of the last, inclusive — including months with no tickets — with the columns `month` (`YYYY-MM`), `open`, `pending`, `closed` (the number of tickets opened that month with that status) and `total`. Order by `month`.

```
month | open | pending | closed | total
2024-01 | 1 | 0 | 2 | 3
2024-02 | 0 | 0 | 0 | 0
2024-03 | 1 | 1 | 0 | 2
2024-04 | 0 | 0 | 1 | 1
```

INPUT 1

```
CREATE TABLE tickets (id INT PRIMARY
KEY, opened_on DATE, status
VARCHAR(10));
INSERT INTO tickets VALUES
  (1, '2024-01-03', 'open'),    (2,
'2024-01-15', 'closed'), (3, '2024-01-
20', 'closed'),
  (4, '2024-03-02', 'pending'), (5,
'2024-03-09', 'open'),   (6, '2024-04-
30', 'closed');
```

OUTPUT 1

```
month | open | pending | closed | total
2024-01 | 1 | 0 | 2 | 3
2024-02 | 0 | 0 | 0 | 0
2024-03 | 1 | 1 | 0 | 2
2024-04 | 0 | 0 | 1 | 1
```

REFERENCE SOLUTION

```sql
WITH RECURSIVE bounds AS (
  SELECT DATE(DATE_FORMAT(MIN(opened_on), '%Y-%m-01')) AS first_month,
         DATE(DATE_FORMAT(MAX(opened_on), '%Y-%m-01')) AS last_month
  FROM tickets
),
months AS (
  SELECT first_month AS month_start FROM bounds
  UNION ALL
  SELECT DATE_ADD(m.month_start, INTERVAL 1 MONTH)
  FROM months AS m
  JOIN bounds AS b ON m.month_start < b.last_month
)
SELECT DATE_FORMAT(m.month_start, '%Y-%m') AS month,
       COUNT(CASE WHEN t.status = 'open'    THEN 1 END) AS open,
       COUNT(CASE WHEN t.status = 'pending' THEN 1 END) AS pending,
       COUNT(CASE WHEN t.status = 'closed'  THEN 1 END) AS closed,
       COUNT(t.id) AS total
FROM months AS m
LEFT JOIN tickets AS t
  ON t.opened_on ⩾ m.month_start
 AND t.opened_on <  DATE_ADD(m.month_start, INTERVAL 1 MONTH)
GROUP BY m.month_start
ORDER BY m.month_start;
```

EXERCISE 3

Top two pay levels per department

`employees`:

id	name	dept	salary
1	Asha	eng	120
2	Bo	eng	120
3	Cara	eng	100
4	Dan	eng	90
5	Eli	ops	70
6	Fay	sales	80
7	Gus	sales	95
8	Hal	sales	60

For every department, return `dept`, `top_salary` (the highest distinct salary), `second_salary` (the second-highest distinct salary, NULL if none) and `gap` (`top_salary - second_salary`, NULL if there is no second). Order by `dept`.

```
dept | top_salary | second_salary | gap
eng | 120 | 100 | 20
ops | 70 | NULL | NULL
sales | 95 | 80 | 15
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(20), dept VARCHAR(10),
salary INT);
INSERT INTO employees VALUES
  (1, 'Asha', 'eng', 120), (2, 'Bo',
'eng', 120), (3, 'Cara', 'eng', 100),
(4, 'Dan', 'eng', 90),
  (5, 'Eli', 'ops', 70),
  (6, 'Fay', 'sales', 80), (7, 'Gus',
'sales', 95), (8, 'Hal', 'sales', 60);
```

OUTPUT 1

```
dept | top_salary | second_salary | gap
eng | 120 | 100 | 20
ops | 70 | NULL | NULL
sales | 95 | 80 | 15
```

REFERENCE SOLUTION

```sql
WITH ranked AS (
  SELECT dept, salary,
         DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS rk
  FROM employees
),
levels AS (
  SELECT dept,
         MAX(CASE WHEN rk = 1 THEN salary END) AS top_salary,
         MAX(CASE WHEN rk = 2 THEN salary END) AS second_salary
  FROM ranked
  GROUP BY dept
)
SELECT dept, top_salary, second_salary,
       top_salary - second_salary AS gap
FROM levels
ORDER BY dept;
```

CHECKPOINT QUESTIONS

1. A sequence of order ids is 1, 2, 3, 7, 8. What does `id - ROW_NUMBER() OVER (ORDER BY id)` give?

A. 0, 0, 0, 3, 3

B. 0, 0, 0, 4, 4

C. 1, 1, 1, 2, 2

D. 0, 1, 2, 3, 4

2. Which query keeps exactly one row per `email`, the most recently updated?

A. `SELECT DISTINCT email, name, updated_at FROM t`

B. `SELECT * FROM (SELECT t.*, ROW_NUMBER() OVER (PARTITION BY email ORDER BY updated_at DESC, id DESC) rn FROM t) x WHERE rn = 1`

C. `SELECT email, name, MAX(updated_at) FROM t GROUP BY email`

D. `SELECT * FROM t GROUP BY email HAVING COUNT(*) = 1`

3. What does MySQL error 1093 complain about?

A. A duplicate key

B. A DELETE or UPDATE whose subquery reads the table being modified

C. A missing GROUP BY column

D. Recursion past the limit

4. In MySQL, what does `SUM(status = 'open')` compute over a group?

A. An error — booleans cannot be summed

B. The number of rows whose status is 'open', because a true comparison is 1 and false is 0

C. The sum of the status column

D. 1 if any row is open

5. Which keyword stacks one SELECT per column when unpivoting, keeping every row?

A. `UNION`

B. `UNION ALL`

C. `INTERSECT`

D. `JOIN`

6. Payments exist on May 1, 3 and 9 only, one row each. On May 9, which rows does `SUM(amount) OVER (ORDER BY paid_on ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)` add up?

A. Only May 9

B. May 3 and May 9

C. May 1, May 3 and May 9

D. None

7. What does the frame `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING` compute with `AVG`?

A. A running average

B. A centred three-row average: the previous row, this row and the next one

C. The average of the whole partition

D. A two-row average

8. Salaries are 900, 800, 800, 700. What is `SELECT (SELECT DISTINCT salary FROM e ORDER BY salary DESC LIMIT 1 OFFSET 2) AS x`?

A. 800

B. 700

C. NULL

D. No rows

9. Why is `RANK() = 2` wrong for the second-highest salary when two people share the top salary?

A. RANK is not a window function

B. RANK leaves a gap after the tie — both top earners are 1 and the next is 3 — so no row has rank 2

C. RANK ignores ties

D. RANK sorts ascending

10. Which part of a recursive CTE decides the column's type in MySQL?

A. The recursive part

B. The anchor (non-recursive) SELECT

C. The outer query

D. The longest value produced

11. What is PostgreSQL's built-in replacement for a recursive date series?

A. `DATE_RANGE()`

B. `generate_series(start, stop, interval)`

C. `CONNECT BY LEVEL`

D. `SEQUENCE()`

12. A gap-filling report must count only paid orders. Where does `o.status = 'paid'` go?

A. In WHERE

B. In the LEFT JOIN's ON clause

C. In HAVING

D. In the recursive CTE

ANSWER KEY · CHECKPOINT — QUERY PATTERNS

1. A Row numbers are 1–5, so the differences are 0, 0, 0, 7−4 = 3 and 8−5 = 3: two islands, labelled 0 and 3.

2. B ROW_NUMBER with a total keeper rule picks one whole row per email. DISTINCT keeps rows that differ; selecting `name` beside MAX with GROUP BY email fails under ONLY_FULL_GROUP_BY or returns an arbitrary name; HAVING COUNT(*) = 1 drops the duplicated emails entirely.

3. B "You can't specify target table for update in FROM clause." Wrap the subquery in a derived table, or use a multi-table DELETE with a self-join.

4. B It is an idiomatic MySQL count of matches. It is not portable: PostgreSQL refuses to SUM a boolean and writes `COUNT(*) FILTER (WHERE status = 'open')`.

5. B UNION ALL keeps every row and skips the duplicate check that UNION performs.

6. C Six preceding *rows* reach back to May 1, eight days earlier. A true seven-day window on May 9 covers May 3 to May 9 only.

7. B Frames can extend forward as well as back. On the first and last rows the frame has only two rows.

8. B The distinct salaries are 900, 800, 700; skipping two leaves 700. The scalar-subquery wrapper would only matter if there were fewer than three distinct values.

9. B DENSE_RANK assigns 1, 1, 2, so rank 2 is the next distinct value.

10. B The anchor's types are the CTE's types. Anchoring on a bare string literal can make the column too short for later values — cast the anchor with `DATE(…)` or `CAST(… AS DATE)`.

11. B `generate_series` returns the series directly. Oracle traditionally uses CONNECT BY LEVEL; MySQL and SQL Server use recursive CTEs.

12. B In WHERE it would reject the NULL rows of empty days and turn the outer join into an inner one. In ON it only limits which orders join.