Module 16 of 16

Interview idioms

The six classic SQL interview questions solved two ways, the theory drill from WHERE vs HAVING to ACID, the idiom sheet of question shapes and their queries, and the twelve pitfalls that quietly fail a query.

Fifteen modules taught SQL. This one teaches the SQL round: the questions interviewers actually ask, the answers they listen for, and the habits that turn a query that runs into a query that is right.

It opens with the six classics every interview reuses — the second highest salary, duplicate emails, employees earning more than their manager, values three times in a row, the top three salaries per department and customers who never ordered — each with the expected answer, a second approach and the follow-ups it leads to. The theory drill is the part of the round without a keyboard: WHERE vs HAVING, the logical order of a query, DELETE vs TRUNCATE vs DROP, UNION vs UNION ALL, keys, normal forms, indexes, ACID and isolation, and ROW_NUMBER vs RANK vs DENSE_RANK, each as a one-sentence answer and the detail that proves you have used it. The idiom sheet maps question wording to query shape — existence and absence, latest per group, top N per group, conditional counts, share of the total, safe ratios, running totals, previous-row comparisons, pairs, median and mode, hierarchies, buckets. The pitfalls lesson is the checklist of twelve mistakes that return a wrong answer instead of an error: `= NULL`, NOT IN with a NULL, a LEFT JOIN undone by WHERE, fan-out sums, COUNT and AVG with NULLs, ungrouped columns, aliases in WHERE, integer division elsewhere, BETWEEN on datetimes, missing ORDER BY, and UNION eating rows.

The exercises are the classics written your own way and the theory made concrete: three readings in a row and the top three pay levels per department, WHERE and HAVING in one query and the three ranking functions side by side, share of payroll and a median per store, and two broken queries to fix — a fan-out join and a NOT IN that a single NULL empties. The final checkpoint sets the highest earners with ties kept, the customers with no order in a given year, and duplicate emails once case and spaces are ignored — and, with the other fifteen modules, completes the plan.

CONTENTS

1	The classic SQL questions — six problems every interview reuses	18 min
2	The SQL theory drill — the questions asked without a keyboard	18 min
3	The idiom sheet — the shapes SQL questions take and the query for each	17 min

4	SQL pitfalls — the twelve mistakes that fail interview queries	17 min
✓	Final checkpoint — Interview idioms	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 · 18 MIN

The classic SQL questions — six problems every interview reuses

Most SQL interview rounds draw from a small pool of questions. The wording changes — salaries become scores, emails become usernames, customers become students — but the shapes repeat, and an interviewer who asks one of them is not testing whether you can invent the answer. They are testing whether you know it, can write it cleanly, and can talk about its edge cases. This lesson works through the six that appear most often, each with the expected answer, a second approach, and the follow-up questions they lead to.

---

1. The second highest salary

Given an `employees` table, return the second highest salary, or NULL if there is none. The nth-highest lesson covered this in depth; the interview version wants two things said out loud — *distinct* salaries, and NULL when there is no answer.

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

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

Follow-ups: the nth highest (`DENSE_RANK() … = n` under a `MAX`), per department (partition the rank), and why `RANK` fails on ties (it skips rank 2 after a tie at the top).

---

2. Duplicate emails

Report every email address that appears in more than one row.

```sql
SELECT email
FROM users
GROUP BY email
HAVING COUNT(*) > 1;
```

This is a test of `HAVING`: the condition is on a group's count, so it cannot go in `WHERE`, which runs before grouping. The alternative is a self-join (`FROM users a JOIN users b ON a.email = b.email AND a.id < b.id`, then `SELECT DISTINCT a.email`) — correct but heavier. Follow-ups: case-insensitive matching (`GROUP BY LOWER(email)` — MySQL's default collation already ignores case, other engines do not), and "now delete the duplicates, keeping the lowest id", which is the deduplication lesson's `DELETE` with the derived-table wrapper or the self-join `DELETE`.

---

3. Earning more than their manager

Each employee row has a `manager_id` pointing at another employee. List the employees paid more than their manager.

```sql
SELECT e.name AS employee
FROM employees AS e
JOIN employees AS m ON e.manager_id = m.id
WHERE e.salary > m.salary;
```

A **self-join**: the same table twice under two aliases, one in the role of the employee and one as the manager. The inner join quietly drops employees with no manager (the CEO), which is right — there is nothing to compare. A correlated subquery says the same thing (`WHERE e.salary > (SELECT m.salary FROM employees m WHERE m.id = e.manager_id)`); a NULL manager makes the subquery NULL and the comparison unknown, so those rows drop out there too. Follow-ups: "show the manager's name and salary as well" (just select from `m`), "employees with no manager" (`WHERE manager_id IS NULL`), and "the whole chain of managers above someone", which needs a recursive CTE.

---

4. Three in a row

A `logs` table has a sequential `id` and a value `num`. Find every value that appears at least three times consecutively.

```sql
SELECT DISTINCT num AS consecutive_num
FROM (
  SELECT num,
         LEAD(num, 1) OVER (ORDER BY id) AS next1,
         LEAD(num, 2) OVER (ORDER BY id) AS next2
  FROM logs
) AS t
WHERE num = next1 AND num = next2;
```

`LEAD(num, 2)` reaches two rows ahead. A run of five equal values matches three times, hence `DISTINCT`. The pre-window answer is a three-way self-join:

```sql
SELECT DISTINCT a.num
FROM logs a
JOIN logs b ON b.id = a.id + 1 AND b.num = a.num
JOIN logs c ON c.id = a.id + 2 AND c.num = a.num;
```

The join version assumes the ids have no holes — `id + 1` must be the next row. The `LEAD` version only needs the ids to give the order, which is safer; say so. For "at least N in a row" with a large N, neither scales nicely, and the gaps-and-islands grouping is the general answer: the two-row-number difference groups each run, and `HAVING COUNT(*) ⩾ N` keeps the long ones.

---

5. The top three salaries in each department

Return each department's employees whose salary is among the three highest distinct salaries of that department.

```sql
SELECT d.name AS department, e.name AS employee, e.salary
FROM (
  SELECT e.*,
         DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) A
  FROM employees AS e
) AS e
JOIN departments AS d ON d.id = e.department_id
WHERE e.rk ⩽ 3
ORDER BY department, e.salary DESC, employee;
```

The whole question turns on the ranking function. "Top three *salaries*" means three salary levels, and everyone on those levels — `DENSE_RANK`. "Top three *earners*" with ties broken

arbitrarily would be `ROW_NUMBER`; "top three, but a tie for second pushes out the third" is `RANK`. Ask which is meant, or state your choice. The pre-window answer is a correlated count: keep a row when fewer than three distinct salaries in its department are higher (`WHERE 3 > (SELECT COUNT(DISTINCT s.salary) FROM employees s WHERE s.department_id = e.department_id AND s.salary > e.salary)`). It is correct, worth knowing, and quadratic.

---

6. Customers who never ordered

List the customers with no orders.

```sql
SELECT c.name
FROM customers AS c
LEFT JOIN orders AS o ON o.customer_id = c.id
WHERE o.id IS NULL;

-- or
SELECT c.name
FROM customers AS c
WHERE NOT EXISTS (SELECT 1 FROM orders AS o WHERE o.customer_id = c.id);
```

This is an **anti-join**: rows of one table with no match in another. Both forms are correct and MySQL 8 optimises them alike. The trap is the third form:

```sql
WHERE c.id NOT IN (SELECT customer_id FROM orders)   -- dangerous
```

If any `orders.customer_id` is NULL, `c.id NOT IN (…, NULL)` is never true — it is either false or unknown — and the query returns **no rows at all**. Use `NOT EXISTS` or the `LEFT JOIN`; if you write `NOT IN`, add `WHERE customer_id IS NOT NULL` inside. Interviewers ask this one partly to see whether you know that trap. A follow-up: "customers with no order *in 2024*" — the date condition belongs in the `ON` clause of the left join (or inside the `NOT EXISTS`), not in `WHERE`, or the anti-join breaks.

How to work through any of them

The pattern of a good answer is the same each time. Restate the question with its ambiguity resolved ("distinct salaries", "ties included"). Name the shape ("self-join", "anti-join", "top-N per group"). Write the version you trust, with a full `ORDER BY`. Then mention the alternative and

one edge case — NULLs, ties, an empty table, missing ids. Those last two sentences are what separate a memorised answer from an understood one, and they take ten seconds.

EXERCISE 1

Three readings in a row

A thermometer writes one row per reading; `id` gives the order, but some ids were skipped when a reading failed.

id	temp
1	21
2	21
3	21
4	22
5	21
6	23
7	23
8	23
9	23

Find every temperature that was recorded **at least three times in a row** (consecutive in `id` order). Return one column, `repeated_value`, each value once, ordered by `repeated_value`.

```
repeated_value
21
23
```

INPUT 1

```
CREATE TABLE readings (id INT PRIMARY
KEY, temp INT);
INSERT INTO readings VALUES
  (1, 21), (2, 21), (3, 21), (4, 22),
(5, 21),
  (6, 23), (7, 23), (8, 23), (9, 23);
```

OUTPUT 1

```
repeated_value
21
23
```

REFERENCE SOLUTION

```sql
SELECT DISTINCT temp AS repeated_value
FROM (
  SELECT temp,
         LEAD(temp, 1) OVER (ORDER BY id) AS next1,
         LEAD(temp, 2) OVER (ORDER BY id) AS next2
  FROM readings
) AS t
WHERE temp = next1 AND temp = next2
ORDER BY repeated_value;
```

EXERCISE 2

Top three pay levels per department

Two tables: `departments(id, name)` and `employees(id, name, salary, department_id)`.

department	employees (salary)
Engineering	Asha 120, Bo 110, Cara 110, Dev 100, Eli 90
Sales	Fay 80, Gus 80
Ops	Hal 70, Ivy 60, Jo 50, Kay 40
Legal	—

List every employee whose salary is among the **three highest distinct salaries of their department** — everyone on a tied level included. Return `department` (the department's name), `employee` and `salary`, ordered by `department`, then `salary` descending, then `employee`.

```
department | employee | salary
Engineering | Asha | 120
Engineering | Bo | 110
Engineering | Cara | 110
Engineering | Dev | 100
Ops | Hal | 70
Ops | Ivy | 60
Ops | Jo | 50
Sales | Fay | 80
Sales | Gus | 80
```

INPUT 1

```
CREATE TABLE departments (id INT PRIMARY
KEY, name VARCHAR(20));
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(20), salary INT,
department_id INT);
INSERT INTO departments VALUES (1,
'Engineering'), (2, 'Sales'), (3,
'Ops'), (4, 'Legal');
INSERT INTO employees VALUES
  (1, 'Asha', 120, 1), (2, 'Bo', 110,
1), (3, 'Cara', 110, 1), (4, 'Dev', 100,
1), (5, 'Eli', 90, 1),
  (6, 'Fay', 80, 2), (7, 'Gus', 80, 2),
  (8, 'Hal', 70, 3), (9, 'Ivy', 60, 3),
(10, 'Jo', 50, 3), (11, 'Kay', 40, 3);
```

OUTPUT 1

```
department | employee | salary
Engineering | Asha | 120
Engineering | Bo | 110
Engineering | Cara | 110
Engineering | Dev | 100
Ops | Hal | 70
Ops | Ivy | 60
Ops | Jo | 50
Sales | Fay | 80
Sales | Gus | 80
```

REFERENCE SOLUTION

```sql
SELECT d.name AS department, e.name AS employee, e.salary
FROM (
  SELECT emp.*,
         DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rk
  FROM employees AS emp
) AS e
JOIN departments AS d ON d.id = e.department_id
WHERE e.rk ⩽ 3
ORDER BY department, e.salary DESC, employee;
```

QUIZ

1. In "employees who earn more than their manager", what happens to the CEO, whose `manager_id` is NULL, in `FROM employees e JOIN employees m ON e.manager_id = m.id`?

A. The CEO is compared with themself

B. The CEO is left out — the inner join finds no manager row, and there is no one to out-earn

C. The query fails

D. The CEO always appears

2. Why is `WHERE id NOT IN (SELECT customer_id FROM orders)` risky for "customers who never ordered"?

A. NOT IN is slower than IN

B. If any `orders.customer_id` is NULL, the condition is never true and the query returns no rows

C. NOT IN cannot be used with subqueries

D. It returns duplicate customers

3. Why does the self-join answer to "three in a row" (`b.id = a.id + 1`, `c.id = a.id + 2`) fail when ids have holes?

A. It does not fail

B. It treats rows as consecutive only when their ids differ by exactly one, so a run across a skipped id is missed

C. It counts every run twice

D. Self-joins cannot compare the same column

4. Which query finds email addresses that appear more than once?

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

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

C. `SELECT DISTINCT email FROM users`

D. `SELECT email, COUNT(*) FROM users`

ANSWER KEY · THE CLASSIC SQL QUESTIONS — SIX PROBLEMS EVERY INTERVIEW REUSES

1. B `NULL = m.id` is never true, so the inner join drops the row. That is the right result: an employee with no manager cannot earn more than their manager.

2. B `id NOT IN (…, NULL)` includes `id <> NULL`, which is unknown. NOT EXISTS and LEFT JOIN … IS NULL are safe.

3. B `LEAD(…) OVER (ORDER BY id)` uses the *next row in order*, whatever its id, so it handles holes. The join version needs gap-free ids.

4. B The count is a property of the group, so the filter belongs in HAVING. An aggregate in WHERE is an error, and the last option selects a non-grouped column with an aggregate.

LESSON 2 · 18 MIN

The SQL theory drill — the questions asked without a keyboard

Part of every SQL interview happens without a keyboard. "What is the difference between WHERE and HAVING?" "Why would you use TRUNCATE instead of DELETE?" "What does ACID mean?" These are not trick questions; they have short, standard answers, and an interviewer listens for the precise sentence and for one supporting detail that shows you have used the thing. This lesson is the drill: each question, the answer to give, and the detail that makes it convincing. The quiz and the two exercises then make you use the two answers that are easiest to get wrong.

WHERE vs HAVING

WHERE filters rows before grouping; HAVING filters groups after. WHERE cannot contain an aggregate, because no groups exist yet; HAVING can. The detail: a condition on a plain column *can* be written in HAVING (MySQL allows it), but it belongs in WHERE, where it shrinks the input before the expensive grouping. A query can use both: `WHERE amount > 100` keeps the large orders, `GROUP BY customer_id`, `HAVING COUNT(*) ⩾ 2` keeps customers with at least two of them.

The logical order of a query

A SELECT is written in one order and evaluated in another:

```
FROM / JOIN  →  WHERE  →  GROUP BY  →  HAVING  →  SELECT (and window functions
             →  DISTINCT  →  ORDER BY  →  LIMIT / OFFSET
```

This explains a whole family of questions. A SELECT alias cannot be used in WHERE (WHERE runs first) but can in ORDER BY (which runs after). A window function cannot be filtered in WHERE — compute it in a subquery or CTE and filter outside. `ORDER BY` can sort by a column not in the SELECT list, except after `DISTINCT`. (MySQL bends the rule by letting HAVING see SELECT aliases; that is a MySQL extension, not the standard.) This is the *logical* order; the optimiser may physically execute things differently as long as the result is the same.

DELETE vs TRUNCATE vs DROP

	DELETE	TRUNCATE	DROP
Removes	matching rows	every row	the table
WHERE	yes	no	no
Rolled back	yes, in a transaction	no in MySQL (implicit commit)	no in MySQL
AUTO_INCREMENT	kept	reset	gone
Triggers	fire per row	do not fire	do not fire
Speed on a big table	slow, logged per row	fast	fast

The detail: TRUNCATE is DDL in MySQL — it drops and recreates the table, which is why it is fast, resets the counter, and cannot be undone inside a transaction. It also fails on a table referenced by a foreign key. (PostgreSQL's TRUNCATE, unlike MySQL's, is transactional — a good point to mention if the interviewer works across databases.)

UNION vs UNION ALL

UNION removes duplicate rows from the combined result; UNION ALL keeps them all. UNION therefore sorts or hashes the whole result, so UNION ALL is faster and is the right choice whenever duplicates are impossible or wanted. Both require the same number of columns with compatible types, and take the column names from the first SELECT. INTERSECT and EXCEPT (MySQL 8.0.31+) complete the set operations.

Keys

- A **primary key** uniquely identifies each row; it is unique and NOT NULL, and a table has at most one (possibly over several columns — a composite key). In InnoDB it is also the clustered index: the table's rows are stored in primary-key order.
- A **unique key** also forbids duplicates, but a table can have several, and in MySQL a unique column may hold many NULLs.

- A **foreign key** makes a column refer to a primary or unique key of another table; the database refuses orphans and can cascade deletes or updates (`ON DELETE CASCADE`).
- A **candidate key** is any minimal set of columns that could be the primary key; a **surrogate key** is an invented id (AUTO_INCREMENT) as opposed to a **natural key** from the data (an email, an ISBN).

Normal forms

- **1NF**: every column holds one atomic value — no lists in a cell, no repeating groups (`phone1, phone2, phone3`).
- **2NF**: 1NF, and no column depends on only *part* of a composite key. In `order_items(order_id, product_id, product_name, qty)`, `product_name` depends on `product_id` alone — move it to `products`.
- **3NF**: 2NF, and no column depends on another non-key column. `employees(id, dept_id, dept_name)` stores `dept_name` through `dept_id` — move it to `departments`.
- **BCNF** tightens 3NF: every determinant must be a candidate key.

The detail: normalisation removes update anomalies (change a department's name in one place, not in every employee row); **denormalisation** deliberately reintroduces redundancy for read speed — reporting tables, cached counters — and accepts the cost of keeping copies in sync.

Indexes

An index is a separate, sorted structure — a B-tree in InnoDB — that lets the database find rows by a column's value without reading the whole table. It speeds up reads that filter, join or sort on its columns and slows down writes, which must maintain it. The details interviewers probe:

- A **composite index** on `(a, b, c)` serves filters on `a`, on `a, b`, and on `a, b, c` — the *leftmost prefix* rule — but not on `b` alone.
- An index is not used when the column is wrapped in a function (`WHERE YEAR(created) = 2024`) or compared after an implicit conversion; rewrite it as a range (`created ⩾ '2024-01-01' AND created < '2025-01-01'`) — a **sargable** condition.
- A **covering index** holds every column the query needs, so the table itself is never read.

- A **clustered** index stores the rows themselves (InnoDB's primary key); a **secondary** index stores the key plus the primary key.
- Low-selectivity columns (a yes/no flag) rarely benefit; `EXPLAIN` shows whether an index is used.

ACID and isolation

Atomicity: all of a transaction's changes happen, or none. **Consistency**: it takes the database from one valid state to another — constraints hold. **Isolation**: concurrent transactions do not see each other's uncommitted work, to the degree the isolation level promises. **Durability**: once committed, the change survives a crash (InnoDB's redo log).

The four isolation levels, weakest first, are READ UNCOMMITTED (dirty reads possible), READ COMMITTED (no dirty reads, but a re-read can change), REPEATABLE READ (MySQL InnoDB's default — re-reads see the same snapshot) and SERIALIZABLE. The anomalies they prevent are dirty reads, non-repeatable reads and phantom reads.

RANK vs DENSE_RANK vs ROW_NUMBER

For scores 100, 90, 90, 80 in descending order:

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

ROW_NUMBER is unique per row (and arbitrary between ties unless the ORDER BY breaks them); RANK shares ties and skips the following numbers, like a sports table; DENSE_RANK shares ties and skips nothing. The detail: "top three per group" questions hinge on which one is meant.

A few more, quickly

- **CHAR vs VARCHAR**: fixed length, padded with spaces, versus variable length with a length prefix.
- **A view** is a stored query; it holds no data (MySQL has no materialised views).
- **COUNT(*) vs COUNT(col) vs COUNT(DISTINCT col)**: rows; non-NULL values; distinct non-NULL values.
- **A correlated subquery** refers to the outer query's row and is evaluated per row.
- **INNER vs LEFT JOIN**: only matches, versus every left row with NULLs where nothing matched.
- **A stored procedure** is server-side code called with `CALL`; a **trigger** runs automatically on INSERT, UPDATE or DELETE.

Say each answer in one sentence, then add the detail. That rhythm — definition, then proof of use — is what a theory round is listening for.

EXERCISE 1

Big spenders: WHERE then HAVING

`orders(id, customer_id, amount)`:

id	customer_id	amount
1	1	150
2	1	90
3	1	200
4	2	100
5	2	100
6	3	500
7–9	4	99 each
10	5	120
11	5	130
12	5	140

A *big order* is one of **100 or more**. Find the customers with **at least two big orders**, and for each return `customer_id`, `big_orders` (how many big orders) and `big_total` (the sum of their big orders only). Order by `big_total` descending, then `customer_id`.

Use WHERE for the row condition and HAVING for the group condition.

```
customer_id | big_orders | big_total
5 | 3 | 390
1 | 2 | 350
2 | 2 | 200
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, amount INT);
INSERT INTO orders VALUES
  (1, 1, 150), (2, 1, 90), (3, 1, 200),
  (4, 2, 100), (5, 2, 100),
  (6, 3, 500),
  (7, 4, 99), (8, 4, 99), (9, 4, 99),
  (10, 5, 120), (11, 5, 130), (12, 5,
140);
```

OUTPUT 1

```
customer_id | big_orders | big_total
5 | 3 | 390
1 | 2 | 350
2 | 2 | 200
```

REFERENCE SOLUTION

```
SELECT customer_id, COUNT(*) AS big_orders, SUM(amount) AS big_total
FROM orders
WHERE amount ⩾ 100
GROUP BY customer_id
HAVING COUNT(*) ⩾ 2
ORDER BY big_total DESC, customer_id;
```

EXERCISE 2

Three ranking functions side by side

`scores(player, points)`:

player	points
Ana	100
Ben	90
Cid	90
Dee	80
Eva	80
Fin	70

Return every player with `player`, `points` and three rankings by points, highest first: `row_num` (`ROW_NUMBER`, ties broken by `player` alphabetically), `rnk` (`RANK`) and `dense_rnk` (`DENSE_RANK`). Order the result by `points` descending, then `player`.

```
player | points | row_num | rnk | dense_rnk
Ana | 100 | 1 | 1 | 1
Ben | 90 | 2 | 2 | 2
Cid | 90 | 3 | 2 | 2
Dee | 80 | 4 | 4 | 3
Eva | 80 | 5 | 4 | 3
Fin | 70 | 6 | 6 | 4
```

INPUT 1

```
CREATE TABLE scores (player VARCHAR(20)
PRIMARY KEY, points INT);
INSERT INTO scores VALUES
  ('Ana', 100), ('Ben', 90), ('Cid',
90), ('Dee', 80), ('Eva', 80), ('Fin',
70);
```

OUTPUT 1

```
player | points | row_num | rnk |
dense_rnk
Ana | 100 | 1 | 1 | 1
Ben | 90 | 2 | 2 | 2
Cid | 90 | 3 | 2 | 2
Dee | 80 | 4 | 4 | 3
Eva | 80 | 5 | 4 | 3
Fin | 70 | 6 | 6 | 4
```

REFERENCE SOLUTION

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

QUIZ

1. In which logical order are these clauses evaluated?

A. SELECT → FROM → WHERE → GROUP BY → HAVING → ORDER BY

B. FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT

C. FROM → GROUP BY → WHERE → SELECT → HAVING → ORDER BY

D. WHERE → FROM → SELECT → GROUP BY → ORDER BY → HAVING

2. Which statement is true of `TRUNCATE TABLE orders` in MySQL?

A. It takes a WHERE clause

B. It removes every row, resets AUTO_INCREMENT, fires no DELETE triggers, and commits implicitly — it cannot be rolled back

C. It removes the table definition

D. It is slower than DELETE on a large table

3. A table `order_items(order_id, product_id, product_name, qty)` has the key `(order_id, product_id)`. Which normal form does `product_name` break?

A. 1NF

B. 2NF — it depends on only part of the composite key

C. 3NF only

D. None

4. A composite index exists on `(last_name, first_name, city)`. Which filter can use it?

A. `WHERE first_name = 'Ana'`

B. `WHERE city = 'Pune'`

C. `WHERE last_name = 'Rao' AND first_name = 'Ana'`

D. `WHERE first_name = 'Ana' AND city = 'Pune'`

5. Which ACID property guarantees that a committed transfer is still there after a power cut?

A. Atomicity

B. Consistency

C. Isolation

D. Durability

ANSWER KEY · THE SQL THEORY DRILL — THE QUESTIONS ASKED WITHOUT A KEYBOARD

1. B That is why a SELECT alias cannot be used in WHERE but can be used in ORDER BY, and why a window function, computed with the SELECT list, cannot be filtered in WHERE.

2. B TRUNCATE is DDL in MySQL: it recreates the table. DELETE removes matching rows one by one and can be rolled back; DROP removes the table itself.

3. B `product_name` is determined by `product_id` alone, a partial dependency on the key. Moving it to a `products` table fixes it. 3NF is about a non-key column depending on another non-key column.

4. C The leftmost-prefix rule: the index serves `last_name`, `last_name + first_name`, and all three. A filter that skips the first column cannot seek on it.

5. D Durability: once committed, changes survive a crash — InnoDB writes them to its redo log before acknowledging the commit. Atomicity is all-or-nothing; isolation is about concurrent transactions; consistency is about constraints holding.

LESSON 3 · 17 MIN

The idiom sheet — the shapes SQL questions take and the query for each

The previous modules taught the clauses; the query-patterns module taught the bigger shapes. This lesson is the compressed version for the day of the interview: a sheet of question shapes, each recognised by its wording, each with the few lines of MySQL that answer it. Fluent candidates do not invent these during the round — they recognise "latest per customer" or "share of the total" and type the idiom. Read it once slowly, then come back to it as a checklist.

Existence and absence

"Customers who have ordered", *"products never sold"*, *"users with no login this month"*.

```sql
-- has at least one
SELECT c.* FROM customers c
WHERE EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.id);

-- has none (anti-join)
SELECT c.* FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
WHERE o.id IS NULL;
```

`EXISTS` stops at the first match and never duplicates the outer row, unlike an inner join to a table with many matches. For absence prefer `NOT EXISTS` or the left join over `NOT IN`, which a single NULL silently breaks.

Latest (or first) row per group

"Each customer's most recent order", *"the current price of each product"*.

```sql
SELECT * FROM (
  SELECT o.*, ROW_NUMBER() OVER (
           PARTITION BY customer_id ORDER BY created_at DESC, id DESC) AS rn
  FROM orders o
) t
WHERE rn = 1;
```

End the window's `ORDER BY` with a unique column so the winner is fixed. The pre-window idiom joins to `SELECT customer_id, MAX(created_at) … GROUP BY customer_id`; it returns two rows when two orders share the latest time.

Top N per group

"The three best-selling products in each category". Same as above with `rn ⩽ 3`. Choose the function by the meaning of ties: `ROW_NUMBER` for exactly N rows, `DENSE_RANK` for N value levels including everyone tied, `RANK` for N places where ties consume places.

Counting several things at once

"For each month, how many orders were paid, refunded and cancelled?" — conditional aggregation:

```sql
SELECT DATE_FORMAT(created_at, '%Y-%m') AS month,
       SUM(status = 'paid')     AS paid,
       SUM(status = 'refunded') AS refunded,
       COUNT(*)                 AS total
FROM orders
GROUP BY month;
```

`COUNT(CASE WHEN … THEN 1 END)` is the portable form. The same idiom gives ratios in one pass: `AVG(status = 'refunded')` is the refund rate, because the average of 0s and 1s is the share of 1s.

Share of the total

"Each employee's share of their department's payroll", *"each product's percentage of revenue"*.

```
SELECT dept, name, salary,
       ROUND(100 * salary / SUM(salary) OVER (PARTITION BY dept), 1) AS pct
FROM employees;
```

A window aggregate gives every row its group's total without collapsing the rows — the key difference from `GROUP BY`. `SUM(salary) OVER ()`, with an empty window, is the grand total.

Ratios that cannot divide by zero

"Conversion rate per campaign".

```
SELECT campaign,
       ROUND(100 * SUM(converted) / NULLIF(COUNT(*), 0), 2) AS conversion_pct
FROM visits GROUP BY campaign;
```

`NULLIF(x, 0)` turns a zero denominator into NULL. In MySQL `/` is always decimal division; in PostgreSQL and SQL Server, integer divided by integer truncates — multiply by `1.0` there.

Running totals and moving windows

"Cumulative revenue by day", *"7-day moving average"*.

```
SUM(amount) OVER (ORDER BY day ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT R(
AVG(amount) OVER (ORDER BY day ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
```

Say `ROWS` explicitly — the default frame is `RANGE`, which treats tied dates as one step — and remember that a ROWS frame is a 7-*day* window only with one row per day.

Compare with the previous row

"Day-over-day change", *"time since the user's previous visit"*, *"rows where the value went up"*.

```sql
SELECT day, revenue,
       revenue - LAG(revenue) OVER (ORDER BY day) AS change_abs,
       DATEDIFF(day, LAG(day) OVER (ORDER BY day)) AS days_since
FROM daily;
```

`LAG(x, n, default)` reaches n rows back, with a default instead of NULL at the start; `LEAD` reaches forward. To filter on the result ("days where revenue fell"), compute it in a CTE and filter outside — a window function cannot appear in `WHERE`.

Pairs and combinations

"All possible matches between teams", *"pairs of users who share a city"*.

```sql
SELECT a.name, b.name
FROM teams a JOIN teams b ON a.id < b.id;
```

`a.id < b.id` removes self-pairs and mirrored duplicates in one condition; `<>` would list every pair twice. Add `AND a.city = b.city` for pairs that share something.

Median and mode

MySQL has no `MEDIAN`. Number the rows and average the middle one or two:

```sql
WITH r AS (
  SELECT store, amount,
         ROW_NUMBER() OVER (PARTITION BY store ORDER BY amount) AS rn,
         COUNT(*)     OVER (PARTITION BY store)                 AS n
  FROM orders
)
SELECT store, ROUND(AVG(amount), 2) AS median_amount
FROM r
WHERE rn IN (FLOOR((n + 1) / 2), CEIL((n + 1) / 2))
GROUP BY store;
```

For an odd `n` both expressions name the same middle row; for an even `n` they name the two middle rows, and `AVG` takes their mean. PostgreSQL has `PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY amount)`. The **mode** — the most frequent value — is `GROUP BY value ORDER BY COUNT(*) DESC, value LIMIT 1`, with the tie-breaker stated.

Distinct counts and lists

"How many different products did each customer buy?" is `COUNT(DISTINCT product_id)` grouped by customer. *"List them"* is `GROUP_CONCAT(DISTINCT name ORDER BY name SEPARATOR ', ')` in MySQL (`STRING_AGG` in PostgreSQL and SQL Server). GROUP_CONCAT's result is cut at `group_concat_max_len`, 1,024 bytes by default — worth a sentence if the list can be long.

Hierarchies and sequences

"Everyone who reports to Asha, at any depth", *"the numbers 1 to 100"*, *"every date in March"* — recursive CTEs:

```
WITH RECURSIVE reports AS (
  SELECT id, name, 1 AS depth FROM employees WHERE manager_id = 7
  UNION ALL
  SELECT e.id, e.name, r.depth + 1
  FROM employees e JOIN reports r ON e.manager_id = r.id
)
SELECT * FROM reports;
```

Bucketing

"Orders by size band", *"users by age group"* — a `CASE` in both the SELECT and the GROUP BY (MySQL lets you group by the alias):

```
SELECT CASE WHEN amount < 100 THEN 'small'
            WHEN amount < 500 THEN 'medium'
            ELSE 'large' END AS band,
       COUNT(*) AS orders
FROM orders GROUP BY band;
```

Order the bands deliberately — alphabetical order (`large, medium, small`) is rarely what the reader wants; `ORDER BY MIN(amount)` sorts them by size.

Using the sheet

When a question arrives, find its shape in the wording: "each … latest" is a ROW_NUMBER, "never" is an anti-join, "share" is a window sum, "compared with the previous" is LAG, "in a row" is LEAD or gaps and islands, "for every day, even empty ones" is a calendar. Then write the idiom, add the full ORDER BY, and mention the edge — ties, NULLs, empty groups — that the idiom handles or does not.

EXERCISE 1

Share of department payroll

`employees(id, name, dept, salary)`:

id	name	dept	salary
1	Asha	eng	120
2	Bo	eng	80
3	Cara	eng	100
4	Dev	ops	50
5	Eli	ops	30
6	Fay	ops	20
7	Gus	sales	100

For every employee, return `dept`, `name`, `salary` and `pct`: their salary as a percentage of their department's total salary, rounded to 1 decimal place. Order by `dept`, then `salary` descending, then `name`.

```
dept | name | salary | pct
eng | Asha | 120 | 40
eng | Cara | 100 | 33.3
eng | Bo | 80 | 26.7
ops | Dev | 50 | 50
ops | Eli | 30 | 30
ops | Fay | 20 | 20
sales | Gus | 100 | 100
```

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', 80), (3, 'Cara', 'eng', 100),
  (4, 'Dev', 'ops', 50), (5, 'Eli',
'ops', 30), (6, 'Fay', 'ops', 20),
  (7, 'Gus', 'sales', 100);
```

OUTPUT 1

```
dept | name | salary | pct
eng | Asha | 120 | 40
eng | Cara | 100 | 33.3
eng | Bo | 80 | 26.7
ops | Dev | 50 | 50
ops | Eli | 30 | 30
ops | Fay | 20 | 20
sales | Gus | 100 | 100
```

REFERENCE SOLUTION

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

EXERCISE 2

Median order value per store

`orders(id, store, amount)`:

id	store	amount
1	A	10
2	A	30
3	A	20
4	B	5
5	B	100
6	B	16
7	B	25
8	C	7

MySQL has no `MEDIAN` function. For each store return `store` and `median_amount` — the middle amount when the store has an odd number of orders, the mean of the two middle amounts when it has an even number — rounded to 2 decimal places. Order by `store`.

```
store | median_amount
A | 20
B | 20.5
C | 7
```

INPUT 1

```
CREATE TABLE orders (id INT PRIMARY KEY,
store VARCHAR(10), amount INT);
INSERT INTO orders VALUES
  (1, 'A', 10), (2, 'A', 30), (3, 'A',
20),
  (4, 'B', 5), (5, 'B', 100), (6, 'B',
16), (7, 'B', 25),
  (8, 'C', 7);
```

OUTPUT 1

```
store | median_amount
A | 20
B | 20.5
C | 7
```

REFERENCE SOLUTION

```sql
WITH ranked AS (
  SELECT store, amount,
         ROW_NUMBER() OVER (PARTITION BY store ORDER BY amount) AS rn,
         COUNT(*)     OVER (PARTITION BY store)                 AS n
  FROM orders
)
SELECT store, ROUND(AVG(amount), 2) AS median_amount
FROM ranked
WHERE rn IN (FLOOR((n + 1) / 2), CEIL((n + 1) / 2))
GROUP BY store
ORDER BY store;
```

QUIZ

1. Which question is an anti-join?

A. Each customer's latest order

B. Products that have never been sold

C. Each product's share of revenue

D. Monthly revenue

2. Why does `FROM teams a JOIN teams b ON a.id < b.id` list each pair once?

A. Because JOIN removes duplicates

B. The strict inequality drops a team paired with itself and keeps only one of (A, B) and (B, A)

C. Because ids are unique

D. It lists each pair twice

3. In MySQL, what does `AVG(status = 'refunded')` return over a group of orders?

A. An error

B. The share of orders that were refunded, as a fraction between 0 and 1

C. The number of refunded orders

D. 1 if any order was refunded

4. Which is the portable way to give each row its group's total while keeping every row?

A. `GROUP BY` the group column

B. `SUM(x) OVER (PARTITION BY g)`

C. `SUM(x) GROUP BY g HAVING …`

D. `DISTINCT SUM(x)`

ANSWER KEY · THE IDIOM SHEET — THE SHAPES SQL QUESTIONS TAKE AND THE QUERY FOR EACH

1. B "Never" means rows with no match in another table: `NOT EXISTS`, or `LEFT JOIN … WHERE right.id IS NULL`.

2. B With `<>` both (A, B) and (B, A) would appear; with `<` only the ordering where the first id is smaller survives.

3. B A comparison is 1 or 0 in MySQL, and the average of 0s and 1s is the proportion of 1s. `SUM(…)` of the same expression would be the count.

4. B A window aggregate attaches the group total to every row; GROUP BY collapses the rows into one per group.

LESSON 4 · 17 MIN

SQL pitfalls — the twelve mistakes that fail interview queries

SQL rarely fails loudly. A wrong query usually runs, returns a plausible-looking table, and is simply wrong — too many rows, too few, a total that is double what it should be. Interviewers know which mistakes produce those quiet failures, and many questions are built to trigger one. This lesson is the checklist: twelve mistakes, why each happens, and the fix. Read your query against it before you say "done".

---

1. `= NULL`

```
WHERE manager_id = NULL      -- always unknown: returns nothing
WHERE manager_id IS NULL     -- right
```

Any comparison with NULL is unknown, and WHERE keeps only true. The same goes for `<> NULL`. MySQL's NULL-safe `<=>` operator treats two NULLs as equal, when that is what you want.

---

2. NOT IN against a column with NULLs

```
SELECT name FROM customers
WHERE id NOT IN (SELECT customer_id FROM complaints);
```

If one complaint has a NULL `customer_id`, this returns **no rows**. `id NOT IN (3, 5, NULL)` expands to `id <> 3 AND id <> 5 AND id <> NULL`; the last part is unknown, so the conjunction is never true. Use `NOT EXISTS (SELECT 1 FROM complaints c WHERE c.customer_id = customers.id)`, or a LEFT JOIN with `IS NULL`. `IN` has no such problem — a NULL in the list just never matches.

---

3. The LEFT JOIN undone by WHERE

```sql
SELECT c.name, COUNT(o.id)
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
WHERE o.status = 'paid'            -- drops customers without paid orders
GROUP BY c.name;
```

Customers with no orders have `o.status` NULL, which fails the WHERE. The LEFT JOIN is now an inner join and the zero rows are gone. Move the condition into the join: `LEFT JOIN orders o ON o.customer_id = c.id AND o.status = 'paid'`. The rule: conditions on the *outer* (preserved) table go in WHERE; conditions on the *optional* table go in ON.

---

4. Fan-out: summing after a join that multiplies rows

A customer has two orders and three payments. Joining both to the customer gives 2 × 3 = 6 rows, and every order amount is counted three times, every payment twice:

```sql
-- wrong
SELECT c.name, SUM(o.amount) AS ordered, SUM(p.amount) AS paid
FROM customers c
LEFT JOIN orders o   ON o.customer_id = c.id
LEFT JOIN payments p ON p.customer_id = c.id
GROUP BY c.name;
```

Aggregate each child *before* joining, so each side contributes one row per customer:

```sql
SELECT c.name,
       COALESCE(o.total, 0) AS ordered,
       COALESCE(p.total, 0) AS paid
FROM customers c
LEFT JOIN (SELECT customer_id, SUM(amount) AS total FROM orders   GROUP BY cus
       ON o.customer_id = c.id
LEFT JOIN (SELECT customer_id, SUM(amount) AS total FROM payments GROUP BY cus
       ON p.customer_id = c.id;
```

Correlated scalar subqueries in the SELECT list also work. The same fan-out bites when summing a parent's column (an order's shipping fee) after joining its items. `COUNT(DISTINCT …)` can hide it for counts but not for sums.

5. COUNT(*) versus COUNT(column)

`COUNT(*)` counts rows; `COUNT(col)` counts non-NULL values of `col`; `COUNT(DISTINCT col)` counts distinct non-NULL values. After a LEFT JOIN, `COUNT(*)` counts an unmatched row as 1 — count a column from the right-hand table to get 0.

6. AVG and SUM skip NULLs

`AVG(score)` over 80, NULL, 100 is 90, not 60: the NULL row is not in the denominator. If a missing score should count as zero, say so with `AVG(COALESCE(score, 0))`. And `SUM` over no rows, or only NULLs, is NULL, not 0 — wrap it in `COALESCE` for display.

7. Selecting columns that are not grouped

```
SELECT dept, name, MAX(salary) FROM employees GROUP BY dept;
```

Which `name`? Under `ONLY_FULL_GROUP_BY` (on by default since MySQL 5.7) the query is refused; with the mode off, MySQL returns a name from an arbitrary row — usually *not* the top earner. Every selected column must be grouped, aggregated or functionally dependent on the group key. For "the row with the maximum", use ROW_NUMBER or join back to the maximum.

8. An alias or a window function in WHERE

WHERE runs before SELECT, so `WHERE total > 100` with `total` defined in the SELECT list fails, and `WHERE ROW_NUMBER() OVER (…) = 1` is not allowed. Repeat the expression, or compute it in a CTE or derived table and filter outside. HAVING can test an aggregate directly.

9. Integer division (in other databases)

In MySQL, `/` always returns a decimal: `7 / 2` is `3.5000`, and `DIV` is integer division. In PostgreSQL and SQL Server, `7 / 2` is `3` — integer divided by integer truncates — so `100 *`

`paid / total` for a percentage silently returns whole numbers or 0. Write `100.0 * paid / total` to be safe across engines, and say why.

---

10. Dates as text and BETWEEN on datetimes

`WHERE created_at BETWEEN '2024-03-01' AND '2024-03-31'` on a DATETIME column stops at `2024-03-31 00:00:00` and misses the last day. Use a half-open range: `created_at ⩾ '2024-03-01' AND created_at < '2024-04-01'`. The same form keeps the condition sargable — `WHERE MONTH(created_at) = 3` reads every row and ignores any index (and matches March of every year).

---

11. No ORDER BY, or an incomplete one

A table has no order; without `ORDER BY` the database returns rows in whatever order is cheapest, which can change with an index, a new MySQL version or the data size. `ORDER BY` inside a subquery or view is not a promise about the outer result either. `LIMIT` without `ORDER BY` picks arbitrary rows. And an `ORDER BY score DESC` with ties leaves the tied rows' order undefined — add a unique tie-breaker (`, id`) whenever the order matters.

---

12. UNION removing rows you meant to keep

`UNION` removes duplicates across the whole combined result — including duplicates *within* one side. When two rows really are two facts (two payments of 100 on one day), `UNION` merges them into one. Use `UNION ALL` unless removing duplicates is the point.

Three more to keep in mind

- **Case and collation.** MySQL's default collation is case-insensitive, so `WHERE name = 'asha'` matches `'Asha'` — on PostgreSQL it does not. Trailing spaces are also ignored in many MySQL comparisons.
- **`DISTINCT` as a bandage.** If a join produced duplicates, `SELECT DISTINCT` hides the symptom; find the join that fanned out instead, because the sums are still wrong.
- **`OR` precedence.** `WHERE a = 1 OR a = 2 AND b = 3` means `a = 1 OR (a = 2 AND b = 3)`. Parenthesise every mix of AND and OR.

The habit

Before saying a query is finished, run it mentally against three small inputs: one with a NULL in every nullable column, one with a tie, and one with an empty group or no matching rows. Most of the twelve mistakes show up on one of those three, and saying "let me check NULLs and ties" aloud is itself a point in your favour.

EXERCISE 1

Fix the fan-out

Three tables: `customers(id, name)`, `orders(id, customer_id, amount)` and `payments(id, customer_id, amount)`.

customer	orders	payments
Asha	100, 50	60, 40, 50
Ben	70	—
Cara	—	20
Dev	—	—

The starter joins both child tables to the customers and sums — and reports Asha as having ordered 450 and paid 300. Fix it. For every customer return `name`, `ordered` (the sum of their orders) and `paid` (the sum of their payments), **0** where there are none, ordered by `name`.

```
name | ordered | paid
Asha | 150 | 150
Ben | 70 | 0
Cara | 0 | 20
Dev | 0 | 0
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(20));
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, amount INT);
CREATE TABLE payments (id INT PRIMARY
KEY, customer_id INT, amount INT);
INSERT INTO customers VALUES (1,
'Asha'), (2, 'Ben'), (3, 'Cara'), (4,
'Dev');
INSERT INTO orders VALUES (1, 1, 100),
(2, 1, 50), (3, 2, 70);
INSERT INTO payments VALUES (1, 1, 60),
(2, 1, 40), (3, 1, 50), (4, 3, 20);
```

OUTPUT 1

```
name | ordered | paid
Asha | 150 | 150
Ben | 70 | 0
Cara | 0 | 20
Dev | 0 | 0
```

REFERENCE SOLUTION

```sql
SELECT c.name,
       COALESCE(o.total, 0) AS ordered,
       COALESCE(p.total, 0) AS paid
FROM customers AS c
LEFT JOIN (SELECT customer_id, SUM(amount) AS total FROM orders GROUP BY customer_id
       ON o.customer_id = c.id
LEFT JOIN (SELECT customer_id, SUM(amount) AS total FROM payments GROUP BY customer_
       ON p.customer_id = c.id
ORDER BY c.name;
```

EXERCISE 2

Customers without a complaint

`customers(id, name)` and `complaints(id, customer_id, note)`. A complaint sent through the anonymous form has a NULL `customer_id`.

customers	complaints (customer_id, note)
1 Asha	1, late delivery
2 Ben	3, broken lid
3 Cara	NULL, anonymous form
4 Dev	

The starter uses `NOT IN` and returns no rows at all. Fix it so it lists the customers who have **no complaint**: one column, `name`, ordered by `name`.

```
name
Ben
Dev
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(20));
CREATE TABLE complaints (id INT PRIMARY
KEY, customer_id INT, note VARCHAR(40));
INSERT INTO customers VALUES (1,
'Asha'), (2, 'Ben'), (3, 'Cara'), (4,
'Dev');
INSERT INTO complaints VALUES (1, 1,
'late delivery'), (2, 3, 'broken lid'),
(3, NULL, 'anonymous form');
```

OUTPUT 1

```
name
Ben
Dev
```

REFERENCE SOLUTION

```sql
SELECT c.name
FROM customers AS c
WHERE NOT EXISTS (
  SELECT 1 FROM complaints AS x WHERE x.customer_id = c.id
)
ORDER BY c.name;
```

QUIZ

1. What does `SELECT * FROM employees WHERE manager_id = NULL` return?

A. The employees with no manager

B. No rows

C. Every row

D. An error

2. `SELECT c.name, COUNT(o.id) FROM customers c LEFT JOIN orders o ON o.customer_id = c.id WHERE o.status = 'paid' GROUP BY c.name` — which customers are missing from the result?

A. None

B. Customers with no paid order, because their NULL or other status fails the WHERE

C. Customers with more than one order

D. Customers with a NULL name

3. Scores are 80, NULL and 100. What is `AVG(score)`?

A. 60

B. 90

C. NULL

D. 180

4. Which filter returns every row from March 2024 of a DATETIME column `created_at`, and can use an index on it?

A. `WHERE MONTH(created_at) = 3`

B. `WHERE created_at BETWEEN '2024-03-01' AND '2024-03-31'`

C. `WHERE created_at ⩾ '2024-03-01' AND created_at < '2024-04-01'`

D. `WHERE DATE_FORMAT(created_at, '%Y-%m') = '2024-03'`

5. In PostgreSQL, what is `100 * 3 / 4` when all three are integers, and in MySQL?

A. 75 in both

B. 75 in PostgreSQL, 75.0000 in MySQL — but `3 / 4 * 100` would be 0 in PostgreSQL and 75.0000 in MySQL

C. 0 in both

D. An error in PostgreSQL

ANSWER KEY · SQL PITFALLS — THE TWELVE MISTAKES THAT FAIL INTERVIEW QUERIES

1. B Any comparison with NULL is unknown, and WHERE keeps only true. Write `IS NULL` (or MySQL's NULL-safe `<=>`).

2. B The WHERE turns the outer join into an inner join. Move `o.status = 'paid'` into the ON clause to keep every customer with a count of 0.

3. B Aggregates skip NULLs, so the average is (80 + 100) / 2. If a missing score should count as 0, write `AVG(COALESCE(score, 0))`.

4. C The half-open range includes the whole of March 31 and is sargable. BETWEEN stops at 31 March 00:00:00; the function-wrapped forms read every row, and MONTH() matches March of every year.

5. B PostgreSQL truncates integer division, so the order of operations decides whether a percentage survives. MySQL's `/` always returns a decimal. Writing `100.0 * …` is safe everywhere.

CHECKPOINT · 30 MIN

Final checkpoint — Interview idioms

The last checkpoint of the SQL plan. It covers the classic interview questions, the theory drill, the idiom sheet and the pitfalls — and, being the last one, it reaches back across the whole track: filtering, NULLs, grouping, joins, subqueries, window functions, data changes, keys and transactions.

How it works. Twelve questions and three queries; 70% on the questions and every query accepted clears the module. With every other module complete, that also completes the plan.

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

- The second highest salary two ways, and how each returns NULL when there is none.
- Why "employees earning more than their manager" is a self-join, and what happens to the employee with no manager.
- Why `NOT IN` returns nothing when the subquery holds a NULL, and the two safe alternatives.
- WHERE vs HAVING, DELETE vs TRUNCATE vs DROP, UNION vs UNION ALL — one sentence each, plus one detail.
- The logical order of a query's clauses, and why an alias cannot be used in WHERE.
- ROW_NUMBER, RANK and DENSE_RANK on the values 100, 90, 90, 80.
- Why a join to two child tables inflates sums, and how pre-aggregating fixes it.
- What each ACID property guarantees, and the leftmost-prefix rule of a composite index.

The three queries are the classics with a twist: the highest earners of each department with ties kept, the customers who placed no order in 2024 (the anti-join with its condition in the right place), and the duplicate emails once case and stray spaces are ignored.

EXERCISE 1

Highest earners per department, ties kept

`departments(id, name)` and `employees(id, name, salary, dept_id)`:

department	employees (salary)
Engineering	Bo 120, Asha 120, Cara 90
Ops	Dev 70
Sales	Eli 80, Fay 85
HR	—

List the **highest-paid employee(s) of each department** — every employee tied for the top salary. Return `department`, `employee` and `salary`, ordered by `department`, then `employee`.

```
department | employee | salary
Engineering | Asha | 120
Engineering | Bo | 120
Ops | Dev | 70
Sales | Fay | 85
```

INPUT 1

```
CREATE TABLE departments (id INT PRIMARY
KEY, name VARCHAR(20));
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(20), salary INT,
dept_id INT);
INSERT INTO departments VALUES (1,
'Engineering'), (2, 'Ops'), (3,
'Sales'), (4, 'HR');
INSERT INTO employees VALUES
  (1, 'Bo', 120, 1), (2, 'Asha', 120,
1), (3, 'Cara', 90, 1),
  (4, 'Dev', 70, 2),
  (5, 'Eli', 80, 3), (6, 'Fay', 85, 3);
```

OUTPUT 1

```
department | employee | salary
Engineering | Asha | 120
Engineering | Bo | 120
Ops | Dev | 70
Sales | Fay | 85
```

REFERENCE SOLUTION

```
SELECT d.name AS department, e.name AS employee, e.salary
FROM (
  SELECT emp.*,
         RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS rk
  FROM employees AS emp
) AS e
JOIN departments AS d ON d.id = e.dept_id
WHERE e.rk = 1
ORDER BY department, employee;
```

EXERCISE 2

No orders in 2024

`customers(id, name)` and `orders(id, customer_id, ordered_on)`:

customer	order dates
Asha	2023-12-31, 2025-01-01
Ben	2024-06-01
Cara	—
Dev	2024-01-01
Eli	2024-12-31, 2023-05-05

The starter tries an anti-join but returns nothing. Fix it: list the customers who placed **no order dated in 2024** (including those who never ordered at all). One column, `name`, ordered by `name`.

```
name
Asha
Cara
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(20));
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, ordered_on DATE);
INSERT INTO customers VALUES (1,
'Asha'), (2, 'Ben'), (3, 'Cara'), (4,
'Dev'), (5, 'Eli');
INSERT INTO orders VALUES
  (1, 1, '2023-12-31'), (2, 1, '2025-01-
01'),
  (3, 2, '2024-06-01'),
  (4, 4, '2024-01-01'),
  (5, 5, '2024-12-31'), (6, 5, '2023-05-
05');
```

OUTPUT 1

```
name
Asha
Cara
```

REFERENCE SOLUTION

```sql
SELECT c.name
FROM customers AS c
LEFT JOIN orders AS o
  ON  o.customer_id = c.id
  AND o.ordered_on ⩾ '2024-01-01'
  AND o.ordered_on <  '2025-01-01'
WHERE o.id IS NULL
ORDER BY c.name;
```

EXERCISE 3

Duplicate emails, any case

`users(id, email)` was filled by hand, so the same address appears with different capitals and stray spaces.

id	email
1	a@x.com
2	A@x.com
3	␣a@x.com
4	b@x.com
5	c@x.com
6	C@X.COM
7	d@x.com␣

(␣ marks a space.) Report every address that appears **more than once** once case and surrounding spaces are ignored: the columns `email` (lower-cased and trimmed) and `copies`, ordered by `copies` descending, then `email`.

```
email | copies
a@x.com | 3
c@x.com | 2
```

INPUT 1

```
CREATE TABLE users (id INT PRIMARY KEY,
email VARCHAR(60));
INSERT INTO users VALUES
  (1, 'a@x.com'), (2, 'A@x.com'), (3, '
a@x.com'), (4, 'b@x.com'),
  (5, 'c@x.com'), (6, 'C@X.COM'), (7,
'd@x.com ');
```

OUTPUT 1

```
email | copies
a@x.com | 3
c@x.com | 2
```

REFERENCE SOLUTION

```sql
SELECT LOWER(TRIM(email)) AS email, COUNT(*) AS copies
FROM users
GROUP BY LOWER(TRIM(email))
HAVING COUNT(*) > 1
ORDER BY copies DESC, email;
```

CHECKPOINT QUESTIONS

1. Which query returns NULL — not an empty result — when there is no second highest salary?

A. `SELECT DISTINCT salary FROM e ORDER BY salary DESC LIMIT 1 OFFSET 1`

B. `SELECT MAX(salary) FROM e WHERE salary < (SELECT MAX(salary) FROM e)`

C. `SELECT salary FROM e ORDER BY salary DESC LIMIT 2`

D. `SELECT salary FROM e WHERE salary = 2`

2. Where should `COUNT(*) ⩾ 2` go in a query that groups orders by customer?

A. WHERE

B. HAVING

C. ON

D. ORDER BY

3. For the points 50, 40, 40, 30 (highest first), what does `RANK()` give?

A. 1, 2, 3, 4

B. 1, 2, 2, 3

C. 1, 2, 2, 4

D. 1, 1, 2, 3

4. Which is true of `UNION` compared with `UNION ALL`?

A. UNION keeps duplicates; UNION ALL removes them

B. UNION removes duplicate rows from the combined result, which costs a sort or hash; UNION ALL keeps every row

C. They are identical in MySQL

D. UNION ALL requires the same column names

5. Why can `SELECT name AS n FROM t WHERE n = 'Ana'` fail?

A. Aliases must be quoted

B. WHERE is evaluated before the SELECT list, so the alias does not exist yet

C. `n` is a reserved word

D. Aliases only work with GROUP BY

6. `orders` has 2 rows for a customer and `payments` has 3. After `customers LEFT JOIN orders LEFT JOIN payments`, how many rows does that customer have?

A. 2

B. 3

C. 5

D. 6

7. `complaints.customer_id` contains a NULL. What does `WHERE id NOT IN (SELECT customer_id FROM complaints)` return?

A. Customers without a complaint

B. No rows

C. Every customer

D. An error

8. Which statement removes every row of a table but cannot be rolled back in MySQL, and resets AUTO_INCREMENT?

A. `DELETE FROM t`

B. `TRUNCATE TABLE t`

C. `DROP TABLE t`

D. `DELETE FROM t WHERE 1 = 1`

9. Which isolation level is InnoDB's default?

A. READ UNCOMMITTED

B. READ COMMITTED

C. REPEATABLE READ

D. SERIALIZABLE

10. A table `employees(id, dept_id, dept_name)` with key `id` stores the department's name in every row. Which normal form does it break?

A. 1NF

B. 2NF

C. 3NF — `dept_name` depends on `dept_id`, a non-key column

D. None

11. Why is `WHERE YEAR(created_at) = 2024` slower than `WHERE created_at ⩾ '2024-01-01' AND created_at < '2025-01-01'` on an indexed column?

A. YEAR() is not a MySQL function

B. Wrapping the column in a function makes the condition non-sargable, so the index cannot be used to seek

C. The range form returns fewer rows

D. There is no difference

12. Which ranking function should a "top three earners per team, exactly three rows each" question use?

A. `RANK()`

B. `DENSE_RANK()`

C. `ROW_NUMBER()` with a tie-breaker in its ORDER BY

D. `NTILE(3)`

ANSWER KEY · FINAL CHECKPOINT — INTERVIEW IDIOMS

1. B An aggregate without GROUP BY always returns one row, NULL when nothing qualifies. The LIMIT/OFFSET form needs a scalar-subquery wrapper to do the same.

2. B It tests a group, which exists only after GROUP BY. WHERE filters individual rows before grouping and cannot contain aggregates.

3. C RANK shares the tie at 2 and skips 3. DENSE_RANK would give 1, 2, 2, 3; ROW_NUMBER 1, 2, 3, 4.

4. B Prefer UNION ALL unless removing duplicates is the point — it is faster and never merges two genuine facts into one.

5. B The logical order is FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY. ORDER BY can use the alias; WHERE cannot.

6. D Each order row pairs with each payment row: 2 × 3 = 6. That fan-out is why sums over the result are inflated — aggregate each child first.

7. B `id <> NULL` is unknown, so the NOT IN can never be true. Use NOT EXISTS or a LEFT JOIN with IS NULL.

8. B TRUNCATE is DDL in MySQL: it recreates the table and commits implicitly. DELETE is transactional and keeps the counter; DROP removes the table itself.

9. C MySQL InnoDB defaults to REPEATABLE READ; PostgreSQL and SQL Server default to READ COMMITTED.

10. C A transitive dependency (id → dept_id → dept_name) breaks 3NF. The key is a single column, so there is no partial dependency for 2NF to forbid.

11. B The index is sorted by `created_at`, not by `YEAR(created_at)`. A range on the bare column lets the optimiser seek to the first matching row.

12. C Only ROW_NUMBER guarantees exactly N rows; a unique tie-breaker makes the choice between tied rows deterministic. RANK and DENSE_RANK can return more rows when there are ties.

CodeKairo · SQL study plan · Module 16: Interview idioms. Solve the exercises online to track your progress and earn XP.