Module 8 of 16

Subqueries

Scalar subqueries and the second-highest salary, IN and EXISTS with multi-column IN and ANY/ALL, correlated subqueries for per-group comparisons and the latest row per group, derived tables that aggregate before joining, and why NOT IN returns nothing when a NULL appears.

A subquery is a query inside another query, and it is how SQL answers questions that need a computed value, list or table before the main question can be asked: who earns more than the average, which customers bought a book, which order is each customer's latest, which customers spend more than the average customer. MySQL 8 accepts a subquery almost anywhere an expression or a table can go, and the optimizer turns many of them into joins — but the meaning, the NULL behaviour and the errors are yours to get right, and interviews test exactly those.

Scalar subqueries covers one-value subqueries in `WHERE`, the select list and `HAVING`, NULL for no rows, MySQL's "Subquery returns more than 1 row", and the second-highest-salary problem. IN and EXISTS covers membership and existence tests, chaining them across tables, "both A and B" questions, multi-column `IN`, and `ANY` / `ALL`. Correlated subqueries covers subqueries that read the outer row: per-department comparisons, the latest row per group with its ties, per-row counts in the select list, how MySQL runs them and the scope rule that silently correlates a typo. Derived tables covers subqueries in `FROM`, the alias rule, aggregates of aggregates, aggregating each child before joining, and `LATERAL`. NOT IN and NULL covers three-valued logic, the outer-side NULL, the safe anti joins and the tree-node classification problem.

Every exercise is a query judged on small datasets with the rows that break careless solutions — NULL keys, NULL values, duplicate rows, ties and empty tables. The checkpoint adds products above their category's average, customers with no order in a year, and the top spender of each city.

CONTENTS

1	Scalar subqueries — a query that returns one value	15 min
2	IN and EXISTS — testing membership with a subquery	15 min
3	Correlated subqueries — a subquery per row	16 min
4	Derived tables — a subquery in FROM	16 min

5	NOT IN and NULL — the subquery trap	15 min
✓	Checkpoint — Subqueries	25 min

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

LESSON 1 · 15 MIN

Scalar subqueries — a query that returns one value

Some questions need a number computed from the whole table before any single row can be judged. Who earns more than the average? What share of total revenue is each product? Which order was the largest? `WHERE salary > AVG(salary)` looks right but is illegal — `WHERE` filters rows before any aggregate exists. The fix is to compute the number in its own query and drop its result into the outer one. A query nested inside another is a **subquery**; one that returns a single value is a **scalar subquery**, and it is the simplest kind. This lesson covers where scalar subqueries may appear, what happens when they return zero or several rows, and the classic "second highest" problem.

The data

```
CREATE TABLE employees (id INT PRIMARY KEY, name VARCHAR(40), dept VARCHAR(20)
INSERT INTO employees VALUES
  (1, 'Asha',  'Eng',   120000),
  (2, 'Ben',   'Eng',    95000),
  (3, 'Chen',  'Sales',  70000),
  (4, 'Dina',  'Sales',  82000),
  (5, 'Esha',  'Ops',    60000);
```

The average salary is 427000 / 5 = 85400.

A subquery in WHERE

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees)
ORDER BY salary DESC, name;
```

name	salary
Asha	120000
Ben	95000

The parenthesised query runs once, produces the single value 85400, and the outer `WHERE` compares every row with it. A scalar subquery is just an expression — the database treats `(SELECT AVG(salary) FROM employees)` the way it would treat the literal `85400`, except that it stays right when the data changes.

The inner query is **independent** (non-correlated): it does not mention the outer row, so MySQL evaluates it once and caches the value. The next lessons cover subqueries that do depend on the outer row.

A subquery in the select list

```
SELECT name, salary,
       salary - (SELECT AVG(salary) FROM employees) AS vs_average,
       ROUND(100 * salary / (SELECT SUM(salary) FROM employees), 1) AS pct_of_
FROM employees
ORDER BY salary DESC, name;
```

name	salary	vs_average	pct_of_payroll
Asha	120000	34600	28.1
Ben	95000	9600	22.2
Dina	82000	-3400	19.2
Chen	70000	-15400	16.4
Esha	60000	-25400	14.1

Each scalar subquery is used inside arithmetic like any number. This "share of the total" shape is common in reports. (Window functions, later in the course, write it as `SUM(salary) OVER ()` without a second read of the table.)

A subquery in HAVING

Departments whose average salary is above the company average compare a group's aggregate with the table's aggregate:

```sql
SELECT dept, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept
HAVING AVG(salary) > (SELECT AVG(salary) FROM employees)
ORDER BY dept;
```

dept	avg_salary
Eng	107500

`HAVING` filters groups after aggregation, and the subquery provides the threshold.

Zero rows: NULL

If a scalar subquery returns no rows, its value is `NULL`:

```sql
SELECT (SELECT salary FROM employees WHERE dept = 'Legal') AS legal_salary;
```

legal_salary
NULL

And `salary > NULL` is unknown, so a `WHERE` comparing against an empty subquery keeps nothing. Usually that is the right behaviour; when it is not, wrap the subquery in `COALESCE(…, 0)`.

More than one row: an error

```sql
SELECT name FROM employees
WHERE salary = (SELECT salary FROM employees WHERE dept = 'Eng');
-- MySQL: ERROR 1242 (21000): Subquery returns more than 1 row
```

Eng has two salaries, and `=` compares with one value. MySQL raises the error when it happens, at run time — so a query that returned one row during testing can start failing in production when a second matching row appears. Make a scalar subquery scalar *by construction*: an aggregate (`MAX`, `AVG`, `COUNT`), a filter on a unique key, or `LIMIT 1` with an `ORDER BY`. If a list was meant, use `IN` — the next lesson.

Not every engine is this strict: SQLite, including the practice engine these exercises run on, quietly uses the first row instead of failing. Write scalar subqueries so that the question never arises; that is what interviews and MySQL expect.

The second highest salary

A favourite interview question: return the second highest *distinct* salary, or NULL if there is none.

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

second_highest
95000

The subquery finds the top salary; the outer query takes the largest salary below it. Duplicates of the top are excluded automatically because they are not *less than* it. If every employee earns the same, no row passes the `WHERE`, and `MAX` of nothing is `NULL` — an aggregate without `GROUP BY` always returns exactly one row. That NULL row is part of the expected answer, and it is why the `MAX` form beats a plain `ORDER BY salary DESC LIMIT 1 OFFSET 1`, which returns *no row* when there is no second salary and returns a duplicate of the top when the top is tied.

The `LIMIT` version can be made correct by wrapping it as a scalar subquery and de-duplicating:

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

An outer `SELECT` with no `FROM` returns one row, and the empty subquery becomes NULL. This version generalises to the *n*th highest by changing the offset; the `MAX … <` version does not.

Subqueries and readability

A scalar subquery is evaluated once when it is independent, so performance is rarely a concern. The cost is readability: three nested levels of parentheses are hard to follow. Indent the inner query, alias every computed column, and when the same subquery appears twice consider a CTE (a later module), which names it once.

In other databases

PostgreSQL and SQL Server behave like MySQL: an error for more than one row ("more than one row returned by a subquery used as an expression" in PostgreSQL), NULL for none. SQL Server writes the `LIMIT … OFFSET` form as `OFFSET 1 ROWS FETCH NEXT 1 ROWS ONLY`, or `TOP`.

Pitfalls

- Writing an aggregate directly in `WHERE` instead of a subquery.
- A "scalar" subquery that can return several rows.
- Forgetting that an empty subquery is NULL and that comparisons with NULL keep nothing.
- `LIMIT 1 OFFSET 1` without `DISTINCT`, or without wrapping, for "second highest".
- Integer division surprises in other databases when computing shares (`salary / total` is 0 in SQL Server for integers; MySQL's `/` is decimal).

Key takeaways

- A scalar subquery returns one row and one column and can be used wherever a value can.
- Use it for comparisons with a table-wide aggregate, shares of a total, and thresholds in `HAVING`.
- No rows → NULL; more than one row → MySQL error 1242.

- Second highest: `MAX(salary) WHERE salary < (SELECT MAX(salary) …)`, NULL when absent.
- Independent subqueries run once; keep them readable with indentation and aliases.

EXERCISE 1

Above the company average

List the employees whose salary is above the average salary of all employees, with how far above it they are. Employees with a NULL salary are not counted in the average (that is how `AVG` behaves) and are never listed.

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

id	name	dept	salary
1	Asha	Eng	120000
2	Ben	Eng	95000
3	Chen	Sales	70000
4	Dina	Sales	82000
5	Esha	Ops	60000

Output: columns `name`, `salary`, `above_by` (salary minus the average, `ROUND`ed to 2 places), ordered by `salary` descending, then `name`.

```
name | salary | above_by
Asha | 120000 | 34600
Ben | 95000 | 9600
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), dept VARCHAR(20),
salary INT);
INSERT INTO employees VALUES (1, 'Asha',
'Eng', 120000), (2, 'Ben', 'Eng',
95000), (3, 'Chen', 'Sales', 70000), (4,
'Dina', 'Sales', 82000), (5, 'Esha',
'Ops', 60000);
```

OUTPUT 1

```
name | salary | above_by
Asha | 120000 | 34600
Ben | 95000 | 9600
```

REFERENCE SOLUTION

```sql
SELECT name, salary, ROUND(salary - (SELECT AVG(salary) FROM employees), 2) AS above
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees)
ORDER BY salary DESC, name;
```

EXERCISE 2

Second highest salary

Return the second highest **distinct** salary as a single row. If there is no second distinct salary (one distinct salary, or no employees at all), return one row holding NULL.

Table: `employees(id, name, salary)`.

id	name	salary
1	Asha	120000
2	Ben	95000
3	Chen	70000
4	Dina	82000
5	Esha	60000

Output: one column `second_highest`, one row.

```
second_highest
95000
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), salary INT);
INSERT INTO employees VALUES (1, 'Asha',
120000), (2, 'Ben', 95000), (3, 'Chen',
70000), (4, 'Dina', 82000), (5, 'Esha',
60000);
```

OUTPUT 1

```
second_highest
95000
```

REFERENCE SOLUTION

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

QUIZ

1. Why does `SELECT name FROM employees WHERE salary > AVG(salary);` fail?

A. `AVG` needs a `GROUP BY`

B. `WHERE` filters rows before aggregation, so an aggregate cannot appear there; compute it in a subquery

C. `AVG` cannot be compared with an integer

D. `salary` must be qualified

2. `SELECT name FROM employees WHERE salary = (SELECT salary FROM employees WHERE dept = 'Eng');` — Eng has two employees. What does MySQL do?

A. Compares with the first Eng salary

B. Raises error 1242: Subquery returns more than 1 row

C. Treats `=` as `IN`

D. Returns no rows

3. Every employee earns 50000. What does this return?

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

A. No rows

B. One row: NULL

C. One row: 50000

D. An error

4. What is the value of `(SELECT salary FROM employees WHERE id = 999)` when no employee has id 999?

A. 0

B. NULL

C. An empty string

D. An error

ANSWER KEY · SCALAR SUBQUERIES — A QUERY THAT RETURNS ONE VALUE

1. B MySQL reports *Invalid use of group function*. `WHERE` sees one row at a time, before any aggregate is computed. `WHERE salary > (SELECT AVG(salary) FROM employees)` computes the average first. `HAVING` can use aggregates, but it filters groups, not individual employees.

2. B A subquery used with `=` must be scalar, and MySQL checks at run time. Use `IN` for a list, or make the subquery return one value (`MAX` , a unique key, `LIMIT 1`). Some engines, SQLite among them, quietly take the first row — do not rely on it.

3. B No row passes `salary < 50000` , and an aggregate without `GROUP BY` over no rows returns one row whose `MAX` is NULL. That is exactly the expected "no second highest" answer.

4. B A scalar subquery that finds no rows evaluates to NULL. Wrap it in `COALESCE(…, 0)` if a default is needed.

LESSON 2 · 15 MIN

IN and EXISTS — testing membership with a subquery

The previous module's semi-join lesson used `EXISTS` and `IN` to ask "does this row have a match?". This lesson looks at them as subquery operators in their own right. `IN` asks whether a value belongs to a *set of values* produced by a query; `EXISTS` asks whether a query produces *any row at all*. Between them they answer most "which rows relate to something over there?" questions — customers who bought from a category, students enrolled in two particular courses, products in a promotion list — and they keep each outer row exactly once, which is what separates them from a join. The quantified comparisons `ANY` and `ALL`, which interviews sometimes ask about, round the lesson off.

IN with a list and with a subquery

`IN` with a literal list is familiar from filtering:

```
SELECT name FROM products WHERE category IN ('Books', 'Music');
```

Replace the list with a query and the list is computed:

```
SELECT c.id, c.name
FROM customers c
WHERE c.id IN (SELECT o.customer_id FROM orders o WHERE o.amount ⩾ 300)
ORDER BY c.id;
```

The subquery must return exactly **one column**. It may return any number of rows, including zero (then `IN` is false for every row) and duplicates (which change nothing — membership is membership). Each customer is kept once, no matter how many qualifying orders match.

`IN (subquery)` is the natural choice when the link between the tables is a single column and the subquery reads as a self-contained list: "customers whose id is among the ids of big orders".

Chaining IN through several tables

Membership tests nest. Customers who bought anything in the Books category, through orders, order items and products:

```sql
SELECT c.id, c.name
FROM customers c
WHERE c.id IN (
  SELECT o.customer_id
  FROM orders o
  WHERE o.id IN (
    SELECT i.order_id
    FROM order_items i
    WHERE i.product_id IN (SELECT p.id FROM products p WHERE p.category = 'Boo
  )
)
ORDER BY c.id;
```

Read it from the inside out: the ids of Books products → the orders containing them → the customers who placed those orders. The same question as a join needs `DISTINCT`, because a customer with two Books items in three orders would appear six times. Deep nesting gets hard to read; a single semi-join with joins inside is often clearer:

```sql
WHERE c.id IN (
  SELECT o.customer_id
  FROM orders o
  JOIN order_items i ON i.order_id = o.id
  JOIN products p    ON p.id = i.product_id
  WHERE p.category = 'Books'
)
```

The duplicates the inner join produces do not matter — `IN` only asks about membership.

"Both" and "either" questions

Students enrolled in **both** SQL and Python:

```sql
SELECT s.name
FROM students s
WHERE s.id IN (SELECT e.student_id FROM enrollments e WHERE e.course = 'SQL')
  AND s.id IN (SELECT e.student_id FROM enrollments e WHERE e.course = 'Python
ORDER BY s.name;
```

A beginner writes `WHERE e.course = 'SQL' AND e.course = 'Python'` on a join — which is never true, because a single enrollment row holds one course. Two independent membership tests, one per condition, express "both". "Either" is `OR` between them (or a single `IN` with `course IN ('SQL', 'Python')`), and "SQL but not Python" is `IN … AND NOT IN …` — with the NULL caution of the last lesson in this module.

Another way to say "both" is grouping: `GROUP BY student_id HAVING COUNT(DISTINCT course) = 2` over the enrollments filtered to the two courses. It scales better to "all five of these courses".

EXISTS

```sql
SELECT d.name
FROM departments d
WHERE EXISTS (
  SELECT 1 FROM employees e
  WHERE e.dept_id = d.id AND e.salary ⩾ 100000
)
ORDER BY d.name;
```

`EXISTS` is true when the subquery returns at least one row; the select list inside does not matter. Because a subquery that does not mention the outer row returns the same thing for every outer row, an `EXISTS` is almost always **correlated** — `e.dept_id = d.id` ties it to the current department. The next lesson explains correlated subqueries properly; here it is enough to read the query as "departments for which some high earner exists".

`EXISTS` is the better fit when:

- the link involves several columns, or extra conditions on the matching row;
- the question is about existence rather than about a list of values;
- NULLs may appear in the linking column (`NOT EXISTS` is safe where `NOT IN` is not).

Multi-column IN

MySQL supports comparing a tuple with a subquery's rows:

```sql
-- each customer's most recent order(s)
SELECT o.*
FROM orders o
WHERE (o.customer_id, o.order_date) IN (
  SELECT customer_id, MAX(order_date) FROM orders GROUP BY customer_id
)
ORDER BY o.customer_id, o.id;
```

The pair on the left must match a row of the subquery on both columns at once. PostgreSQL supports it too; SQL Server does not, and uses `EXISTS` with both conditions instead.

ANY, SOME and ALL

MySQL also offers quantified comparisons against a subquery:

```sql
-- earns more than everyone in Sales
SELECT name FROM employees
WHERE salary > ALL (SELECT salary FROM employees WHERE dept = 'Sales');

-- earns more than at least one person in Sales
SELECT name FROM employees
WHERE salary > ANY (SELECT salary FROM employees WHERE dept = 'Sales');
```

`SOME` is a synonym for `ANY`. `= ANY (…)` is exactly `IN (…)`, and `<> ALL (…)` is exactly `NOT IN (…)` — including its NULL problem. `> ALL` is usually written `> (SELECT MAX(…))` and `> ANY` as `> (SELECT MIN(…))`, which is clearer and works everywhere, but the two forms differ at the edges: `> ALL` of an **empty** subquery is true (there is no value to fail against), while `> (SELECT MAX(…))` of an empty set compares with NULL and is never true. Interviewers occasionally probe exactly that. SQLite has no `ANY` / `ALL`, so the exercises here use `MIN` / `MAX`.

IN versus a join, in short

Need	Use
Columns from both tables	`JOIN`
Rows of A that have a match in B, each once	`IN` or `EXISTS`
Rows of A with no match in B	`NOT EXISTS`
A value compared with every value of a set	`> ALL`, or `> (SELECT MAX …)`

Pitfalls

- A subquery after `IN` that returns two columns (error: "Operand should contain 1 column").
- Testing two values of one column on a single row ("both" questions).
- Using a join plus `DISTINCT` where a membership test was meant.
- `<> ALL` and `NOT IN` against a nullable column.
- Assuming `> ALL (empty)` behaves like `> MAX(empty)`.

Key takeaways

- `IN (subquery)` tests a value against a one-column list; duplicates in the list do not matter.
- `EXISTS (subquery)` tests whether any row exists, usually correlated to the outer row.
- Both keep each outer row once; MySQL 8 plans both as semi joins.
- "Both A and B" is two membership tests, not one row with two values.
- `= ANY` is `IN`, `<> ALL` is `NOT IN`, and `> ALL (empty)` is true.

EXERCISE 1

Customers who bought a book

List the customers who bought at least one product in the category `'Books'`, each once.

Tables: `customers(id, name)`, `orders(id, customer_id)`, `order_items(order_id, product_id)`, `products(id, name, category)`.

customers.id	name
1	Asha
2	Ben
3	Chen
4	Dina

products.id	name	category
1	SQL Guide	Books
2	Chess	Games
3	Novel	Books

orders.id	customer_id
10	1
11	1
12	2
13	3
14	4

order_id	product_id
10	1
10	3
11	1
12	2
13	3
14	2

Output: columns `id`, `name`, ordered by `name`, then `id`.

```
id | name
1 | Asha
3 | Chen
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE products (id INT PRIMARY
KEY, name VARCHAR(40), category
VARCHAR(20));
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT);
CREATE TABLE order_items (order_id INT,
product_id INT);
INSERT INTO customers VALUES (1,
'Asha'), (2, 'Ben'), (3, 'Chen'), (4,
'Dina');
INSERT INTO products VALUES (1, 'SQL
Guide', 'Books'), (2, 'Chess', 'Games'),
(3, 'Novel', 'Books');
INSERT INTO orders VALUES (10, 1), (11,
1), (12, 2), (13, 3), (14, 4);
INSERT INTO order_items VALUES (10, 1),
(10, 3), (11, 1), (12, 2), (13, 3), (14,
2);
```

OUTPUT 1

```
id | name
1 | Asha
3 | Chen
```

REFERENCE SOLUTION

```sql
SELECT c.id, c.name
FROM customers c
WHERE c.id IN (
  SELECT o.customer_id
  FROM orders o
  WHERE o.id IN (
    SELECT i.order_id
    FROM order_items i
    WHERE i.product_id IN (SELECT p.id FROM products p WHERE p.category = 'Books')
  )
)
ORDER BY c.name, c.id;
```

EXERCISE 2

Enrolled in both SQL and Python

List the students enrolled in **both** `'SQL'` and `'Python'`. The enrolment table may contain the same (student, course) row twice.

Tables: `students(id, name)` and `enrollments(student_id, course)`.

students.id	name
1	Anu
2	Bilal
3	Chitra
4	Dev

student_id	course
1	SQL
1	Python
2	SQL
3	Python
3	SQL
3	Java
4	Python

Output: one column `student`, ordered by `student`.

```
student
Anu
Chitra
```

INPUT 1

```
CREATE TABLE students (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE enrollments (student_id
INT, course VARCHAR(20));
INSERT INTO students VALUES (1, 'Anu'),
(2, 'Bilal'), (3, 'Chitra'), (4, 'Dev');
INSERT INTO enrollments VALUES (1,
'SQL'), (1, 'Python'), (2, 'SQL'), (3,
'Python'), (3, 'SQL'), (3, 'Java'), (4,
'Python');
```

OUTPUT 1

```
student
Anu
Chitra
```

REFERENCE SOLUTION

```sql
SELECT s.name AS student
FROM students s
WHERE s.id IN (SELECT e.student_id FROM enrollments e WHERE e.course = 'SQL')
  AND s.id IN (SELECT e.student_id FROM enrollments e WHERE e.course = 'Python')
ORDER BY student;
```

QUIZ

1. What happens with `WHERE id IN (SELECT customer_id, amount FROM orders)`?

A. It matches on either column

B. An error: the subquery after `IN` must return one column (MySQL: "Operand should contain 1 column(s)")

C. It matches on the first column only

D. It matches on both columns

2. In MySQL 8, which is generally true of `IN (subquery)` and `EXISTS`?

A. `EXISTS` is always much faster

B. `IN` is always much faster

C. The optimizer usually turns both into the same semi-join plan, so choose by readability

D. `IN` cannot use indexes

3. Which condition is equivalent to `x IN (SELECT y FROM t)`?

A. `x = ALL (SELECT y FROM t)`

B. `x = ANY (SELECT y FROM t)`

C. `x > ANY (SELECT y FROM t)`

D. `x <> ALL (SELECT y FROM t)`

4. `SELECT name FROM employees WHERE salary > ALL (SELECT salary FROM employees WHERE dept = 'Legal');` — there are no Legal employees. What is returned?

A. No rows

B. Every employee with a non-NULL salary

C. An error

D. Only the highest-paid employee

ANSWER KEY · IN AND EXISTS — TESTING MEMBERSHIP WITH A SUBQUERY

1. B A single value can only be compared with a one-column list. To match on two columns, put a row on the left too: `(id, amount) IN (SELECT customer_id, amount …)`.

2. C The "EXISTS is faster" rule comes from MySQL 5.5 and earlier, which ran `IN` subqueries row by row. MySQL 5.6+ and 8 apply semi-join strategies to both.

3. B `= ANY` is true when x equals at least one value — the definition of `IN`. `<> ALL` is `NOT IN`, and `= ALL` requires every value to equal x.

4. B `> ALL` of an empty set is true: there is no value to fail against. Compare `salary > (SELECT MAX(salary) … )`, which compares with NULL and returns no rows — the two forms differ exactly here.

LESSON 3 · 16 MIN

Correlated subqueries — a subquery per row

The subqueries so far were independent: `(SELECT AVG(salary) FROM employees)` gives the same answer whichever row is asking. Many questions need a different comparison for each row. Is this employee paid above *their department's* average? Is this order *its customer's* most recent? Does *this* product cost more than the average of *its* category? The inner query must know which outer row it is answering for, so it refers to the outer row's columns. That makes it a **correlated subquery**. This lesson covers how to write and read them, the per-group patterns they make easy, their use in the select list, how they perform, and a scope rule that produces some of the most baffling bugs in SQL.

The data

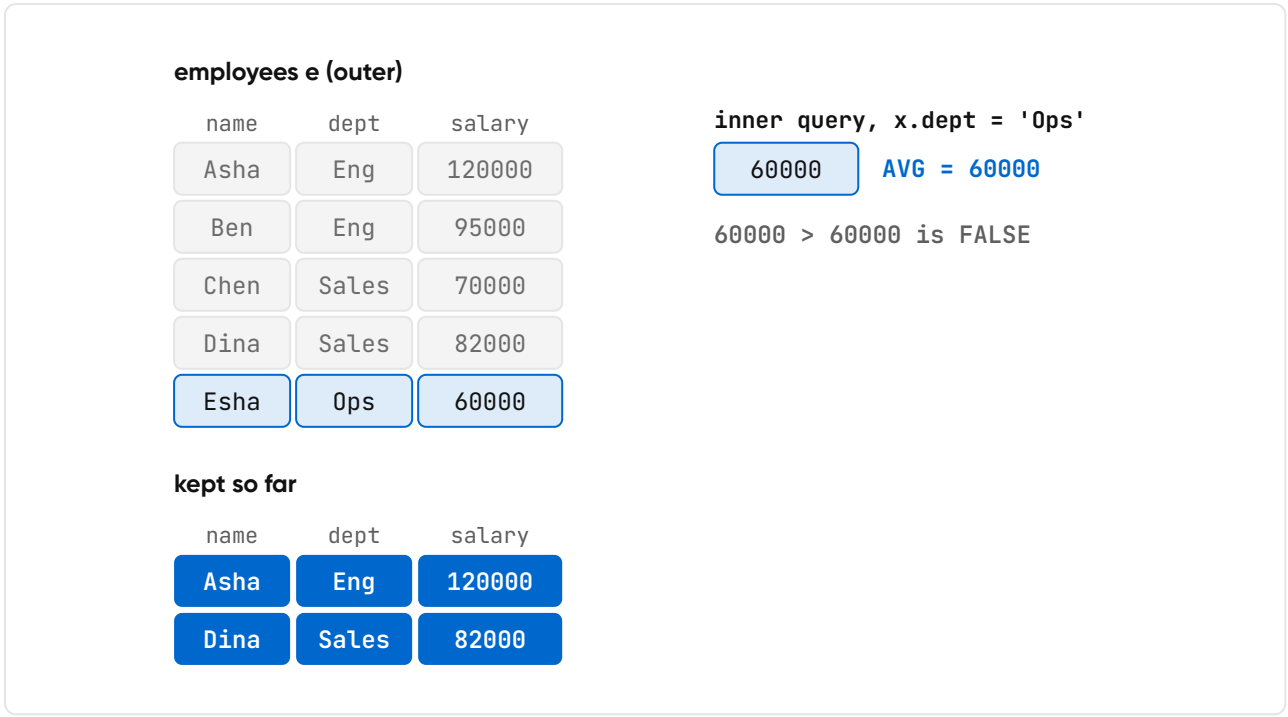
```
CREATE TABLE employees (id INT PRIMARY KEY, name VARCHAR(40), dept VARCHAR(20)
INSERT INTO employees VALUES
  (1, 'Asha', 'Eng',   120000),
  (2, 'Ben',  'Eng',    95000),
  (3, 'Chen', 'Sales',  70000),
  (4, 'Dina', 'Sales',  82000),
  (5, 'Esha', 'Ops',    60000);
```

Department averages: Eng 107500, Sales 76000, Ops 60000.

Above their own department's average

```
SELECT e.name, e.dept, e.salary
FROM employees e
WHERE e.salary > (
  SELECT AVG(x.salary)
  FROM employees x
  WHERE x.dept = e.dept
)
ORDER BY e.dept, e.salary DESC, e.name;
```

A correlated subquery runs once per outer row



1. The inner query mentions e.dept, a column of the outer row, so it cannot run once on its own. Conceptually it runs again for every employee, with that employee's department filled in.
2. For Asha the inner query averages the 2 Eng salaries: 107500. 120000 > 107500 is TRUE, so Asha is kept.
3. For Ben the inner query averages the 2 Eng salaries: 107500. 95000 > 107500 is FALSE, so Ben is left out.
4. For Chen the inner query averages the 2 Sales salaries: 76000. 70000 > 76000 is FALSE, so Chen is left out.
5. For Dina the inner query averages the 2 Sales salaries: 76000. 82000 > 76000 is TRUE, so Dina is kept.
6. Esha is alone in Ops, so the average is her own salary, 60000. 60000 > 60000 is FALSE: equal is not above, and she is left out.

The inner query mentions `e.dept`, a column of the **outer** row. Conceptually the database does this for every employee: take their department, compute that department's average, compare. For Asha it computes the Eng average (107500); for Chen the Sales average (76000). Esha is alone in Ops, equal to her own average, so `>` excludes her.

Two aliases of the same table appear — `e` outside and `x` inside — exactly as in a self join. They must be different, and every column should be qualified so it is obvious which copy it comes from.

The latest row per group

```sql
SELECT o.customer_id, o.id AS order_id, o.order_date, o.amount
FROM orders o
WHERE o.order_date = (
  SELECT MAX(x.order_date)
  FROM orders x
  WHERE x.customer_id = o.customer_id
)
ORDER BY o.customer_id, o.id;
```

For each order, the subquery finds the latest date among that customer's orders; the order is kept if it is dated on that day. This "greatest-n-per-group" pattern works for any extreme: the most expensive product per category, the first login per user, the highest score per student. Two properties to state in an interview:

- **ties are kept** — a customer with two orders on the latest date gets both rows. If exactly one is wanted, add a tie-breaker (`AND o.id = (SELECT MAX(x.id) … )`) or use `ROW_NUMBER()` from the window-functions module;
- customers with **no** orders do not appear, because there is no order row to start from.

Correlated subqueries in the select list

A correlated scalar subquery can compute a per-row value:

```sql
SELECT c.name,
       (SELECT COUNT(*) FROM orders o WHERE o.customer_id = c.id) AS orders,
       (SELECT MAX(o.order_date) FROM orders o WHERE o.customer_id = c.id) AS
FROM customers c
ORDER BY c.name;
```

Customers with no orders get 0 and NULL — `COUNT` over no rows is 0, `MAX` is NULL — so this is an alternative to a left join with `GROUP BY`, and it never suffers from fan-out because each subquery aggregates its own table. The cost is that each subquery is a separate lookup per customer; for a handful of per-row figures over an indexed column that is fine, for a dozen it is better to aggregate once in a derived table (next lesson).

EXISTS is usually correlated

```sql
-- departments where someone earns more than the department's head
SELECT d.name
FROM departments d
WHERE EXISTS (
  SELECT 1 FROM employees e
  WHERE e.dept_id = d.id AND e.salary > d.head_salary
);
```

An uncorrelated `EXISTS` would be true for all rows or none, which is rarely what anyone means. When you see `EXISTS`, look for the outer column inside it.

How it actually runs

"Once per outer row" is the logical model, not necessarily the plan. MySQL 8:

- converts `IN` / `EXISTS` subqueries into semi joins and `NOT EXISTS` into antijoins;
- can turn some scalar subqueries into joins with a derived table (the `subquery_to_derived` optimizer switch, off by default);
- otherwise runs the subquery as a `DEPENDENT SUBQUERY`, once per outer row.

What makes the per-row lookups cheap is an index on the correlated column — `orders(customer_id)` or `orders(customer_id, order_date)` for the latest-order query. Without one, each evaluation scans the table, and a 100,000-row outer table means 100,000 scans. If `EXPLAIN` shows `DEPENDENT SUBQUERY` on a large table with no usable index, rewrite as a join with a derived table of per-group aggregates.

The scope trap

Name resolution inside a subquery looks in the subquery's own tables first, then in the outer query's. That rule makes correlation possible — and it means a typo can silently correlate:

```sql
-- intended: customers who have a row in the `vip` table
SELECT name FROM customers
WHERE id IN (SELECT id FROM vip);   -- but vip's column is called customer_id
```

`vip` has no column `id`, so the inner `id` resolves to `customers.id`. The subquery becomes "for each vip row, the outer customer's id", and `id IN (id, id, …)` is true for every customer whenever `vip` is non-empty. No error, every row returned. Qualifying the inner column

(`SELECT v.id FROM vip v`) turns the mistake into the error it should be. Qualify every column inside a subquery — this is the single best habit for avoiding subquery bugs.

Correlated subquery or join?

Both can express most per-group comparisons. Guidelines:

- a per-row condition against an aggregate of related rows (above the department average, latest per customer) reads naturally as a correlated subquery;
- when the outer query also needs several columns from the related rows, a join is usually clearer;
- when the related table is large and the outer one too, aggregate once (derived table) and join.

Pitfalls

- Reusing the same alias inside and outside, so the "correlation" refers to the inner table itself.
- Unqualified inner columns silently binding to the outer query.
- Forgetting that latest-per-group returns ties.
- A correlated aggregate over a large table with no index on the correlated column.
- Expecting groups with no rows to appear when the outer query starts from the child table.

Key takeaways

- A correlated subquery refers to an outer column and is logically evaluated per outer row.
- It is the natural way to compare a row with an aggregate of its own group.
- `col = (SELECT MAX(col) … WHERE same group)` returns the latest row per group, ties included.
- Correlated scalar subqueries in the select list compute per-row counts without fan-out.
- Index the correlated column, and qualify every inner column to avoid accidental correlation.

EXERCISE 1

Above their department's average

List the employees who earn more than the average salary **of their own department**. Employees with no department (NULL `dept`) are not compared and never listed.

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

id	name	dept	salary
1	Asha	Eng	120000
2	Ben	Eng	95000
3	Farid	Eng	101000
4	Gopal	Eng	110000
5	Chen	Sales	70000
6	Dina	Sales	82000
7	Esha	Ops	60000

Output: columns `name`, `dept`, `salary`, ordered by `dept`, then `salary` descending, then `name`.

```
name | dept | salary
Asha | Eng | 120000
Gopal | Eng | 110000
Dina | Sales | 82000
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), dept VARCHAR(20),
salary INT);
INSERT INTO employees VALUES (1, 'Asha',
'Eng', 120000), (2, 'Ben', 'Eng',
95000), (3, 'Farid', 'Eng', 101000), (4,
'Gopal', 'Eng', 110000), (5, 'Chen',
'Sales', 70000), (6, 'Dina', 'Sales',
82000), (7, 'Esha', 'Ops', 60000);
```

OUTPUT 1

```
name | dept | salary
Asha | Eng | 120000
Gopal | Eng | 110000
Dina | Sales | 82000
```

REFERENCE SOLUTION

```sql
SELECT e.name, e.dept, e.salary
FROM employees e
WHERE e.salary > (
  SELECT AVG(x.salary)
  FROM employees x
  WHERE x.dept = e.dept
)
ORDER BY e.dept, e.salary DESC, e.name;
```

EXERCISE 2

Each customer's latest order

For each customer, list their most recent order. If several orders share the latest date, list them all. Customers with no orders are not listed, and orders without a known customer are ignored.

Tables: `customers(id, name)` and `orders(id, customer_id, order_date, amount)`.

customers.id	name
1	Asha
2	Ben
3	Chen

orders.id	customer_id	order_date	amount
10	1	2024-01-05	250
11	1	2024-02-11	100
12	2	2024-02-20	400
13	2	2024-02-20	50
14	2	2024-01-01	999

Output: columns `customer`, `order_id`, `order_date`, `amount`, ordered by `customer`, then `order_id`.

```
customer | order_id | order_date | amount
Asha | 11 | 2024-02-11 | 100
Ben | 12 | 2024-02-20 | 400
Ben | 13 | 2024-02-20 | 50
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, order_date DATE, amount
INT);
INSERT INTO customers VALUES (1,
'Asha'), (2, 'Ben'), (3, 'Chen');
INSERT INTO orders VALUES (10, 1, '2024-
01-05', 250), (11, 1, '2024-02-11',
100), (12, 2, '2024-02-20', 400), (13,
2, '2024-02-20', 50), (14, 2, '2024-01-
01', 999);
```

OUTPUT 1

```
customer | order_id | order_date |
amount
Asha | 11 | 2024-02-11 | 100
Ben | 12 | 2024-02-20 | 400
Ben | 13 | 2024-02-20 | 50
```

REFERENCE SOLUTION

```sql
SELECT c.name AS customer, o.id AS order_id, o.order_date, o.amount
FROM orders o
JOIN customers c ON c.id = o.customer_id
WHERE o.order_date = (
  SELECT MAX(x.order_date)
  FROM orders x
  WHERE x.customer_id = o.customer_id
)
ORDER BY customer, order_id;
```

QUIZ

1. What makes a subquery *correlated*?

A. It appears in the `WHERE` clause

B. It refers to a column of the outer query, so its result depends on the current outer row

C. It uses an aggregate

D. It joins two tables

2. `vip` has a column `customer_id` but no column `id`. What does this return?

```sql
SELECT name FROM customers
WHERE id IN (SELECT id FROM vip);
```

A. An error: unknown column `id`

B. Every customer, as long as `vip` has at least one row

C. Only the VIP customers

D. No rows

3. A "latest order per customer" query uses `WHERE o.order_date = (SELECT MAX(x.order_date) FROM orders x WHERE x.customer_id = o.customer_id)`. A customer has two orders on their latest date. What is returned for them?

A. The order with the lower id

B. Both orders

C. Neither

D. An error

4. Which index helps most for `WHERE o.order_date = (SELECT MAX(x.order_date) FROM orders x WHERE x.customer_id = o.customer_id)`?

A. An index on `orders(order_date)` alone

B. An index on `orders(customer_id, order_date)`

C. An index on `customers(name)`

D. No index can help a subquery

ANSWER KEY · CORRELATED SUBQUERIES — A SUBQUERY PER ROW

1. B An independent subquery gives the same answer for every row and can run once; a correlated one mentions an outer column (`e.dept`) and is logically evaluated per row. Where it appears and what it computes do not matter.

2. B The inner `id` is not found in `vip`, so it resolves to the outer `customers.id`. The subquery returns the current customer's own id once per vip row, and `id IN (id, …)` is true. Qualifying it (`SELECT v.id FROM vip v`) would have raised the error.

3. B Both rows satisfy the condition. To keep exactly one, add a tie-breaker (for example the highest id among them) or use `ROW_NUMBER()`.

4. B Each evaluation looks up one customer's orders and wants their maximum date; a composite index on `(customer_id, order_date)` answers that by reading the last entry for the customer.

LESSON 4 · 16 MIN

Derived tables — a subquery in FROM

A query's result is itself a table: rows and named columns. SQL lets you use that result wherever a table can go, including in `FROM` — a **derived table** (an *inline view* in Oracle's vocabulary). It is the tool for questions that need two steps of computation: aggregate, then compare the aggregates; summarise each child table, then join the summaries; rank, then filter on the rank. This lesson covers the syntax and its alias rule, the aggregate-before-join pattern that fixes fan-out, aggregates of aggregates, finding the top row per group by joining back, and where derived tables sit next to CTEs and `LATERAL`.

Syntax

```
SELECT t.customer_id, t.total
FROM (
  SELECT customer_id, SUM(amount) AS total
  FROM orders
  GROUP BY customer_id
) AS t
WHERE t.total ⩾ 500
ORDER BY t.total DESC, t.customer_id;
```

The inner query runs and produces a table with columns `customer_id` and `total`; the outer query treats it as a table named `t`. Three rules:

- **an alias is mandatory** in MySQL — without `AS t` you get *ERROR 1248: Every derived table must have its own alias*;
- every computed column needs a name (`SUM(amount) AS total`), or the outer query cannot refer to it sensibly;
- the inner `ORDER BY` is meaningless without a `LIMIT` — a table has no order, and MySQL ignores it. Order in the outer query.

This particular example could have used `HAVING SUM(amount) ⩾ 500`. Derived tables earn their place when one aggregation is not enough.

Aggregates of aggregates

"What is the average customer's total spend?" needs the total per customer first, then the average of those totals:

```
SELECT ROUND(AVG(t.total), 2) AS avg_customer_spend
FROM (
  SELECT customer_id, SUM(amount) AS total
  FROM orders
  GROUP BY customer_id
) AS t;
```

`AVG(SUM(amount))` is illegal — aggregates do not nest in one query level — so the inner level sums and the outer level averages. Note this is not the same number as the average order amount: a customer with ten small orders counts once here.

Combine it with a scalar subquery and you get "customers who spend more than the average customer":

```
SELECT c.name, t.total
FROM customers c
JOIN (
  SELECT customer_id, SUM(amount) AS total FROM orders GROUP BY customer_id
) AS t ON t.customer_id = c.id
WHERE t.total > (
  SELECT AVG(x.total)
  FROM (SELECT SUM(amount) AS total FROM orders GROUP BY customer_id) AS x
)
ORDER BY t.total DESC, c.name;
```

The per-customer totals appear twice; a CTE (a later module) would name them once. Joining a derived table works like joining any table — it needs an `ON` condition, and it can be inner or outer.

Aggregate before you join: the fan-out fix

The joins module showed the fan-out bug: joining customers to both orders and payments multiplies the rows and inflates both sums. The cure is to reduce each child to **one row per customer** before joining:

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

Each derived table has at most one row per customer, so each customer row meets at most one row from each side — no multiplication, no `DISTINCT` tricks. The left joins keep customers with no orders or no payments, and `COALESCE` reports their figures as 0. This is the standard shape for any report that combines several independent child tables: invoices and refunds, logins and purchases, commits and reviews.

Top row per group by joining back

Correlated subqueries found the latest order per customer. A derived table does the same with one aggregation and a join:

```sql
SELECT o.customer_id, o.id AS order_id, o.order_date
FROM orders o
JOIN (
  SELECT customer_id, MAX(order_date) AS last_date
  FROM orders
  GROUP BY customer_id
) AS m ON m.customer_id = o.customer_id AND m.last_date = o.order_date
ORDER BY o.customer_id, o.id;
```

The derived table holds one row per customer with their latest date; joining back on both columns picks the orders that sit on it (ties included, as before). On large tables this "group, then join back" form is often faster than the correlated version, because the aggregation runs once.

How MySQL runs a derived table

MySQL either **merges** a simple derived table into the outer query — as if you had written the joins directly — or **materialises** it into an internal temporary table, which it may index automatically for the join. Derived tables with `GROUP BY`, `DISTINCT`, `LIMIT` or aggregates are materialised. Since 8.0.22, MySQL can also push a condition from the outer `WHERE` down into a materialised derived table (`derived_condition_pushdown`), so filtering outside does not always mean aggregating everything first. In the past, unnecessary derived tables could be slow; in MySQL 8 the bigger concern is readability.

LATERAL derived tables

An ordinary derived table cannot refer to the other tables in the same `FROM`: it is computed on its own. MySQL 8.0.14 added `LATERAL`, which lifts that restriction — the derived table is evaluated per row of the tables before it, like a correlated subquery that returns rows:

```sql
-- each customer's three largest orders (MySQL 8.0.14+, PostgreSQL)
SELECT c.name, t.id, t.amount
FROM customers c
JOIN LATERAL (
  SELECT o.id, o.amount
  FROM orders o
  WHERE o.customer_id = c.id
  ORDER BY o.amount DESC, o.id
  LIMIT 3
) AS t ON TRUE;
```

SQL Server's equivalent is `CROSS APPLY` / `OUTER APPLY`. Window functions usually express the same "top n per group" more simply.

Pitfalls

- Missing alias on a derived table (MySQL error 1248).
- Unnamed computed columns inside the derived table.
- An `ORDER BY` inside a derived table expected to order the final result.
- Joining raw child tables when per-parent aggregates were needed (fan-out).
- An inner join to a derived table of aggregates silently dropping parents with no child rows.

Key takeaways

- A derived table is a subquery in `FROM`, used like a table; MySQL requires an alias.
- Use it for two-step questions: aggregate, then aggregate or filter the aggregates.
- Aggregate each child table per parent before joining — the clean fan-out fix.
- Group-then-join-back finds the top row per group, ties included.
- `LATERAL` (MySQL 8.0.14+) lets a derived table depend on earlier tables; CTEs name subqueries for reuse.

EXERCISE 1

Above the average customer

Find the customers whose total spend is greater than the **average customer's** total spend. A customer's total is the sum of their orders; only customers who exist and have at least one order take part (in the average and in the list). Orders of unknown customers are ignored entirely.

Tables: `customers(id, name)` and `orders(id, customer_id, amount)`.

customers.id	name
1	Asha
2	Ben
3	Chen
4	Dina

orders.id	customer_id	amount
10	1	250
11	1	100
12	2	400
13	3	50
14	3	30
15	9	600

Totals: Asha 350, Ben 400, Chen 80; the average customer spends 276.67.

Output: columns `customer`, `total`, ordered by `total` descending, then `customer`.

```
customer | total
Ben | 400
Asha | 350
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, amount INT);
INSERT INTO customers VALUES (1,
'Asha'), (2, 'Ben'), (3, 'Chen'), (4,
'Dina');
INSERT INTO orders VALUES (10, 1, 250),
(11, 1, 100), (12, 2, 400), (13, 3, 50),
(14, 3, 30), (15, 9, 600);
```

OUTPUT 1

```
customer | total
Ben | 400
Asha | 350
```

REFERENCE SOLUTION

```sql
SELECT c.name AS customer, t.total
FROM customers c
JOIN (
  SELECT customer_id, SUM(amount) AS total
  FROM orders
  GROUP BY customer_id
) AS t ON t.customer_id = c.id
WHERE t.total > (
  SELECT AVG(x.total)
  FROM (
    SELECT o.customer_id, SUM(o.amount) AS total
    FROM orders o
    JOIN customers k ON k.id = o.customer_id
    GROUP BY o.customer_id
  ) AS x
)
ORDER BY t.total DESC, customer;
```

EXERCISE 2

Orders and payments without double counting

For **every** customer, report the total they ordered, the total they paid, and what they still owe (`ordered - paid`). A customer with no orders or no payments shows `0` for that side. Joining both

tables directly would multiply the rows — aggregate first.

Tables: `customers(id, name)`, `orders(id, customer_id, amount)`, `payments(id, customer_id, amount)`.

customers.id	name
1	Asha
2	Ben
3	Chen

orders.id	customer_id	amount
1	1	100
2	1	150
3	1	50
4	2	400

payments.id	customer_id	amount
1	1	200
2	1	50
3	3	30

Output: columns `customer`, `ordered`, `paid`, `due`, ordered by `customer`, then `due`.

```
customer | ordered | paid | due
Asha | 300 | 250 | 50
Ben | 400 | 0 | 400
Chen | 0 | 30 | -30
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(40));
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, 'Chen');
INSERT INTO orders VALUES (1, 1, 100),
(2, 1, 150), (3, 1, 50), (4, 2, 400);
INSERT INTO payments VALUES (1, 1, 200),
(2, 1, 50), (3, 3, 30);
```

OUTPUT 1

```
customer | ordered | paid | due
Asha | 300 | 250 | 50
Ben | 400 | 0 | 400
Chen | 0 | 30 | -30
```

REFERENCE SOLUTION

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

QUIZ

1. What does MySQL say about `SELECT * FROM (SELECT customer_id, SUM(amount) AS total FROM orders GROUP BY customer_id);`?

A. It runs normally

B. Error 1248: Every derived table must have its own alias

C. Error: `SUM` is not allowed in a subquery

D. It ignores the subquery

2. How do you compute the average of each customer's total spend?

A. `SELECT AVG(SUM(amount)) FROM orders GROUP BY customer_id`

B. `SELECT AVG(t.total) FROM (SELECT customer_id, SUM(amount) AS total FROM orders GROUP BY customer_id) AS t`

C. `SELECT AVG(amount) FROM orders`

D. `SELECT SUM(amount) / COUNT(*) FROM orders`

3. An `ORDER BY total DESC` is written inside a derived table, and the outer query has none. What order does the final result have?

A. Descending by total

B. No guaranteed order — a derived table is a set, and MySQL may ignore an inner ORDER BY without LIMIT

C. Ascending by total

D. The order of the primary key

4. What does `LATERAL` (MySQL 8.0.14+) allow a derived table to do?

A. Have no alias

B. Refer to columns of tables listed before it in the same `FROM`, so it is evaluated per row of those tables

C. Be updated

D. Be used twice

ANSWER KEY · DERIVED TABLES — A SUBQUERY IN FROM

1. B A subquery in `FROM` needs a name: `… GROUP BY customer_id) AS t`. The aggregate inside is fine.

2. B Aggregates do not nest in one level, so the totals are computed in a derived table and averaged outside. `AVG(amount)` and `SUM/COUNT(*)` are the average *order*, which weights a customer by how many orders they placed.

3. B Only the outermost `ORDER BY` decides the result's order. An inner `ORDER BY` matters only together with `LIMIT` (for example, "the top three").

4. B An ordinary derived table is computed independently. A lateral one can say `WHERE o.customer_id = c.id`, for example to take each customer's top three orders. SQL Server spells it `CROSS APPLY`.

LESSON 5 · 15 MIN

NOT IN and NULL — the subquery trap

`NOT IN` reads like English — "customers whose id is not in the list of ordering customers" — and on clean test data it gives the right answer. Then one row with a NULL arrives in the column the subquery reads, and the query returns nothing at all, with no error. This is one of the most common SQL bugs in production and one of the most popular interview questions, because it tests whether you really understand NULL. This lesson works through why it happens, the related case of a NULL on the outer side, the three reliable alternatives, and a classic problem where the trap is built in.

The symptom

```sql
CREATE TABLE customers (id INT PRIMARY KEY, name VARCHAR(40));
CREATE TABLE orders (id INT PRIMARY KEY, customer_id INT);
INSERT INTO customers VALUES (1, 'Asha'), (2, 'Ben'), (3, 'Chen');
INSERT INTO orders VALUES (10, 1), (11, 2);

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

name
Chen

Correct: Chen never ordered. Now a guest checkout is stored with no customer:

```sql
INSERT INTO orders VALUES (12, NULL);
-- the same query now returns:
```

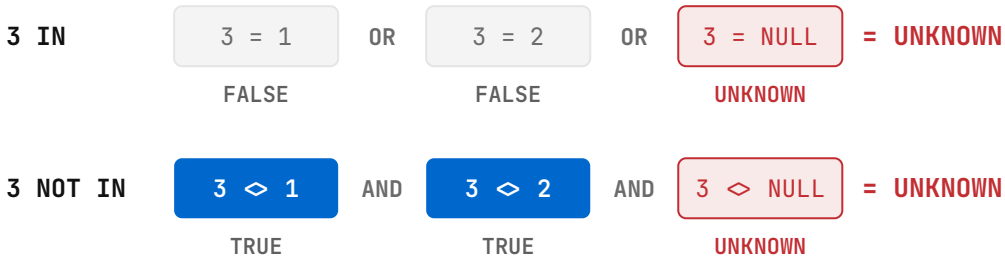
name
(no rows)

Why: three-valued logic

SQL comparisons have three outcomes: true, false and **unknown**. Any comparison with NULL — `3 = NULL`, `3 <> NULL`, even `NULL = NULL` — is unknown, because NULL means "no value known", and nobody can say whether an unknown value differs from 3.

`IN` and `NOT IN` against a list expand into comparisons:

Why NOT IN returns nothing once the list holds a NULL



WHERE id NOT IN (1, 2, NULL)

id	name	
1	Asha	FALSE (1 <> 1 is FALSE) → dropped
2	Ben	FALSE (2 <> 2 is FALSE) → dropped
3	Chen	UNKNOWN (3 <> NULL is UNKNOWN) → dropped

result: (no rows)

1. x IN (list) is shorthand for one equality test per value, joined by OR. For Chen, 3 IN (1, 2, NULL) becomes three comparisons.
2. 3 = 1 and 3 = 2 are plainly FALSE. So far nothing matched.
3. 3 = NULL is UNKNOWN: nobody can say whether a missing value equals 3. FALSE OR FALSE OR UNKNOWN is UNKNOWN — not FALSE.
4. NOT IN negates it: an AND of <> tests. 3 <> 1 and 3 <> 2 are TRUE, but 3 <> NULL is UNKNOWN, and TRUE AND TRUE AND UNKNOWN is UNKNOWN.
5. Run for every customer: Asha and Ben are FALSE (their id is in the list), Chen is UNKNOWN. WHERE keeps only TRUE, so the query returns no rows at all.

`AND` is true only if every part is true; one unknown part makes the whole unknown (unless another part is false, which makes it false). `WHERE` keeps only rows where the condition is **true**, so it discards unknown. For Chen, every row: discarded. For Asha, `1 <> 1` is false, so false — discarded as before. Nobody survives.

`IN` with a NULL in the list is mostly harmless: a match makes the `OR` true no matter what else is there. The non-matching rows are unknown rather than false, which you only notice once something negates them — and `NOT IN` is the negation.

NULL on the outer side

The trap has a mirror image. If the outer column is NULL:

```
-- products.category_id is NULL for unclassified products
SELECT name FROM products
WHERE category_id NOT IN (SELECT id FROM discontinued_categories);
```

For an unclassified product the test is `NULL NOT IN (…)`, unknown whenever the list is non-empty — so the product is never returned, even though it is certainly not in a discontinued category. `NOT EXISTS (… WHERE d.id = p.category_id)` returns it, because no row can equal NULL and so no matching row exists. Neither behaviour is wrong in itself; they answer subtly different questions, and you should choose deliberately.

Three reliable anti joins

NOT EXISTS — the default choice:

```
SELECT c.name
FROM customers c
WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.id)
ORDER BY c.name;
```

It asks a yes-or-no question per customer — is there a matching order? — and a NULL `customer_id` simply never matches. No unknowns leak out.

NOT IN with the NULLs filtered out:

```
SELECT c.name
FROM customers c
WHERE c.id NOT IN (SELECT o.customer_id FROM orders o WHERE o.customer_id IS N
ORDER BY c.name;
```

Correct, but it relies on everyone remembering the filter every time. Declaring the column `NOT NULL` in the schema removes the risk at the source — when the data model allows it.

LEFT JOIN … IS NULL:

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

Correct as long as the tested column is never NULL in a real match (the primary key is ideal).

All three agree on the outer-side NULL case too, for the same reason — they test for a matching row rather than compare against a list. MySQL 8.0.17+ plans `NOT EXISTS` and `NOT IN` as antijoins; PostgreSQL often plans `NOT IN` worse precisely because it must honour the NULL semantics. That is the second reason to default to `NOT EXISTS`.

The empty subquery

`x NOT IN (empty)` is true for every x, including NULL — an `AND` of zero comparisons is true. So "products not in any discontinued category" correctly returns every product when nothing is discontinued. The trouble starts only when the list is non-empty and contains a NULL (or the outer value is NULL).

A classic: classifying tree nodes

A table stores a tree as `(id, p_id)`, with `p_id` NULL for the root. Label each node `Root`, `Inner` (has a parent and children) or `Leaf` (has a parent, no children). The tempting solution:

```sql
SELECT id,
       CASE
         WHEN p_id IS NULL THEN 'Root'
         WHEN id IN (SELECT p_id FROM tree) THEN 'Inner'
         ELSE 'Leaf'
       END AS type
FROM tree
ORDER BY id;
```

This one is correct, and it is instructive to see why: `IN` with the root's NULL `p_id` in the list still finds the matches, and for leaves the unknown result just falls through to `ELSE`. Now flip the test, as many people do:

```
WHEN id NOT IN (SELECT p_id FROM tree) THEN 'Leaf'
ELSE 'Inner'
```

The root's NULL `p_id` is in the list, so `NOT IN` is never true, and every leaf is labelled `Inner`. Same data, same intent, one negation — wrong answer. Writing the child test as `EXISTS (SELECT 1 FROM tree c WHERE c.p_id = t.id)` makes both versions correct.

Rules to remember

- `NOT IN (subquery)` is correct only if the subquery's column cannot be NULL **and** the outer value cannot be NULL.
- `NOT EXISTS` is correct regardless; use it by default for "has no match".
- `<> ALL (subquery)` is the same as `NOT IN` and has the same trap.
- The same three-valued logic explains why `WHERE col <> 'x'` skips rows where `col` is NULL.

Pitfalls

- `NOT IN` on a nullable column — the query empties the day the first NULL arrives.
- Forgetting the outer side: a NULL outer value is never returned by `NOT IN`.
- Assuming `NOT (x IN …)` behaves differently — it is the same expression.
- Negating a correct `IN` test into an incorrect `NOT IN` test while refactoring.
- Testing with data that has no NULLs and concluding the query is right.

Key takeaways

- A comparison with NULL is unknown; `WHERE` keeps only true.
- `x NOT IN (…, NULL, …)` is never true, so one NULL empties the result.
- `NOT EXISTS` (or `LEFT JOIN … IS NULL`) is the safe anti join; `NOT IN` needs `WHERE col IS NOT NULL` inside.
- `IN` survives NULLs for matching; its negation does not.
- `NOT IN` of an empty subquery is true for every row.

EXERCISE 1

Employees who manage nobody

List the employees who are nobody's manager. The top of the hierarchy has a NULL `manager_id` — so `id NOT IN (SELECT manager_id FROM employees)` returns nothing here.

Table: `employees(id, name, manager_id)`.

id	name	manager_id
1	Meera	NULL
2	Arjun	1
3	Kiran	2
4	Leela	2
5	Omar	1

Output: columns `id`, `name`, ordered by `id`.

```
id | name
3 | Kiran
4 | Leela
5 | Omar
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), manager_id INT);
INSERT INTO employees VALUES (1,
'Meera', NULL), (2, 'Arjun', 1), (3,
'Kiran', 2), (4, 'Leela', 2), (5,
'Omar', 1);
```

OUTPUT 1

```
id | name
3 | Kiran
4 | Leela
5 | Omar
```

REFERENCE SOLUTION

```
SELECT e.id, e.name
FROM employees e
WHERE NOT EXISTS (
  SELECT 1 FROM employees r WHERE r.manager_id = e.id
)
ORDER BY e.id;
```

EXERCISE 2

Classify the tree nodes

A tree is stored as `tree(id, p_id)`, where `p_id` is the parent's id and NULL for the root. Label every node: `Root` if it has no parent, `Inner` if it has a parent and at least one child, `Leaf` otherwise.

Table: `tree(id, p_id)`.

id	p_id
1	NULL
2	1
3	1
4	2
5	2

Output: columns `id`, `type`, ordered by `id`.

```
id | type
1 | Root
2 | Inner
3 | Leaf
4 | Leaf
5 | Leaf
```

INPUT 1

```
CREATE TABLE tree (id INT PRIMARY KEY,
p_id INT);
INSERT INTO tree VALUES (1, NULL), (2,
1), (3, 1), (4, 2), (5, 2);
```

OUTPUT 1

```
id | type
1 | Root
2 | Inner
3 | Leaf
4 | Leaf
5 | Leaf
```

REFERENCE SOLUTION

```sql
SELECT t.id,
       CASE
         WHEN t.p_id IS NULL THEN 'Root'
         WHEN EXISTS (SELECT 1 FROM tree c WHERE c.p_id = t.id) THEN 'Inner'
         ELSE 'Leaf'
       END AS type
FROM tree t
ORDER BY t.id;
```

QUIZ

1. What is the value of `3 NOT IN (1, 2, NULL)`?

A. TRUE

B. FALSE

C. UNKNOWN (NULL)

D. An error

2. What is the value of `1 IN (1, NULL)`?

A. TRUE

B. FALSE

C. UNKNOWN (NULL)

D. An error

3. `products.category_id` is NULL for some products. Which query returns those products as "not in a discontinued category"?

A. `WHERE category_id NOT IN (SELECT id FROM discontinued)`

B. `WHERE NOT EXISTS (SELECT 1 FROM discontinued d WHERE d.id = p.category_id)`

C. Both

D. Neither

4. Which change makes `WHERE c.id NOT IN (SELECT o.customer_id FROM orders o)` safe when `customer_id` can be NULL?

A. Add `DISTINCT` inside the subquery

B. Add `WHERE o.customer_id IS NOT NULL` inside the subquery

C. Write `NOT (c.id IN (…))`

D. Use `<> ALL` instead

5. What does `x NOT IN (SELECT y FROM t)` return when `t` is empty?

A. TRUE for every row

B. FALSE for every row

C. UNKNOWN for every row

D. TRUE except where x is NULL

ANSWER KEY · NOT IN AND NULL — THE SUBQUERY TRAP

1. C It expands to `3 <> 1 AND 3 <> 2 AND 3 <> NULL` = true AND true AND unknown = unknown. `WHERE` keeps only true, so the row is discarded.

2. A `1 = 1 OR 1 = NULL` = true OR unknown = true. A NULL in the list does not stop `IN` from finding a match; it only makes non-matches unknown instead of false.

3. B `NULL NOT IN (…)` is unknown for a non-empty list, so `NOT IN` never returns them. `NOT EXISTS` finds no row equal to NULL and keeps them. (If `discontinued` were empty, both would.)

4. B Removing the NULLs from the list removes the unknown comparison. `DISTINCT` keeps one NULL; `NOT (x IN …)` and `<> ALL` are the same expression as `NOT IN`.

5. A There is nothing to compare with, and an empty `AND` is true — even when x is NULL. The trap needs a NULL in a non-empty list (or a NULL x with a non-empty list).

CHECKPOINT · 25 MIN

Checkpoint — Subqueries

This checkpoint covers the whole module: scalar subqueries in `WHERE`, the select list and `HAVING`, with their zero-row and many-row behaviour; membership with `IN`, existence with `EXISTS`, multi-column `IN` and `ANY` / `ALL`; correlated subqueries for per-group comparisons and the latest row per group; derived tables for aggregates of aggregates and for joining per-parent totals without fan-out; and the `NOT IN` trap with NULL.

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

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

- What does a scalar subquery return for zero rows, and what does MySQL do for two?
- How do you find the second highest distinct salary, returning NULL when there is none?
- When would you choose `IN`, `EXISTS` or a join?
- What makes a subquery correlated, and why should every inner column be qualified?
- How do you return the latest row per group, and what happens with ties?
- Why must a derived table have an alias, and how does aggregating before a join prevent fan-out?
- Why does `NOT IN` return nothing when its subquery yields a NULL, and what do you write instead?

The three queries are products priced above their own category's average, customers with no order in 2024 when some orders have no customer, and the top-spending customer of each city using a derived table.

EXERCISE 1

Priced above their category average

List the products priced above the average price of their **own** category.

Table: `products(id, name, category, price)`.

id	name	category	price
1	Atlas	Books	40
2	Novel	Books	15
3	Guide	Books	30
4	Chess	Games	25
5	Go	Games	35
6	Dice	Games	5
7	Vinyl	Music	20

Output: columns `product`, `category`, `price`, ordered by `category`, then `price` descending, then `product`.

```
product | category | price
Atlas | Books | 40
Guide | Books | 30
Go | Games | 35
Chess | Games | 25
```

INPUT 1

```
CREATE TABLE products (id INT PRIMARY
KEY, name VARCHAR(40), category
VARCHAR(20), price INT);
INSERT INTO products VALUES (1, 'Atlas',
'Books', 40), (2, 'Novel', 'Books', 15),
(3, 'Guide', 'Books', 30), (4, 'Chess',
'Games', 25), (5, 'Go', 'Games', 35),
(6, 'Dice', 'Games', 5), (7, 'Vinyl',
'Music', 20);
```

OUTPUT 1

```
product | category | price
Atlas | Books | 40
Guide | Books | 30
Go | Games | 35
Chess | Games | 25
```

REFERENCE SOLUTION

```sql
SELECT p.name AS product, p.category, p.price
FROM products p
WHERE p.price > (
  SELECT AVG(x.price) FROM products x WHERE x.category = p.category
)
ORDER BY p.category, p.price DESC, product;
```

EXERCISE 2

No order in 2024

List the customers who placed **no** order dated in 2024 (orders in other years do not count). Some orders have no customer (NULL `customer_id`).

Tables: `customers(id, name)` and `orders(id, customer_id, order_date)`.

customers.id	name
1	Asha
2	Ben
3	Chen
4	Dina

orders.id	customer_id	order_date
10	1	2024-03-01
11	2	2023-12-31
12	2	2025-01-01
13	NULL	2024-05-05
14	4	2024-12-31

Output: columns `id`, `name`, ordered by `id`.

```
id | name
2 | Ben
3 | Chen
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, order_date DATE);
INSERT INTO customers VALUES (1,
'Asha'), (2, 'Ben'), (3, 'Chen'), (4,
'Dina');
INSERT INTO orders VALUES (10, 1, '2024-
03-01'), (11, 2, '2023-12-31'), (12, 2,
'2025-01-01'), (13, NULL, '2024-05-05'),
(14, 4, '2024-12-31');
```

OUTPUT 1

```
id | name
2 | Ben
3 | Chen
```

REFERENCE SOLUTION

```sql
SELECT c.id, c.name
FROM customers c
WHERE NOT EXISTS (
  SELECT 1 FROM orders o
  WHERE o.customer_id = c.id
    AND o.order_date ⩾ '2024-01-01' AND o.order_date < '2025-01-01'
)
ORDER BY c.id;
```

EXERCISE 3

Top spender in each city

For each city, find the customer (or customers, if tied) with the highest total spend. Only customers with at least one order take part; customers with no city are ignored, and orders of unknown customers are ignored.

Tables: `customers(id, name, city)` and `orders(id, customer_id, amount)`.

customers.id	name	city
1	Asha	Pune
2	Ben	Delhi
3	Chen	Pune
4	Dina	Delhi
5	Eli	Kochi

orders.id	customer_id	amount
1	1	200
2	1	100
3	3	250
4	2	90
5	4	60
6	4	30
7	9	1000

Output: columns `city`, `customer`, `total`, ordered by `city`, then `customer`.

```
city | customer | total
Delhi | Ben | 90
Delhi | Dina | 90
Pune | Asha | 300
```

INPUT 1

```
CREATE TABLE customers (id INT PRIMARY
KEY, name VARCHAR(40), city
VARCHAR(40));
CREATE TABLE orders (id INT PRIMARY KEY,
customer_id INT, amount INT);
INSERT INTO customers VALUES (1, 'Asha',
'Pune'), (2, 'Ben', 'Delhi'), (3,
'Chen', 'Pune'), (4, 'Dina', 'Delhi'),
(5, 'Eli', 'Kochi');
INSERT INTO orders VALUES (1, 1, 200),
(2, 1, 100), (3, 3, 250), (4, 2, 90),
(5, 4, 60), (6, 4, 30), (7, 9, 1000);
```

OUTPUT 1

```
city | customer | total
Delhi | Ben | 90
Delhi | Dina | 90
Pune | Asha | 300
```

REFERENCE SOLUTION

```sql
SELECT t.city, t.name AS customer, t.total
FROM (
  SELECT c.id, c.name, c.city, SUM(o.amount) AS total
  FROM customers c
  JOIN orders o ON o.customer_id = c.id
  GROUP BY c.id, c.name, c.city
) AS t
JOIN (
  SELECT x.city, MAX(x.total) AS top
  FROM (
    SELECT c.city, SUM(o.amount) AS total
    FROM customers c
    JOIN orders o ON o.customer_id = c.id
    GROUP BY c.id, c.city
  ) AS x
  GROUP BY x.city
) AS m ON m.city = t.city AND m.top = t.total
ORDER BY t.city, customer;
```

CHECKPOINT QUESTIONS

1. Where may a scalar subquery appear?

A. Only in `WHERE`

B. Anywhere a single value may: `WHERE`, the select list, `HAVING`, inside expressions

C. Only in `FROM`

D. Only after `IN`

2. Which query returns the second highest distinct salary, or NULL when there is none?

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

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

C. `SELECT MAX(salary) - 1 FROM employees`

D. `SELECT MIN(salary) FROM employees`

3. A subquery used with `=` returns two rows in MySQL. What happens?

A. The first row is used

B. Error 1242: Subquery returns more than 1 row

C. The comparison is false

D. The comparison is NULL

4. Which is the clearest way to find students enrolled in both 'SQL' and 'Python'?

A. `… JOIN enrollments e … WHERE e.course = 'SQL' AND e.course = 'Python'`

B. `WHERE id IN (… course = 'SQL') AND id IN (… course = 'Python')`

C. `WHERE id IN (… course IN ('SQL', 'Python'))`

D. `WHERE EXISTS (SELECT 1 FROM enrollments)`

5. What does `EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.id)` check?

A. Whether the subquery returns the value 1

B. Whether at least one order of customer `c` exists

C. Whether customer `c` has exactly one order

D. Whether `orders` has any row at all

6. Which subquery is correlated?

A. `(SELECT AVG(salary) FROM employees)`

B. `(SELECT AVG(x.salary) FROM employees x WHERE x.dept = e.dept)`

C. `(SELECT MAX(order_date) FROM orders)`

D. `(SELECT id FROM products WHERE category = 'Books')`

7. Why qualify every column inside a subquery, as in `SELECT v.customer_id FROM vip v`?

A. MySQL requires it

B. An unqualified column missing from the inner table silently resolves to the outer query, turning a typo into a wrong answer instead of an error

C. It makes the subquery faster

D. It avoids the NOT IN trap

8. Customers have orders and payments. Which approach gives correct totals of both per customer?

A. Join customers, orders and payments, then `SUM` both

B. Join per-customer aggregates of orders and of payments, each computed in its own derived table

C. `SUM(DISTINCT o.amount)` and `SUM(DISTINCT p.amount)` over the three-way join

D. `COUNT(*)` instead of `SUM`

9. What is wrong with `SELECT * FROM (SELECT dept, AVG(salary) FROM employees GROUP BY dept);` in MySQL?

A. Nothing

B. The derived table has no alias (error 1248), and its computed column has no name

C. `AVG` cannot be used in a subquery

D. `GROUP BY` is not allowed in a subquery

10. `orders.customer_id` contains a NULL. What does `WHERE c.id NOT IN (SELECT customer_id FROM orders)` return?

A. The customers without orders

B. No rows

C. Every customer

D. An error

11. Which anti join is correct whatever NULLs the linking column holds?

A. `WHERE x NOT IN (SELECT y FROM t)`

B. `WHERE NOT EXISTS (SELECT 1 FROM t WHERE t.y = outer.x)`

C. `WHERE x <> ALL (SELECT y FROM t)`

D. `WHERE NOT (x IN (SELECT y FROM t))`

12. In the tree-node problem, why is `WHEN id NOT IN (SELECT p_id FROM tree) THEN 'Leaf'` wrong?

A. `CASE` cannot contain a subquery

B. The root's `p_id` is NULL, so `NOT IN` is never true and no node is ever labelled Leaf

C. It labels the root as Leaf

D. It is correct

ANSWER KEY · CHECKPOINT — SUBQUERIES

1. B A scalar subquery is an expression whose value is one value. A subquery in `FROM` is a derived table, and one after `IN` may return many rows.

2. B The `MAX … <` form skips every copy of the top salary and returns NULL (one row) when nothing is left. The `LIMIT` form returns a duplicate of the top when it is tied, and no row at all when there is no second salary.

3. B MySQL checks scalar subqueries at run time. Make the subquery return one value, or use `IN` if a list was intended.

4. B Two membership tests, one per course. One row cannot hold both courses, `course IN (…)` means "either", and an uncorrelated `EXISTS` is true for everyone.

5. B `EXISTS` is true when the subquery returns any row; the select list is irrelevant. The correlation `o.customer_id = c.id` ties it to the current customer.

6. B It refers to `e.dept`, a column of the outer query, so its value changes per outer row. The others mention only their own tables.

7. B Name resolution looks inside first, then outside. Qualified names make a missing column an error.

8. B Each derived table has one row per customer, so nothing multiplies. The direct join fans out, and `SUM(DISTINCT …)` merges different rows with equal amounts.

9. B Write `(SELECT dept, AVG(salary) AS avg_salary FROM employees GROUP BY dept) AS d` and refer to `d.avg_salary` outside.

10. B Every `NOT IN` test includes `c.id <> NULL`, which is unknown, so no row's condition is true. Use `NOT EXISTS` or filter out the NULLs inside.

11. B `NOT EXISTS` asks whether a matching row exists; NULL never matches, so no unknown leaks into the result. The other three are all the same `NOT IN` comparison.

12. B The list of parent ids contains the root's NULL, so `id NOT IN (…)` is unknown for every node that is not a parent, and `CASE` falls through. `NOT EXISTS (SELECT 1 FROM tree c WHERE c.p_id = t.id)` is correct.