Module 7 of 16

Joins

INNER JOIN and row multiplication, LEFT and RIGHT JOIN with the ON-versus-WHERE trap, FULL OUTER JOIN emulated in MySQL, CROSS JOIN grids, self joins for hierarchies and pairs, chains through junction tables with the fan-out problem, and semi and anti joins.

A relational database stores each fact once and links tables by keys, so almost every real question — which customer placed this order, which teacher runs this course, which products never sold — needs rows from more than one table. Joins put them back together. This module teaches every kind of join MySQL 8 offers, the one it does not (FULL OUTER JOIN) and how to write it anyway, and the traps that make a join return plausible but wrong numbers: rows that vanish, rows that repeat, and filters in the wrong clause.

INNER JOIN covers `JOIN … ON`, aliases and qualified columns, the "Cartesian product, then filter" model, one-to-many row repetition, and `USING`, comma and `NATURAL` joins. LEFT and RIGHT JOIN covers NULL-filled rows, `COUNT(col)` versus `COUNT(*)`, `COALESCE` on sums and the classic ON-versus-WHERE trap. FULL and CROSS joins covers the `LEFT JOIN … UNION ALL` emulation, why `UNION` can lose rows, and cross-join grids that report empty cells as zero. Self joins covers managers, pairs with `a.id < b.id` and day-to-day comparisons with `DATE_ADD`. Joining many tables covers chains, junction tables, an inner join undoing a left join, and fan-out. Semi and anti joins covers `EXISTS`, `IN`, `NOT EXISTS`, `LEFT JOIN … IS NULL` and why `NOT IN` breaks on NULL.

Every exercise is a query judged against small datasets with the awkward rows real data has — orders for customers who do not exist, keys that are NULL, empty tables, duplicate names, ties. The checkpoint adds a department report that keeps empty departments, a same-department pairing and an anti join restricted to one term.

CONTENTS

1	INNER JOIN — matching rows across tables	15 min
2	LEFT and RIGHT JOIN — keeping the rows without a match	16 min
3	FULL OUTER and CROSS JOIN — both sides, and every pair	15 min
4	Self joins — a table joined to itself	15 min

5	Joining three or more tables	16 min
6	Semi joins and anti joins — "has a match" and "has none"	15 min
✓	Checkpoint — Joins	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

INNER JOIN — matching rows across tables

A relational database splits data into tables so that each fact is stored once: a customer's name and city live in `customers`, and each order in `orders` points back to its customer by id. That design keeps data consistent — rename a customer once and every order sees the new name — but it means almost every useful question needs rows from more than one table. A **join** is how SQL puts them back together. This lesson covers the `INNER JOIN`: what it computes, how to write the condition, why aliases matter, what happens with one-to-many relationships, and the alternative spellings you will meet in other people's code.

The data for this module

Most examples in the joins module use two small tables.

```
CREATE TABLE customers (id INT PRIMARY KEY, name VARCHAR(40), city VARCHAR(40)
CREATE TABLE orders (id INT PRIMARY KEY, customer_id INT, order_date DATE, amo

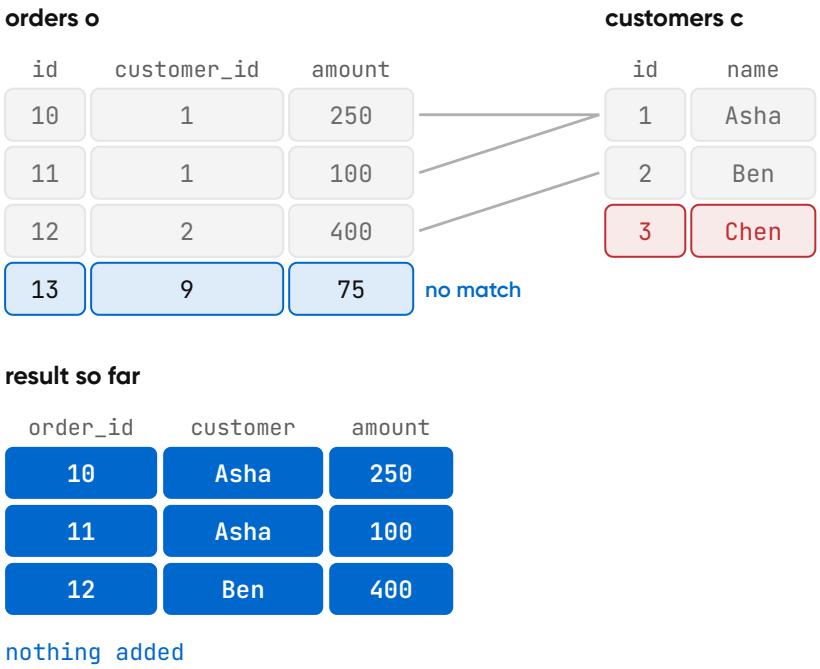
INSERT INTO customers VALUES (1, 'Asha', 'Pune'), (2, 'Ben', 'Delhi'), (3, 'Ch
INSERT INTO orders VALUES
  (10, 1, '2024-01-05', 250),
  (11, 1, '2024-02-11', 100),
  (12, 2, '2024-02-20', 400),
  (13, 9, '2024-03-01', 75);
```

`orders.customer_id` is a **foreign key**: it holds the `id` of a row in `customers`. Notice the awkward cases on purpose: Chen (3) has no orders, and order 13 points at customer 9, who does not exist. Real data has both, and the kind of join you choose decides what happens to them.

Writing an INNER JOIN

```
SELECT o.id AS order_id, c.name AS customer, o.amount
FROM orders AS o
JOIN customers AS c ON c.id = o.customer_id
ORDER BY o.id;
```

INNER JOIN: each order looks for its customer



1. ON c.id = o.customer_id is tested for each order against every customer. Each pair that comes out TRUE becomes one result row; nothing else does.
2. Order 10 has customer_id 1, which equals Asha's id: one result row, order 10 with Asha.
3. Order 11 has customer_id 1, which equals Asha's id: one result row, order 11 with Asha.
4. Order 12 has customer_id 2, which equals Ben's id: one result row, order 12 with Ben.
5. Order 13 has customer_id 9, and no customer has id 9. No pair is TRUE, so an inner join adds nothing for it — the order disappears.
6. 3 rows. Chen was never anyone's match, so Chen is missing too: an inner join keeps only rows that found a partner, on both sides.

Read it as: take each order, find every customer whose `id` equals the order's `customer_id`, and output one combined row per pair. Order 13 has no matching customer, so it disappears. Chen has no matching order, so Chen disappears too. That is the defining property of an inner join: **only rows that found a partner survive**, from both sides.

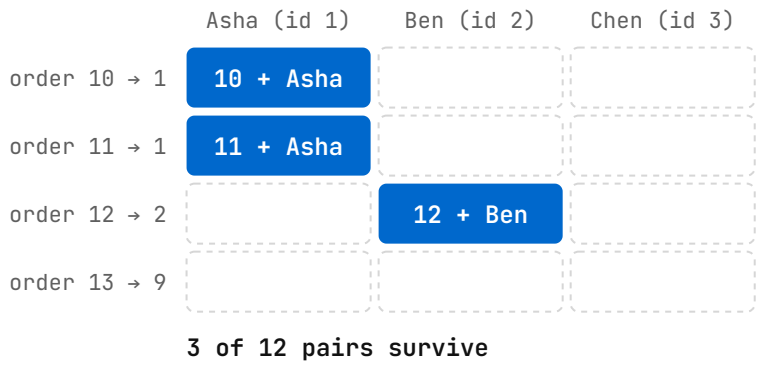
`JOIN` and `INNER JOIN` are the same thing in MySQL; the word `INNER` is optional. The `ON` clause is a boolean condition evaluated for each candidate pair — usually key equality, but any expression is allowed.

A mental model: filter the Cartesian product

Logically, a join builds every possible pair of rows — the **Cartesian product**, 4 orders × 3 customers = 12 pairs here — and keeps the pairs where `ON` is true. The database never

actually builds all twelve; it uses indexes, hash tables or sorted merges. But the model gives the right answer every time, including in the strange cases.

The model: every pair, then keep where ON is TRUE



1. Logically a join starts from the Cartesian product: every order paired with every customer, 4 × 3 = 12 pairs. The engine never builds them all, but the answer is always as if it had.
2. ON is evaluated for every pair. Only three cells are TRUE: the pairs where the customer's id equals the order's customer_id.
3. The TRUE cells are the result. Order 13's row has no TRUE cell and Chen's column has none, so neither appears anywhere in an inner join.

The strange cases, read off the grid:

- a row that matches nothing produces no pair;
- a row that matches three rows produces three pairs;
- a `NULL` key matches nothing, because `NULL = 1` is not true — it is unknown, and `ON` keeps only true.

Aliases and qualified columns

Both tables have an `id` column. Writing `SELECT id FROM orders JOIN customers ON …` fails in MySQL with *Column 'id' in field list is ambiguous*. Qualify every column: `o.id`, `c.id`. Table aliases (`orders AS o`, or just `orders o`) keep that short. Once a table has an alias, use the alias everywhere in the query — MySQL refuses `orders.id` when the table was renamed `o`.

Also alias the output columns. Two columns both called `id` or `name` in a result are confusing to read and impossible to tell apart in application code that reads columns by name.

One-to-many: why rows repeat

```sql
SELECT c.name, o.id AS order_id, o.amount
FROM customers c
JOIN orders o ON o.customer_id = c.id
ORDER BY c.name, o.id;
```

name	order_id	amount
Asha	10	250
Asha	11	100
Ben	12	400

Asha appears twice because she has two orders. This is correct, and it is what makes aggregation after a join work:

```sql
SELECT c.name, COUNT(*) AS orders, SUM(o.amount) AS spent
FROM customers c
JOIN orders o ON o.customer_id = c.id
GROUP BY c.id, c.name
ORDER BY spent DESC, c.name;
```

name	orders	spent
Ben	1	400
Asha	2	350

Group by the key (`c.id`) as well as the name: two different customers can share a name, and grouping by name alone would merge them. The danger comes when a column from the "one" side is summed after the join — summing a customer's credit limit over their order rows counts it once per order. Keep that in mind; the joining-many-tables lesson returns to it.

Joining and filtering

`WHERE` runs after the join and filters the combined rows, so it may refer to either table:

```sql
SELECT o.id, c.name, o.amount
FROM orders o
JOIN customers c ON c.id = o.customer_id
WHERE c.city = 'Pune' AND o.amount ⩾ 200
ORDER BY o.id;
```

For an inner join it makes no difference whether a condition is placed in `ON` or in `WHERE` — the rows that survive are the same. Convention puts the relationship (`c.id = o.customer_id`) in `ON` and the filters in `WHERE`. For outer joins the placement changes the answer, which is the first trap of the next lesson.

Other spellings you will meet

```sql
-- USING: the key has the same name in both tables
SELECT * FROM orders JOIN shipments USING (order_id);

-- Comma join: the older, implicit form
SELECT o.id, c.name FROM orders o, customers c WHERE c.id = o.customer_id;

-- NATURAL JOIN: joins on every column name the tables share
SELECT * FROM orders NATURAL JOIN customers;   -- avoid
```

`USING` is fine when the names match; it also prints the shared column once. The comma join is equivalent to an inner join, but if you forget the `WHERE` condition you silently get the Cartesian product — the explicit `JOIN … ON` makes the missing condition obvious. `NATURAL JOIN` is a trap: here both tables have `id`, so it would join `orders.id = customers.id`, which is meaningless, and adding a column later can change the result without any error.

Conditions other than equality

Most joins are equi-joins, but the `ON` clause accepts any condition. A price band lookup is a classic **non-equi join**:

```sql
SELECT p.name, b.label
FROM products p
JOIN price_bands b ON p.price BETWEEN b.low AND b.high
ORDER BY p.name;
```

Each product finds the band whose range contains its price. If the bands overlap, a product matches two bands and appears twice — the same row-multiplication rule as before.

Pitfalls

- Selecting an unqualified column that exists in both tables (ambiguous column error).
- Expecting unmatched rows to appear — an inner join drops them on both sides.
- Grouping by a non-unique column such as a name instead of the key.
- Summing a parent-side value after a one-to-many join and counting it repeatedly.
- `NATURAL JOIN` or a comma join without its condition.

Key takeaways

- `JOIN … ON` pairs rows whose condition is true; rows without a partner vanish from both sides.
- Think "Cartesian product, then filter by `ON`" to predict any result.
- `NULL` keys never match, because `NULL = x` is unknown.
- One-to-many joins repeat the parent row once per child — correct, and the basis of join + `GROUP BY`.
- Alias tables, qualify every column, alias output columns, and prefer `ON` to `NATURAL`.

EXERCISE 1

Orders with their customers

List every order together with the name and city of the customer who placed it. Orders whose customer does not exist (or is NULL) are not listed.

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

customers.id	name	city
1	Asha	Pune
2	Ben	Delhi
3	Chen	Pune

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	9	2024-03-01	75

Output: columns `order_id`, `customer`, `city`, `amount`, ordered by `order_id`.

```
order_id | customer | city | amount
10 | Asha | Pune | 250
11 | Asha | Pune | 100
12 | Ben | Delhi | 400
```

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, order_date DATE, amount
INT);
INSERT INTO customers VALUES (1, 'Asha',
'Pune'), (2, 'Ben', 'Delhi'), (3,
'Chen', 'Pune');
INSERT INTO orders VALUES (10, 1, '2024-
01-05', 250), (11, 1, '2024-02-11',
100), (12, 2, '2024-02-20', 400), (13,
9, '2024-03-01', 75);
```

OUTPUT 1

```
order_id | customer | city | amount
10 | Asha | Pune | 250
11 | Asha | Pune | 100
12 | Ben | Delhi | 400
```

REFERENCE SOLUTION

```sql
SELECT o.id AS order_id, c.name AS customer, c.city, o.amount
FROM orders o
JOIN customers c ON c.id = o.customer_id
ORDER BY o.id;
```

EXERCISE 2

Revenue per city

For each city, report how many orders its customers placed and their total amount. Only orders that belong to an existing customer count; cities with no orders are not listed.

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

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

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	3	2024-02-25	50
14	9	2024-03-01	75

Output: columns `city`, `orders`, `revenue`, ordered by `revenue` descending, then `city` ascending.

```
city | orders | revenue
Delhi | 1 | 400
Pune | 3 | 400
```

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, order_date DATE, amount
INT);
INSERT INTO customers VALUES (1, 'Asha',
'Pune'), (2, 'Ben', 'Delhi'), (3,
'Chen', 'Pune'), (4, 'Dina', 'Kochi');
INSERT INTO orders VALUES (10, 1, '2024-
01-05', 250), (11, 1, '2024-02-11',
100), (12, 2, '2024-02-20', 400), (13,
3, '2024-02-25', 50), (14, 9, '2024-03-
01', 75);
```

OUTPUT 1

```
city | orders | revenue
Delhi | 1 | 400
Pune | 3 | 400
```

REFERENCE SOLUTION

```sql
SELECT c.city, COUNT(*) AS orders, SUM(o.amount) AS revenue
FROM orders o
JOIN customers c ON c.id = o.customer_id
GROUP BY c.city
ORDER BY revenue DESC, c.city;
```

QUIZ

1. `customers` has 3 rows and `orders` has 4. Customer 3 has no orders and one order points at a customer id that does not exist. How many rows does `customers JOIN orders ON orders.customer_id = customers.id` return?

A. 3

B. 4

C. 2 (one per customer who has orders)

D. 12

2. `SELECT id, name, amount FROM orders JOIN customers ON customers.id = orders.customer_id;` fails in MySQL. Why?

A. `JOIN` needs the word `INNER`

B. `id` exists in both tables, so the unqualified column is ambiguous

C. The tables must be given aliases before they can be joined

D. `ON` must compare columns with the same name

3. Which statement about a one-to-many join between `customers` and `orders` is true?

A. Each customer appears once, with their orders combined

B. A customer appears once per matching order

C. MySQL removes duplicate customer rows automatically

D. Only the first matching order is joined

4. Why is `NATURAL JOIN` usually avoided?

A. MySQL does not support it

B. It joins on every column name the two tables share, so an unrelated shared column (like `id` or `created_at`) silently becomes part of the condition

C. It is a cross join

D. It only works with primary keys

ANSWER KEY · INNER JOIN — MATCHING ROWS ACROSS TABLES

1. A Only the three orders whose customer exists find a partner, so three rows come back. The customer with no orders and the orphan order are both dropped — an inner join keeps only matched pairs. 12 would be the Cartesian product, which is what you get with no `ON` at all. "One per customer" is wrong: a join returns pairs, and a customer with two orders appears twice.

2. B MySQL reports *Column 'id' in field list is ambiguous*. Qualify it (`orders.id`). `INNER` is optional, aliases are a convenience rather than a requirement for two different tables, and `ON` can compare any columns.

3. B A join outputs one row per matching pair, so a customer with three orders appears three times. Nothing is combined or de-duplicated unless you `GROUP BY` or use `DISTINCT`.

4. B MySQL supports it, but the condition is implicit: tables that both have `id` join on `a.id = b.id`, which is meaningless for a foreign-key relationship, and adding a column later can change the result with no error. `JOIN … ON` states the condition.

LESSON 2 · 16 MIN

LEFT and RIGHT JOIN — keeping the rows without a match

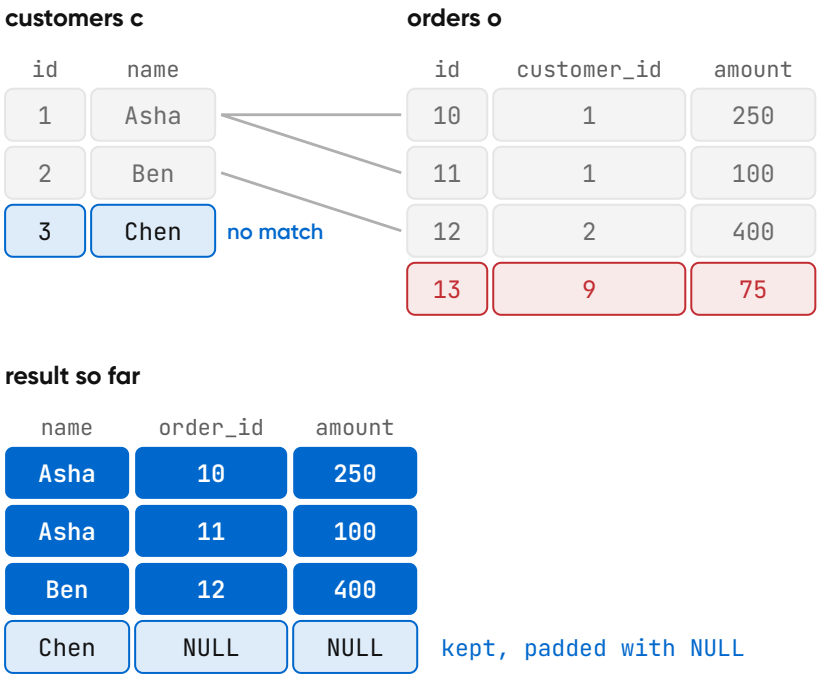
An inner join answers "which pairs match?". Many real questions are different: *every* customer and how much each spent, including the ones who spent nothing; every product and its last sale, if any; every student and their grade, blank if not yet graded. Dropping the unmatched rows would give a wrong report — the customers who never ordered are often exactly the ones the business wants to see. **Outer joins** keep them. This lesson covers `LEFT JOIN`, its mirror `RIGHT JOIN`, counting and summing after an outer join, and the most common outer-join bug in interviews and production code: a condition in the wrong clause.

LEFT JOIN

```sql
SELECT c.name, o.id AS order_id, o.amount
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
ORDER BY c.name, o.id;
```

With the module's data (Asha has orders 10 and 11, Ben has 12, Chen has none, order 13 belongs to a missing customer 9):

LEFT JOIN: every customer, matched or not



1. LEFT JOIN guarantees every row of the table written first, customers. Each customer is paired with its matching orders, exactly as in an inner join.
2. Asha (id 1) matches orders 10 and 11: 2 result rows, one per match.
3. Ben (id 2) matches order 12: 1 result row, one per match.
4. Chen matches no order. An inner join would drop Chen; a left join keeps one row with NULL in every column that comes from orders.
5. Order 13, for the missing customer 9, is still absent: the right side keeps only matched rows. Swap the tables with RIGHT JOIN and it would be the one kept with a NULL name.

The rule: **every row of the left table appears at least once**. A left row with matches appears once per match, exactly as in an inner join. A left row with no match appears once, with every column of the right table set to `NULL`. Order 13 is still gone — it is on the right side, and the right side keeps only matched rows.

"Left" simply means the table written before `LEFT JOIN`. Choosing it is choosing whose rows are guaranteed: here, customers.

RIGHT JOIN is the same thing backwards

```sql
SELECT c.name, o.id AS order_id
FROM customers c
RIGHT JOIN orders o ON o.customer_id = c.id
ORDER BY o.id;
```

name	order_id
Asha	10
Asha	11
Ben	12
NULL	13

Now every order is kept and the orphan order 13 shows a `NULL` name. `A RIGHT JOIN B` equals `B LEFT JOIN A`, so it never adds expressive power. Most style guides write only left joins — the preserved table is then always the one in `FROM`, which you can see at a glance. You should still read right joins fluently, because they appear in legacy code and interview questions.

Counting after an outer join

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

name	wrong_count	orders	spent
Asha	2	2	350
Ben	1	1	400
Chen	1	0	0

Chen's group has one row — the NULL-filled one — so `COUNT(*)` says 1. `COUNT(o.id)` counts non-NULL values and says 0, which is the truth. Count a right-side column that is never NULL in a real match (its primary key is ideal). `SUM` over nothing but NULLs is `NULL`, not 0; `COALESCE(SUM(o.amount), 0)` turns it into a number for the report.

The trap: ON versus WHERE

Suppose the report should count only orders from February 2024, but still list every customer.

```sql
-- Wrong: the filter is in WHERE
SELECT c.name, COUNT(o.id) AS feb_orders
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
WHERE o.order_date ⩾ '2024-02-01' AND o.order_date < '2024-03-01'
GROUP BY c.id, c.name
ORDER BY c.name;
```

name	feb_orders
Asha	1
Ben	1

Chen vanished. The join produced Chen's NULL-filled row as promised — and then `WHERE` evaluated `NULL ⩾ '2024-02-01'`, which is unknown, and discarded it. Any `WHERE` condition on a right-table column that rejects `NULL` turns the left join back into an inner join.

```sql
-- Right: the filter is part of the match
SELECT c.name, COUNT(o.id) AS feb_orders
FROM customers c
LEFT JOIN orders o
  ON o.customer_id = c.id
 AND o.order_date ⩾ '2024-02-01' AND o.order_date < '2024-03-01'
GROUP BY c.id, c.name
ORDER BY c.name;
```

name	feb_orders
Asha	1
Ben	1
Chen	0

The trap: a right-table filter in WHERE undoes the left join

LEFT JOIN … ON id AND date

name	order_id	order_date
Asha	11	2024-02-11
Ben	12	2024-02-20
Chen	NULL	NULL

GROUP BY → result

name	feb_orders
Asha	1
Ben	1
Chen	0

1. The left join runs first and keeps Chen as promised, with NULL for every order column, order_date included.
2. WHERE then tests each joined row. January's order is FALSE; Chen's NULL date makes NULL >= '2024-02-01' UNKNOWN, and WHERE keeps only TRUE.
3. Only the two TRUE rows reach GROUP BY, so Chen has vanished — the WHERE condition quietly turned the left join into an inner join.
4. Move the date test into ON and it only decides which orders count as a match. January's order no longer matches Asha, and Chen, matching nothing, is kept with NULLs.
5. Counting o.id now gives every customer a row: Asha 1, Ben 1, Chen 0. Right-table filters go in ON; left-table filters go in WHERE.

The rule to memorise:

- a condition on the **right (optional) table** that decides *which rows count as a match* belongs in `ON`;
- a condition on the **left (preserved) table** that decides *which left rows appear at all* belongs in `WHERE` — putting it in `ON` does not filter the left table, it only stops those left rows from matching, and they still appear with NULLs.

```sql
-- Only Pune customers, each with their orders (or NULLs)
SELECT c.name, o.id
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
WHERE c.city = 'Pune'
ORDER BY c.name, o.id;
```

There is one deliberate exception: `WHERE o.id IS NULL` after a left join keeps *only* the unmatched rows. That is the anti-join pattern, covered in the semi- and anti-joins lesson.

Chaining outer joins

Left joins chain left to right: each one keeps every row produced so far.

```sql
SELECT c.name, o.id AS order_id, s.shipped_on
FROM customers c
LEFT JOIN orders o    ON o.customer_id = c.id
LEFT JOIN shipments s ON s.order_id = o.id
ORDER BY c.name, o.id;
```

Chen still appears (NULL order, NULL shipment), and an order with no shipment appears with a NULL date. If the second join were an inner `JOIN`, its condition `s.order_id = o.id` would fail for Chen's NULL order, and Chen would disappear — one inner join after a left join quietly undoes it. The joining-many-tables lesson works through that case.

Telling a real NULL from a missing match

If a right-table column can itself be NULL — a `shipped_on` that is NULL for an order not yet shipped — then a NULL in the result is ambiguous: no matching row, or a matching row with an empty value? Test the right table's primary key instead: `s.id IS NULL` means no match, because a real row always has its key.

In other databases

`LEFT [OUTER] JOIN` and `RIGHT [OUTER] JOIN` are standard and behave identically in PostgreSQL, SQL Server, Oracle and SQLite. Oracle's legacy `(+)` syntax (`WHERE c.id = o.customer_id(+)`) is the same left join in old code. The ON-versus-WHERE behaviour is part of the standard, so the trap exists everywhere.

Pitfalls

- Filtering the optional table in `WHERE` and losing the unmatched rows.
- `COUNT(*)` after a left join counting the NULL row as 1.
- `SUM` returning NULL for "nothing" instead of 0.
- An inner join chained after a left join.

- Testing a nullable right-side column to detect "no match".

Key takeaways

- `LEFT JOIN` keeps every left row; unmatched rows get NULLs for the right table's columns.
- `A RIGHT JOIN B` = `B LEFT JOIN A`; prefer writing left joins.
- Count a right-side key (`COUNT(o.id)`), and `COALESCE` sums to 0.
- Conditions on the optional table go in `ON`; conditions on the preserved table go in `WHERE`.
- Detect "no match" with the right table's primary key being NULL.

EXERCISE 1

Every customer's orders and spend

List **every** customer with the number of orders they placed and the total they spent. Customers with no orders show `0` and `0`. An order whose `amount` is NULL still counts as an order but adds nothing to the total.

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

customers.id	name	city
1	Asha	Pune
2	Ben	Delhi
3	Chen	Pune

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	9	2024-03-01	75

Output: columns `customer`, `orders`, `spent`, ordered by `spent` descending, then `customer` ascending.

```
customer | orders | spent
Ben | 1 | 400
Asha | 2 | 350
Chen | 0 | 0
```

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, order_date DATE, amount
INT);
INSERT INTO customers VALUES (1, 'Asha',
'Pune'), (2, 'Ben', 'Delhi'), (3,
'Chen', 'Pune');
INSERT INTO orders VALUES (10, 1, '2024-
01-05', 250), (11, 1, '2024-02-11',
100), (12, 2, '2024-02-20', 400), (13,
9, '2024-03-01', 75);
```

OUTPUT 1

```
customer | orders | spent
Ben | 1 | 400
Asha | 2 | 350
Chen | 0 | 0
```

REFERENCE SOLUTION

```sql
SELECT c.name AS customer, COUNT(o.id) AS orders, COALESCE(SUM(o.amount), 0) AS sper
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
GROUP BY c.id, c.name
ORDER BY spent DESC, customer;
```

EXERCISE 2

This term's enrolments for every student

List **every** student with the number of courses they are enrolled in for the term `'2024-S1'`. Students with no enrolment in that term show `0`. Two students can share a name; they are different students.

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

students.id	name
1	Anu
2	Bilal
3	Chitra

student_id	course_id	term
1	101	2024-S1
1	102	2024-S1
2	101	2023-S2
3	103	2024-S1
1	103	2023-S2

Output: columns `student`, `courses`, ordered by `student`, then `courses`.

```
student | courses
Anu | 2
Bilal | 0
Chitra | 1
```

INPUT 1

```
CREATE TABLE students (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE enrollments (student_id
INT, course_id INT, term VARCHAR(10));
INSERT INTO students VALUES (1, 'Anu'),
(2, 'Bilal'), (3, 'Chitra');
INSERT INTO enrollments VALUES (1, 101,
'2024-S1'), (1, 102, '2024-S1'), (2,
101, '2023-S2'), (3, 103, '2024-S1'),
(1, 103, '2023-S2');
```

OUTPUT 1

```
student | courses
Anu | 2
Bilal | 0
Chitra | 1
```

REFERENCE SOLUTION

```sql
SELECT s.name AS student, COUNT(e.course_id) AS courses
FROM students s
LEFT JOIN enrollments e ON e.student_id = s.id AND e.term = '2024-S1'
GROUP BY s.id, s.name
ORDER BY student, courses;
```

QUIZ

1. This query should list every customer with their count of paid orders, including customers with none. What does it actually return?

```sql
SELECT c.name, COUNT(o.id) AS paid
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
WHERE o.status = 'paid'
GROUP BY c.id, c.name;
```

A. Every customer, with 0 for those without paid orders

B. Only customers with at least one paid order

C. An error: `WHERE` cannot refer to the right table of a left join

D. Every customer, but with NULL instead of 0

2. After `customers c LEFT JOIN orders o ON o.customer_id = c.id … GROUP BY c.id`, what does `COUNT(*)` report for a customer with no orders?

A. 0

B. 1

C. NULL

D. The query fails

3. Which query returns the same rows as `SELECT … FROM a RIGHT JOIN b ON b.a_id = a.id`?

A. `SELECT … FROM a LEFT JOIN b ON b.a_id = a.id`

B. `SELECT … FROM b LEFT JOIN a ON b.a_id = a.id`

C. `SELECT … FROM a JOIN b ON b.a_id = a.id`

D. `SELECT … FROM b RIGHT JOIN a ON b.a_id = a.id`

4. You want only Pune customers, each with their orders (NULLs if none). Where does `c.city = 'Pune'` go?

A. In `WHERE`, because it filters the preserved (left) table

B. In `ON`, because every condition of a left join belongs there

C. Either; the result is the same

D. In `HAVING`

ANSWER KEY · LEFT AND RIGHT JOIN — KEEPING THE ROWS WITHOUT A MATCH

1. B For a customer with no orders, the left join produces a row whose `o.status` is NULL; `NULL = 'paid'` is unknown, so `WHERE` drops it — and a customer whose orders are all unpaid is dropped the same way. The outer join has become an inner join. Moving `o.status = 'paid'` into the `ON` clause fixes it. It is not an error: `WHERE` may refer to any table.

2. B The left join still produces one row for that customer (with NULL order columns), and `COUNT(*)` counts rows. `COUNT(o.id)` skips NULLs and gives the correct 0.

3. B A right join keeps every row of the table on its right, `b`. Swapping the tables and using `LEFT JOIN` keeps every row of `b` too. The first option keeps every `a` instead, the inner join keeps neither side's unmatched rows, and the last keeps every `a`.

4. A A condition on the left table decides which left rows appear at all, so it goes in `WHERE`. In `ON` it would only stop non-Pune customers from *matching* — they would still all appear, with NULL orders. Only for inner joins are `ON` and `WHERE` interchangeable.

LESSON 3 · 15 MIN

FULL OUTER and CROSS JOIN — both sides, and every pair

`INNER JOIN` keeps matched pairs; `LEFT JOIN` also keeps the unmatched rows of one side. Two more joins complete the family. A **full outer join** keeps the unmatched rows of *both* sides — the natural tool for reconciling two lists that should agree but do not. A **cross join** has no matching at all: it pairs every row with every row, which sounds useless until you need a grid of every combination. MySQL lacks the first, so this lesson teaches the emulation every MySQL developer writes, and then the right and wrong uses of the second.

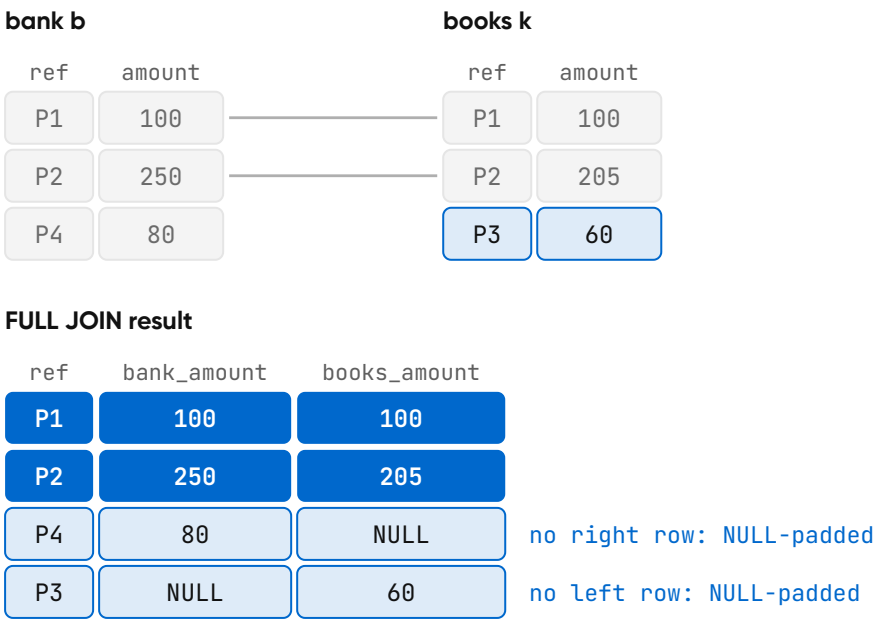
What a full outer join returns

Take two lists of payment references: what the bank says it received, and what the accounts team recorded.

```
CREATE TABLE bank  (ref VARCHAR(10) PRIMARY KEY, amount INT);
CREATE TABLE books (ref VARCHAR(10) PRIMARY KEY, amount INT);
INSERT INTO bank  VALUES ('P1', 100), ('P2', 250), ('P4', 80);
INSERT INTO books VALUES ('P1', 100), ('P2', 205), ('P3', 60);
```

A full outer join on `ref` returns:

FULL OUTER JOIN: matched pairs, plus the leftovers of both sides



1. A full outer join keeps matched pairs once, plus the unmatched rows of both sides. Bank rows are read first, as a left join reads them.
2. Bank P1 finds P1 in the books: one row with both amounts, 100 and 100.
3. Bank P2 finds P2 in the books: one row with both amounts, 250 and 205. They differ — a discrepancy the report surfaces.
4. Bank P4 has no row in the books. The left half keeps it, with NULL for books_amount: money received but never recorded.
5. Last, the books rows no bank row matched — P3 — are added with NULL for bank_amount. In MySQL that is the second half of LEFT JOIN … UNION ALL … WHERE b.ref IS NULL.

Matched references appear once with both amounts; P4 exists only at the bank; P3 only in the books. All three kinds of discrepancy are visible in one result. In PostgreSQL or SQL Server this is one clause:

```
-- PostgreSQL / SQL Server / SQLite 3.39+
SELECT COALESCE(b.ref, k.ref) AS ref, b.amount AS bank_amount, k.amount AS boo
FROM bank b
FULL OUTER JOIN books k ON k.ref = b.ref;
```

Note `COALESCE(b.ref, k.ref)`: on a books-only row, `b.ref` is NULL, so the key must be taken from whichever side has it.

Emulating it in MySQL

MySQL 8 has no `FULL OUTER JOIN`. Worse, `FULL` is not a reserved word there: `FROM bank FULL JOIN books ON …` is read as the table `bank` with the alias `FULL`, followed by a plain inner join — it runs, and silently returns only the matched rows. `FULL OUTER JOIN` at least fails with a syntax error. The emulation splits the answer into the two parts a left join can produce:

1. every bank row, matched or not — `bank LEFT JOIN books`;
2. the books rows that matched no bank row — `books LEFT JOIN bank … WHERE bank.ref IS NULL`.

```sql
SELECT b.ref AS ref, b.amount AS bank_amount, k.amount AS books_amount
FROM bank b
LEFT JOIN books k ON k.ref = b.ref
UNION ALL
SELECT k.ref, NULL, k.amount
FROM books k
LEFT JOIN bank b ON b.ref = k.ref
WHERE b.ref IS NULL
ORDER BY ref;
```

How `UNION ALL` works (the set-operations module covers it fully): the two `SELECT`s must return the same number of columns, the result takes its column names from the first one, the rows of the second are appended, and an `ORDER BY` at the end sorts the combined result. The second part's rows have no bank side, so it writes `NULL` for `bank_amount` explicitly.

The second query is often written `RIGHT JOIN` instead, which is the same thing with the tables in the original order:

```sql
SELECT b.ref, b.amount, k.amount FROM bank b LEFT JOIN books k ON k.ref = b.re
UNION ALL
SELECT k.ref, b.amount, k.amount FROM bank b RIGHT JOIN books k ON k.ref = b.r
WHERE b.ref IS NULL;
```

Why UNION ALL and not UNION

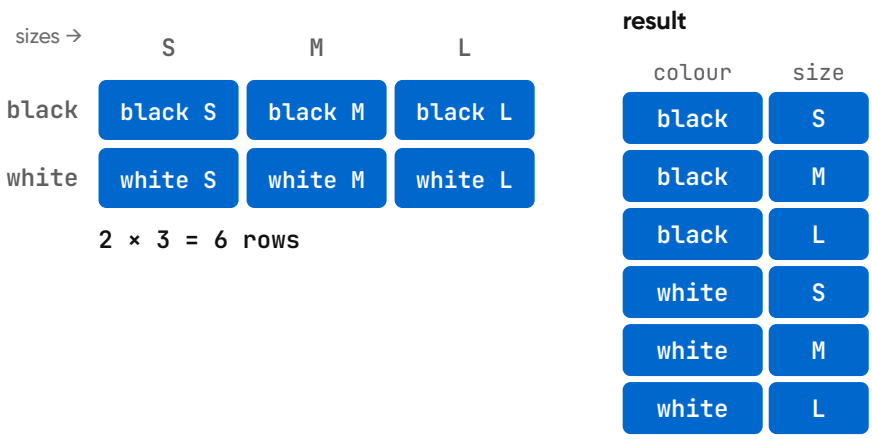
You will often see the shorter version: `… LEFT JOIN … UNION … RIGHT JOIN …` with no `WHERE`. The matched rows appear in both halves, and `UNION` (which removes duplicate rows) collapses them. That works only if every genuine result row is distinct. If the joined data can contain two identical rows — two line items with the same product and quantity, say — `UNION` deletes one of them, and nothing warns you. The `UNION ALL … WHERE key IS NULL` form is exact and avoids the duplicate-removal sort. Interviewers like candidates who know the difference.

CROSS JOIN: every pair

```sql
CREATE TABLE sizes   (label VARCHAR(4), position INT);
CREATE TABLE colours (name VARCHAR(10));
INSERT INTO sizes   VALUES ('S', 1), ('M', 2), ('L', 3);
INSERT INTO colours VALUES ('black'), ('white');

SELECT c.name AS colour, s.label AS size
FROM colours c
CROSS JOIN sizes s
ORDER BY c.name, s.position;
```

CROSS JOIN: every colour with every size



1. A cross join has no ON. Its result is the whole grid: one row for every pairing of a colour with a size, 2 × 3 cells.
2. The first colour, black, is paired with each of the 3 sizes: three rows.
3. Then white with each size again. Every row of one table meets every row of the other; if either table were empty, so would be the result.
4. Read off row by row, ordered by colour then size position, the grid is the six-row result. This is also what an accidental join without ON returns, on tables far bigger than these.

Two colours × three sizes = six rows. There is no `ON`: every combination is the point. If either table is empty, the product is empty.

The grid pattern

The most useful cross join builds the frame of a report so that missing combinations show up as zeros instead of vanishing. Sales by region and month, where some regions sold nothing in some months:

```sql
SELECT r.name AS region, m.month, COALESCE(SUM(s.amount), 0) AS total
FROM regions r
CROSS JOIN months m
LEFT JOIN sales s ON s.region = r.name AND s.month = m.month
GROUP BY r.name, m.month
ORDER BY r.name, m.month;
```

The cross join produces every (region, month) cell; the left join attaches the sales that belong in each cell, using both conditions in `ON`; cells with no sales keep one NULL row, and `COALESCE(SUM(…), 0)` reports 0. Without the grid, a `GROUP BY` over `sales` alone can only report combinations that occurred — a chart built from it would silently skip the empty months.

Accidental cross joins

The dangerous cross join is the one nobody asked for.

```sql
SELECT o.id, c.name FROM orders o, customers c;        -- forgot the WHERE
SELECT o.id, c.name FROM orders o JOIN customers c;    -- MySQL: forgot the ON
```

Both pair every order with every customer. On ten-thousand-row tables that is a hundred million rows. In MySQL, `JOIN`, `INNER JOIN` and `CROSS JOIN` are interchangeable, and any of them without `ON` is a Cartesian product; PostgreSQL refuses `JOIN` without `ON`, but the comma form is accepted everywhere. A result with suspiciously many rows, or each row repeated once per row of another table, is the symptom.

A join whose `ON` condition is true too often has the same effect in miniature — joining orders to customers on `city` instead of on the id pairs each order with every customer in the same city.

Pitfalls

- Writing `FULL OUTER JOIN` in MySQL (a syntax error) or `FULL JOIN` (an inner join in disguise).
- Emulating it with `UNION` when real rows can be identical.
- Forgetting `COALESCE(a.key, b.key)` and showing NULL keys for one-sided rows.

- Putting the grid's filter on the sales table in `WHERE` instead of `ON` (the previous lesson's trap).
- A join with no condition, or a condition that is not a key.

Key takeaways

- A full outer join keeps matched pairs and the unmatched rows of both sides; MySQL emulates it with `LEFT JOIN … UNION ALL …` the other side's unmatched rows.
- `UNION ALL` plus `WHERE key IS NULL` is exact; plain `UNION` can delete real duplicates.
- `CROSS JOIN` returns m × n rows: every combination.
- Cross join the dimensions, then left join the facts, to report every cell including the empty ones.
- In MySQL a `JOIN` without `ON` is a cross join — check row counts when a result looks too big.

EXERCISE 1

Reconcile two ledgers

The bank's list of payments and the accounts team's list should match, but do not. Show every reference that appears in **either** list, with the amount on each side (NULL where a side has no record). Write it the MySQL way: MySQL has no `FULL OUTER JOIN`.

Tables: `bank(ref, amount)` and `books(ref, amount)`; `ref` is the primary key of each.

bank.ref	amount
P1	100
P2	250
P4	80

books.ref	amount
P1	100
P2	205
P3	60

Output: columns `ref`, `bank_amount`, `books_amount`, ordered by `ref`.

```
ref | bank_amount | books_amount
P1 | 100 | 100
P2 | 250 | 205
P3 | NULL | 60
P4 | 80 | NULL
```

INPUT 1

```
CREATE TABLE bank (ref VARCHAR(10)
PRIMARY KEY, amount INT);
CREATE TABLE books (ref VARCHAR(10)
PRIMARY KEY, amount INT);
INSERT INTO bank VALUES ('P1', 100),
('P2', 250), ('P4', 80);
INSERT INTO books VALUES ('P1', 100),
('P2', 205), ('P3', 60);
```

OUTPUT 1

```
ref | bank_amount | books_amount
P1 | 100 | 100
P2 | 250 | 205
P3 | NULL | 60
P4 | 80 | NULL
```

REFERENCE SOLUTION

```sql
SELECT b.ref AS ref, b.amount AS bank_amount, k.amount AS books_amount
FROM bank b
LEFT JOIN books k ON k.ref = b.ref
UNION ALL
SELECT k.ref, NULL, k.amount
FROM books k
LEFT JOIN bank b ON b.ref = k.ref
WHERE b.ref IS NULL
ORDER BY ref;
```

EXERCISE 2

A sales grid with zeros

Report total sales for **every** combination of region and month listed in the `regions` and `months` tables — including combinations with no sales, which show `0`. Sales for a region or month that is not listed are ignored, and a NULL amount adds nothing.

Tables: `regions(name)`, `months(month)` and `sales(id, region, month, amount)`.

regions.name
North
South

months.month
2024-01
2024-02
2024-03

sales.id	region	month	amount
1	North	2024-01	100
2	North	2024-01	50
3	North	2024-03	70
4	South	2024-02	30
5	East	2024-01	999

Output: columns `region`, `month`, `total`, ordered by `region`, then `month`.

```
region | month | total
North | 2024-01 | 150
North | 2024-02 | 0
North | 2024-03 | 70
South | 2024-01 | 0
South | 2024-02 | 30
South | 2024-03 | 0
```

INPUT 1

```
CREATE TABLE regions (name VARCHAR(20));
CREATE TABLE months (month VARCHAR(7));
CREATE TABLE sales (id INT PRIMARY KEY,
region VARCHAR(20), month VARCHAR(7),
amount INT);
INSERT INTO regions VALUES ('North'),
('South');
INSERT INTO months VALUES ('2024-01'),
('2024-02'), ('2024-03');
INSERT INTO sales VALUES (1, 'North',
'2024-01', 100), (2, 'North', '2024-01',
50), (3, 'North', '2024-03', 70), (4,
'South', '2024-02', 30), (5, 'East',
'2024-01', 999);
```

OUTPUT 1

```
region | month | total
North | 2024-01 | 150
North | 2024-02 | 0
North | 2024-03 | 70
South | 2024-01 | 0
South | 2024-02 | 30
South | 2024-03 | 0
```

REFERENCE SOLUTION

```sql
SELECT r.name AS region, m.month, COALESCE(SUM(s.amount), 0) AS total
FROM regions r
CROSS JOIN months m
LEFT JOIN sales s ON s.region = r.name AND s.month = m.month
GROUP BY r.name, m.month
ORDER BY region, m.month;
```

QUIZ

1. What happens when you run `SELECT * FROM a FULL OUTER JOIN b ON b.id = a.id;` on MySQL 8?

A. It returns matched rows plus the unmatched rows of both tables

B. A syntax error — MySQL does not support FULL OUTER JOIN

C. It runs as a LEFT JOIN

D. It runs as a CROSS JOIN

2. Why is `… LEFT JOIN … UNION ALL … RIGHT JOIN … WHERE left_key IS NULL` preferred over `… LEFT JOIN … UNION … RIGHT JOIN …` for emulating a full outer join?

A. `UNION` is not supported in MySQL

B. `UNION` removes duplicate rows, so two genuinely identical result rows would be collapsed into one

C. `UNION ALL` sorts the result

D. There is no difference

3. `colours` has 4 rows and `sizes` has 5. How many rows does `colours CROSS JOIN sizes` return?

A. 9

B. 20

C. 5

D. 4

4. In MySQL, what does `SELECT * FROM orders JOIN customers;` (no `ON`) return?

A. An error: `JOIN` requires `ON`

B. Every order paired with every customer — a Cartesian product

C. Orders joined on columns with the same name

D. Only the orders that have a customer

ANSWER KEY · FULL OUTER AND CROSS JOIN — BOTH SIDES, AND EVERY PAIR

1. B MySQL has no full outer join. Note the nastier variant: `a FULL JOIN b ON …` (without `OUTER`) is accepted, because `FULL` is read as an alias for `a` — it silently runs as an inner join. PostgreSQL, SQL Server and Oracle support `FULL OUTER JOIN` directly.

2. B The `UNION` version relies on duplicate removal to merge the matched rows that both halves return — and it also merges real rows that happen to be identical. Restricting the second half to unmatched rows and using `UNION ALL` is exact, and skips the de-duplication work.

3. B A cross join pairs every row with every row: 4 × 5 = 20. If either table were empty, the result would be empty.

4. B In MySQL `JOIN`, `INNER JOIN` and `CROSS JOIN` are synonyms, and without a condition the result is the Cartesian product. PostgreSQL rejects a `JOIN` without `ON`; joining on shared column names is `NATURAL JOIN`.

LESSON 4 · 15 MIN

Self joins — a table joined to itself

Some questions compare rows of a single table with each other. Which employees earn more than their own manager? Which customers live in the same city? On which days was it warmer than the day before? The data needed for both sides of each comparison is in one table, so the join is a **self join**: the same table named twice, under two aliases, and joined like any other pair of tables. Nothing new is needed in SQL — only the habit of thinking of the two aliases as two different tables that happen to hold the same rows.

Hierarchies: employees and managers

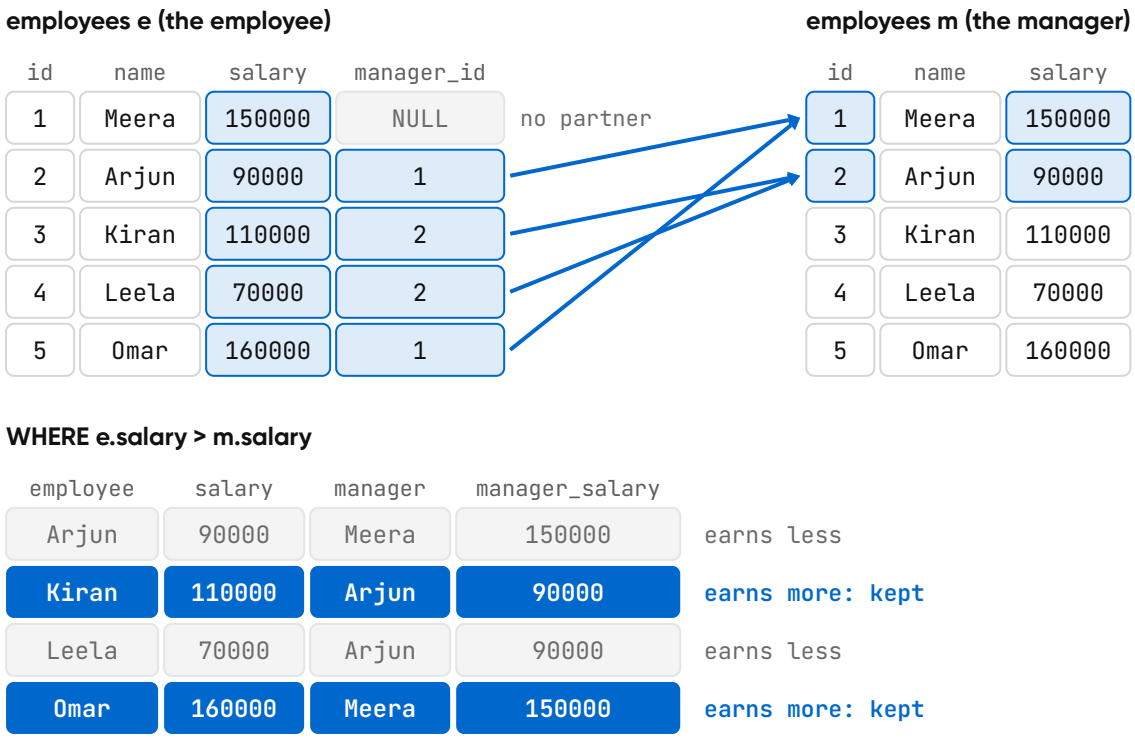
A common design stores a hierarchy in one table, each row pointing at its parent:

```
CREATE TABLE employees (id INT PRIMARY KEY, name VARCHAR(40), salary INT, mana
INSERT INTO employees VALUES
  (1, 'Meera', 150000, NULL),
  (2, 'Arjun',  90000, 1),
  (3, 'Kiran', 110000, 2),
  (4, 'Leela',  70000, 2),
  (5, 'Omar',  160000, 1);
```

`manager_id` is a foreign key into the same table. To show each employee next to their manager's name, join `employees` as `e` (the employee) to `employees` as `m` (the manager):

```
SELECT e.name AS employee, m.name AS manager
FROM employees e
JOIN employees m ON m.id = e.manager_id
ORDER BY e.name;
```

A self join: one table, read as employee and as manager



1. A self join reads one table twice under two aliases: e plays the employee, m the manager. They are the same rows; only the roles differ.
2. Each employee's manager_id points at the m row with that id. Meera's manager_id is NULL, and NULL equals nothing, so her row has no partner and an inner self join drops her.
3. Each arrow is one result row: the employee beside the manager it points at, 4 rows sorted by employee. A LEFT JOIN would add Meera with a NULL manager.
4. With both rows side by side, comparing them is a WHERE clause. Only Kiran (over Arjun) and Omar (over Meera) earn more than their manager.

Read the `ON` clause out loud: "the manager row is the one whose id equals the employee's `manager_id`". Getting that direction backwards (`e.id = m.manager_id`) is the classic mistake: it swaps the roles and lists each manager under their report's name.

Meera is missing: her `manager_id` is NULL, which matches nothing. If the report must list everyone, use a left join and label the top:

```sql
SELECT e.name AS employee, COALESCE(m.name, '(none)') AS manager
FROM employees e
LEFT JOIN employees m ON m.id = e.manager_id
ORDER BY e.name;
```

Comparing a row with its partner

Once the two rows sit side by side, any comparison is a `WHERE` clause. The interview classic — employees who earn more than their manager:

```
SELECT e.name AS employee, e.salary, m.name AS manager, m.salary AS manager_sa
FROM employees e
JOIN employees m ON m.id = e.manager_id
WHERE e.salary > m.salary
ORDER BY e.name;
```

employee	salary	manager	manager_salary
Kiran	110000	Arjun	90000
Omar	160000	Meera	150000

The same shape answers "employees hired before their manager", "products cheaper than the product they replaced" or "tasks due before the task they depend on".

Counting through a self join works too — the number of direct reports per manager:

```
SELECT m.name AS manager, COUNT(e.id) AS reports
FROM employees m
LEFT JOIN employees e ON e.manager_id = m.id
GROUP BY m.id, m.name
ORDER BY reports DESC, m.name;
```

Here `m` is on the left because every employee is a potential manager, and the left join gives zero to the people who manage nobody.

Going up more than one level

Each extra level of the hierarchy is one more alias:

```
SELECT e.name AS employee, m.name AS manager, g.name AS managers_manager
FROM employees e
LEFT JOIN employees m ON m.id = e.manager_id
LEFT JOIN employees g ON g.id = m.manager_id
ORDER BY e.name;
```

That works for a fixed depth. A hierarchy of unknown depth — "everyone under Meera, however far down" — needs a recursive CTE, which comes in a later module.

Pairs of rows

Customers in the same city:

```sql
SELECT a.name AS customer_a, b.name AS customer_b, a.city
FROM customers a
JOIN customers b ON b.city = a.city AND a.id < b.id
ORDER BY a.city, customer_a, customer_b;
```

Why `a.id < b.id`? Without it, every customer matches themselves (same city, after all), and every real pair appears twice, as (Asha, Chen) and (Chen, Asha). `a.id <> b.id` removes the self-pairs but keeps both orderings. The strict `<` keeps exactly one ordering of each pair. A city with *n* customers gives *n(n − 1)/2* pairs. Customers whose city is NULL never pair, because `NULL = NULL` is not true.

The same trick finds duplicate records — two rows with the same email but different ids:

```sql
SELECT a.id AS keep_id, b.id AS duplicate_id, a.email
FROM users a
JOIN users b ON b.email = a.email AND a.id < b.id
ORDER BY a.email, b.id;
```

Comparing with the previous day

Daily readings, one row per date — on which days was it warmer than the day before?

```sql
SELECT w.day, w.temp, p.temp AS previous_temp
FROM weather w
JOIN weather p ON w.day = DATE_ADD(p.day, INTERVAL 1 DAY)
WHERE w.temp > p.temp
ORDER BY w.day;
```

The partner of each row is the row dated exactly one day earlier. If a date is missing from the table, the day after the gap has no partner and drops out — "the day before" is a calendar fact, not "the previous row". Comparing by subtracting dates (`w.day - 1`) does not work in

MySQL: arithmetic on a `DATE` treats it as the number 20240102, so subtracting 1 from 20240101 gives 20240100, not December 31st. Use `DATE_ADD` / `DATE_SUB` or `DATEDIFF(w.day, p.day) = 1`. In PostgreSQL, `p.day + 1` on a `date` is correct, and in SQL Server the function is `DATEADD(day, 1, p.day)`.

Window functions (`LAG`) compare with the previous *row* instead and are often simpler; they come in a later module. Self joins remain the answer when the partner is defined by a condition rather than by position, and interviewers still ask for them.

Performance note

A self join is executed like any join: the database looks up partners using indexes on the join columns. An index on `manager_id`, or on `(city)`, keeps a large self join fast. A pair query with only `a.id < b.id` and no equality condition is close to a cross join — $n^2/2$ rows — so make sure an equality narrows it down.

Pitfalls

- Reversing the `ON` direction and swapping the roles of the two aliases.
- Forgetting the top of the hierarchy, whose parent key is NULL.
- Pair queries without `a.id < b.id`: self-pairs and every pair twice.
- Date arithmetic with `- 1` in MySQL instead of `DATE_ADD` or `DATEDIFF`.
- Aliases like `t1` / `t2` that hide which row plays which role.

Key takeaways

- A self join names one table twice with two aliases and joins them like two tables.
- Name the aliases by role (`e`, `m`) and read `ON` aloud to check the direction.
- Use `LEFT JOIN` when rows with no partner (the top manager) must stay.
- `a.id < b.id` produces each pair once and never a row with itself.
- For "the day before", join on `DATE_ADD(p.day, INTERVAL 1 DAY)`; gaps in dates are respected.

EXERCISE 1

Out-earning the manager

Find the employees who earn strictly more than their own manager.

Table: `employees(id, name, salary, manager_id)`; `manager_id` is the `id` of the employee's manager, NULL for the top.

id	name	salary	manager_id
1	Meera	150000	NULL
2	Arjun	90000	1
3	Kiran	110000	2
4	Leela	70000	2
5	Omar	160000	1

Output: columns `employee`, `salary`, `manager`, `manager_salary`, ordered by `employee`.

```
employee | salary | manager | manager_salary
Kiran | 110000 | Arjun | 90000
Omar | 160000 | Meera | 150000
```

INPUT 1

```
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), salary INT,
manager_id INT);
INSERT INTO employees VALUES (1,
'Meera', 150000, NULL), (2, 'Arjun',
90000, 1), (3, 'Kiran', 110000, 2), (4,
'Leela', 70000, 2), (5, 'Omar', 160000,
1);
```

OUTPUT 1

```
employee | salary | manager |
manager_salary
Kiran | 110000 | Arjun | 90000
Omar | 160000 | Meera | 150000
```

REFERENCE SOLUTION

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

EXERCISE 2

Warmer than the day before

Find the days that were warmer than the **calendar** day before. A day whose previous date has no reading is not compared. The rows may be stored in any order.

Table: `weather(recorded_on, temp)` ; one row per date, `recorded_on` is a `DATE` .

recorded_on	temp
2024-01-01	10
2024-01-02	12
2024-01-03	11
2024-01-04	15
2024-01-06	20
2024-01-07	25

Output: columns `recorded_on` , `temp` , `previous_temp` , ordered by `recorded_on` .

```
recorded_on | temp | previous_temp
2024-01-02 | 12 | 10
2024-01-04 | 15 | 11
2024-01-07 | 25 | 20
```

INPUT 1

```
CREATE TABLE weather (recorded_on DATE
PRIMARY KEY, temp INT);
INSERT INTO weather VALUES ('2024-01-
01', 10), ('2024-01-02', 12), ('2024-01-
03', 11), ('2024-01-04', 15), ('2024-01-
06', 20), ('2024-01-07', 25);
```

OUTPUT 1

```
recorded_on | temp | previous_temp
2024-01-02 | 12 | 10
2024-01-04 | 15 | 11
2024-01-07 | 25 | 20
```

REFERENCE SOLUTION

```sql
SELECT w.recorded_on, w.temp, p.temp AS previous_temp
FROM weather w
JOIN weather p ON w.recorded_on = DATE_ADD(p.recorded_on, INTERVAL 1 DAY)
WHERE w.temp > p.temp
ORDER BY w.recorded_on;
```

QUIZ

1. In `FROM employees e JOIN employees m ON …` , `e` is the employee and `m` the manager. Which `ON` condition is right?

A. `e.id = m.manager_id`

B. `m.id = e.manager_id`

C. `e.id = m.id`

D. `e.manager_id = m.manager_id`

2. `customers` has three customers in Pune. How many rows does this return for Pune?

```sql
SELECT a.name, b.name
FROM customers a
JOIN customers b ON b.city = a.city AND a.id <> b.id;
```

A. 3

B. 6

C. 9

D. 1

3. Why does `FROM employees JOIN employees ON …` fail?

A. A table cannot be joined to itself

B. Both occurrences have the same name, so at least one needs an alias — MySQL reports "Not unique table/alias"

C. Self joins require `LEFT JOIN`

D. Self joins need `USING`

ANSWER KEY · SELF JOINS — A TABLE JOINED TO ITSELF

1. B The manager's row is the one whose `id` is the employee's `manager_id`. `e.id = m.manager_id` swaps the roles (each row in `e` would be a manager of `m`), `e.id = m.id` pairs each row with itself, and the last pairs colleagues who share a manager.

2. B `<>` removes the self-pairs but keeps both orderings, so each of the 3 pairs appears twice: 6 rows. Without the condition it would be 9 (every row with every row, including itself); with `a.id < b.id` it would be 3.

3. B A self join is a normal join; it only needs distinct names for the two copies, and every column must then be qualified with the alias it belongs to.

LESSON 5 · 16 MIN

Joining three or more tables

Real schemas have more than two tables, and real questions cross several of them: which teacher teaches each course a student takes; how much revenue each product category made from orders placed in 2024; which customers bought from which supplier. SQL handles any number of tables with the same tool — one more `JOIN … ON` per table — but three new hazards appear once there are more than two: junction tables that must be crossed in two steps, outer joins undone by a later inner join, and the **fan-out** that silently multiplies totals. This lesson covers each.

A chain of joins

A school schema:

```
CREATE TABLE students    (id INT PRIMARY KEY, name VARCHAR(40));
CREATE TABLE teachers    (id INT PRIMARY KEY, name VARCHAR(40));
CREATE TABLE courses     (id INT PRIMARY KEY, title VARCHAR(40), teacher_id IN
CREATE TABLE enrollments (student_id INT, course_id INT, grade INT);
```

To list each student's courses with the teacher's name:

```
SELECT s.name AS student, c.title AS course, t.name AS teacher
FROM students s
JOIN enrollments e ON e.student_id = s.id
JOIN courses c     ON c.id = e.course_id
JOIN teachers t    ON t.id = c.teacher_id
ORDER BY s.name, c.title;
```

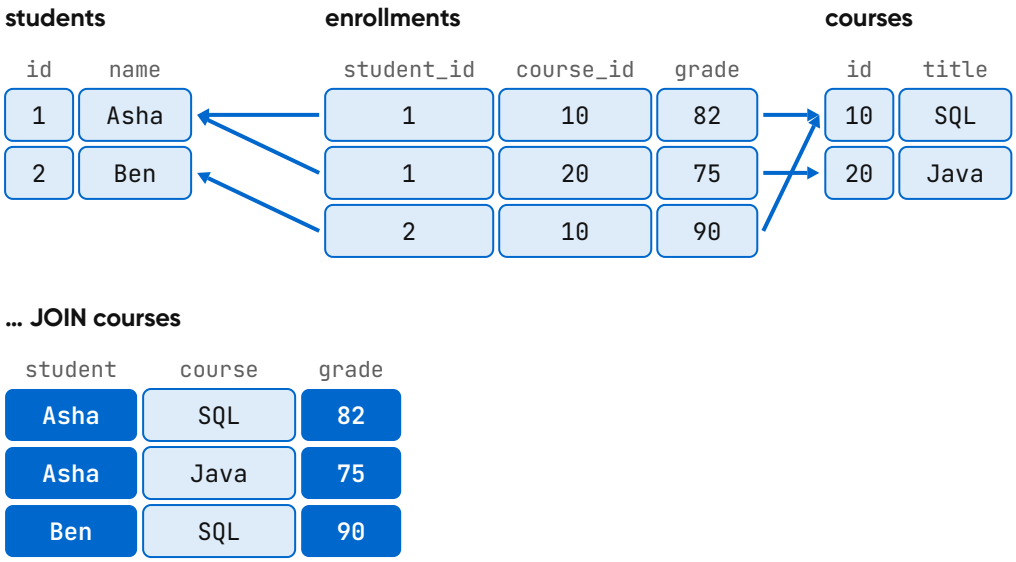
Read it top to bottom as a pipeline. Start with students. Join enrollments: now there is one row per (student, enrollment). Join courses: each enrollment row gains its course. Join teachers: each course row gains its teacher. Each `ON` links the new table to one already present — the query fails, or worse, cross joins, if a table is added without a condition connecting it.

For inner joins the textual order is for the reader, not the database: the optimizer reorders inner joins freely and returns the same rows. Write them in the order that tells the story, usually following the foreign keys.

Many-to-many through a junction table

Students and courses are many-to-many: one student takes many courses, one course has many students. A relational table cannot hold a list in a column, so the relationship lives in its own table — `enrollments`, one row per (student, course) pair, often with facts about the pair (a grade, a date). Going from a student to their courses always crosses it:

Many-to-many: the pairs live in a junction table



1. A student takes many courses and a course has many students. No column can hold a list, so each (student, course) pair is a row of enrollments, pointing at both sides.
2. The first join gives each student one row per enrollment: Asha 2 rows, Ben 1. The course is still only an id, and the junction's own column, grade, comes along.
3. The second join gives each of those rows its course, through course_id. Each ON links the new table to one already present; two joins cross the junction table.

Two joins, and the junction table's columns (here `grade`) are available along the way. Counting the students of each course is a join plus a group:

```
SELECT c.title AS course, COUNT(e.student_id) AS students
FROM courses c
LEFT JOIN enrollments e ON e.course_id = c.id
GROUP BY c.id, c.title
ORDER BY students DESC, c.title;
```

Mixing LEFT and INNER joins

Now suppose some courses have no teacher yet (`teacher_id` is NULL) and the timetable must still show them. The fourth join must be a left join:

```
SELECT s.name AS student, c.title AS course, t.name AS teacher
FROM students s
JOIN enrollments e ON e.student_id = s.id
JOIN courses c     ON c.id = e.course_id
LEFT JOIN teachers t ON t.id = c.teacher_id
ORDER BY s.name, c.title;
```

The opposite mistake is the one that hides in long queries. Start from students with a left join, so that students with no enrollments appear — and then add courses with a plain `JOIN`:

```
SELECT s.name, c.title
FROM students s
LEFT JOIN enrollments e ON e.student_id = s.id
JOIN courses c ON c.id = e.course_id      -- removes the unenrolled students (
ORDER BY s.name, c.title;
```

For an unenrolled student the left join produced a row with `e.course_id` NULL; the inner join's condition `c.id = NULL` is unknown, and the row is dropped. The rule: **once a table is optional, everything joined through it must be left-joined too.** Change the last join to `LEFT JOIN` and the unenrolled students come back with a NULL course.

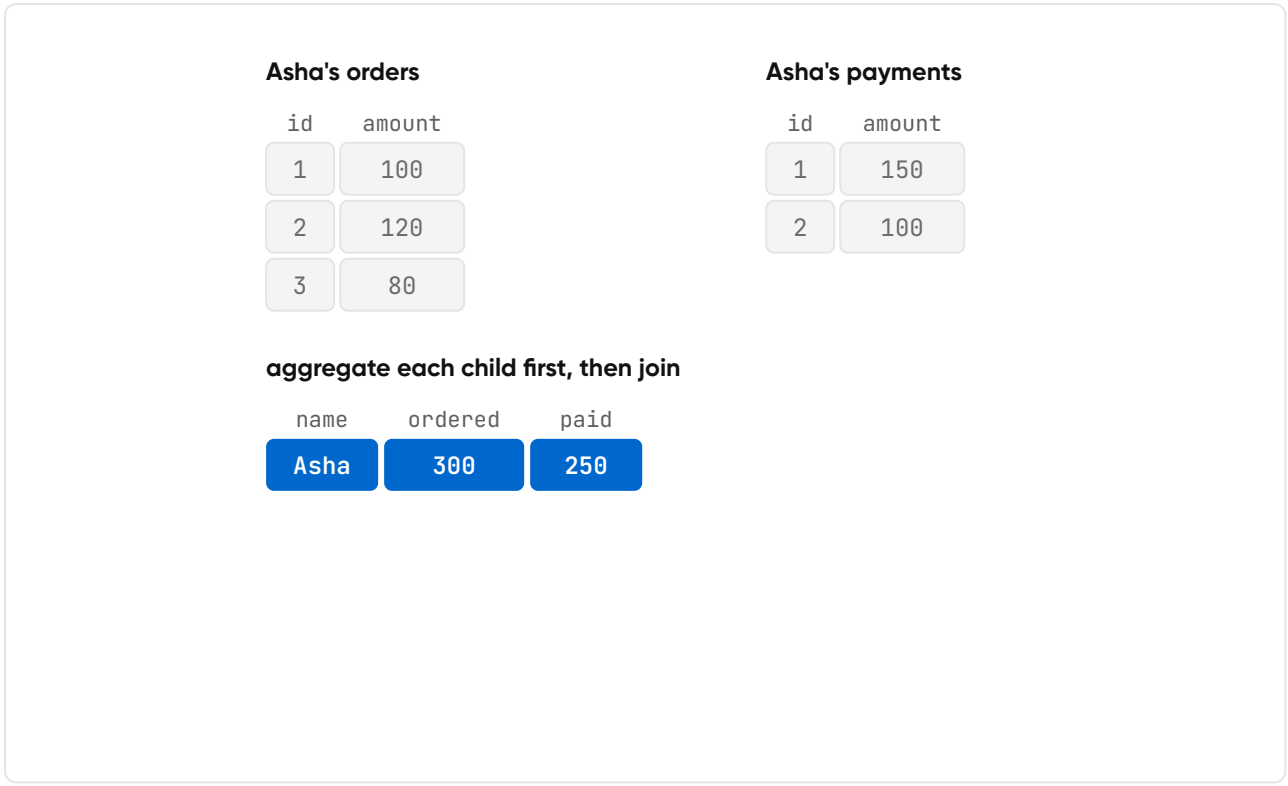
(Standard SQL also allows nesting — `LEFT JOIN (enrollments e JOIN courses c ON …) ON e.student_id = s.id` — which means "the enrollment-with-course pairs, optionally". It is correct but rarely written; a chain of left joins is easier to read.)

Fan-out: when joins multiply totals

Each one-to-many join multiplies rows. With one child table that is expected. With two *independent* child tables it is a bug:

```
-- Customer 1 has 3 orders (total 300) and 2 payments (total 250).
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.id, c.name;
```

Fan-out: two independent child tables multiply each other



1. Asha has 3 orders, totalling 300, and 2 payments, totalling 250. The two tables are not related to each other — only to her.
2. Joining both to the customer pairs every order row with every payment row: 3 × 2 = 6 rows, though she has only 5 child rows.
3. Summing now counts each order 2 times and each payment 3 times: 600 ordered and 750 paid. No error, plausible numbers — which is why the bug survives review.
4. The fix: total each child first, one row per customer each, then join the totals. 300 and 250, the real figures.

Orders and payments are not related to each other, so every order row was paired with every payment row: 3 × 2 = 6 rows. Each order was counted twice (once per payment) and each payment three times. The query runs without error and the numbers look plausible — which is why this bug survives code review.

Fixes, best first:

- **aggregate each child first, then join the totals** — a derived table per child (the subqueries module shows it), so each customer meets at most one order-total row and one payment-total row;
- for counts, `COUNT(DISTINCT o.id)` undoes the multiplication, but there is no safe `SUM(DISTINCT …)` — two different orders of 100 would collapse into one;
- run two separate queries.

Fan-out is not limited to two children. Joining `orders` to `order_items` and then summing `orders.shipping_fee` counts the fee once per item. Before summing a column, ask which table it lives in and whether the joins after that table can repeat its rows.

Filtering in a long join

Filters still go in `WHERE` when they apply to a table that must match anyway:

```sql
SELECT cat.name AS category, SUM(i.qty * p.price) AS revenue
FROM orders o
JOIN order_items i ON i.order_id = o.id
JOIN products p    ON p.id = i.product_id
JOIN categories cat ON cat.id = p.category_id
WHERE o.order_date ⩾ '2024-01-01' AND o.order_date < '2025-01-01'
GROUP BY cat.id, cat.name
ORDER BY revenue DESC, category;
```

The date range is written as a half-open interval on the raw column rather than `YEAR(o.order_date) = 2024`; both return the same rows, but the first can use an index on `order_date` (the indexes module explains why). If a category with no 2024 sales must still appear, start from categories and left-join the rest — and then the date filter belongs in the `ON` of the orders join, for the reason in the outer-joins lesson.

Pitfalls

- A table added without a condition linking it to the rest (a cross join).
- An inner join after a left join on the optional side.
- Summing across two independent one-to-many joins (fan-out).
- `SUM(DISTINCT …)` as a fan-out fix — it merges equal values from different rows.
- Grouping by a name instead of a key when several tables have names.

Key takeaways

- Each `JOIN … ON` adds one table to the rows built so far; read the chain as a pipeline.
- Inner join order does not change the result; outer join order does.
- Many-to-many relationships go through a junction table and need two joins.
- Everything joined through an optional table must be left-joined.
- Two independent child tables multiply each other's rows; aggregate each before joining.

EXERCISE 1

Student timetable

List each enrolment as the student's name, the course title and the teacher's name. A course with no teacher (NULL `teacher_id`, or one that matches no teacher) still appears, with a NULL teacher. Enrolments in courses that do not exist are not listed, and students with no enrolments are not listed.

Tables: `students(id, name)`, `teachers(id, name)`, `courses(id, title, teacher_id)`, `enrollments(student_id, course_id)`.

students.id	name
1	Anu
2	Bilal
3	Chitra

teachers.id	name
1	Dr Rao
2	Ms Iyer

courses.id	title	teacher_id
101	Databases	1
102	Networks	2
103	Statistics	NULL

student_id	course_id
1	101
1	103
2	102
2	101

Output: columns `student`, `course`, `teacher`, ordered by `student`, then `course`.

```
student | course | teacher
Anu | Databases | Dr Rao
Anu | Statistics | NULL
Bilal | Databases | Dr Rao
Bilal | Networks | Ms Iyer
```

INPUT 1

```
CREATE TABLE students (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE teachers (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE courses (id INT PRIMARY
KEY, title VARCHAR(40), teacher_id INT);
CREATE TABLE enrollments (student_id
INT, course_id INT);
INSERT INTO students VALUES (1, 'Anu'),
(2, 'Bilal'), (3, 'Chitra');
INSERT INTO teachers VALUES (1, 'Dr
Rao'), (2, 'Ms Iyer');
INSERT INTO courses VALUES (101,
'Databases', 1), (102, 'Networks', 2),
(103, 'Statistics', NULL);
INSERT INTO enrollments VALUES (1, 101),
(1, 103), (2, 102), (2, 101);
```

OUTPUT 1

```
student | course | teacher
Anu | Databases | Dr Rao
Anu | Statistics | NULL
Bilal | Databases | Dr Rao
Bilal | Networks | Ms Iyer
```

REFERENCE SOLUTION

```sql
SELECT s.name AS student, c.title AS course, t.name AS teacher
FROM students s
JOIN enrollments e ON e.student_id = s.id
JOIN courses c ON c.id = e.course_id
LEFT JOIN teachers t ON t.id = c.teacher_id
ORDER BY student, course;
```

EXERCISE 2

Revenue by category in 2024

For orders placed in 2024, report the units sold and the revenue (`qty × price`) per product category. Categories with no 2024 sales are not listed.

Tables: `orders(id, order_date)`, `order_items(order_id, product_id, qty)`, `products(id, name, category_id, price)`, `categories(id, name)`.

categories.id	name
1	Books
2	Games
3	Music

products.id	name	category_id	price
1	SQL Guide	1	500
2	Chess	2	300
3	Novel	1	200
4	Vinyl	3	800

orders.id	order_date
1	2024-01-10
2	2024-03-05
3	2023-12-31
4	2024-12-31

order_id	product_id	qty
1	1	2
1	2	1
2	3	3
3	4	5
4	2	2

Output: columns `category`, `units`, `revenue`, ordered by `revenue` descending, then `category`.

```
category | units | revenue
Books | 5 | 1600
Games | 3 | 900
```

INPUT 1

```
CREATE TABLE categories (id INT PRIMARY KEY, name VARCHAR(40));
CREATE TABLE products (id INT PRIMARY KEY, name VARCHAR(40), category_id INT, price INT);
CREATE TABLE orders (id INT PRIMARY KEY, order_date DATE);
CREATE TABLE order_items (order_id INT, product_id INT, qty INT);
INSERT INTO categories VALUES (1, 'Books'), (2, 'Games'), (3, 'Music');
INSERT INTO products VALUES (1, 'SQL Guide', 1, 500), (2, 'Chess', 2, 300), (3, 'Novel', 1, 200), (4, 'Vinyl', 3, 800);
INSERT INTO orders VALUES (1, '2024-01-10'), (2, '2024-03-05'), (3, '2023-12-31'), (4, '2024-12-31');
INSERT INTO order_items VALUES (1, 1, 2), (1, 2, 1), (2, 3, 3), (3, 4, 5), (4, 2, 2);
```

OUTPUT 1

```
category | units | revenue
Books | 5 | 1600
Games | 3 | 900
```

REFERENCE SOLUTION

```sql
SELECT cat.name AS category, SUM(i.qty) AS units, SUM(i.qty * p.price) AS revenue
FROM orders o
JOIN order_items i ON i.order_id = o.id
JOIN products p ON p.id = i.product_id
JOIN categories cat ON cat.id = p.category_id
WHERE o.order_date ⩾ '2024-01-01' AND o.order_date < '2025-01-01'
GROUP BY cat.id, cat.name
ORDER BY revenue DESC, category;
```

QUIZ

1. What does this return for a student who has no enrolments?

```sql
SELECT s.name, c.title
FROM students s
LEFT JOIN enrollments e ON e.student_id = s.id
JOIN courses c ON c.id = e.course_id;
```

A. One row with a NULL title

B. No row — the inner join on `e.course_id`, which is NULL for that student, removes it

C. An error

D. One row per course

2. Customer Asha has 3 orders totalling 300 and 2 payments totalling 250. What does this report for her?

```sql
SELECT SUM(o.amount) AS ordered, SUM(p.amount) AS paid
FROM customers c
JOIN orders o ON o.customer_id = c.id
JOIN payments p ON p.customer_id = c.id
WHERE c.name = 'Asha';
```

A. 300 and 250

B. 600 and 750

C. 900 and 500

D. 300 and 750

3. Students and courses are many-to-many. How is that stored and queried?

A. A `course_ids` column in `students` holding a list

B. A junction table such as `enrollments(student_id, course_id)`, joined from students and then to courses

C. A `student_id` column in `courses`

D. A cross join of students and courses

4. Does rearranging the order of three inner joins change the result?

A. Yes, the first table always determines the rows

B. No; inner joins can be reordered freely, and the optimizer chooses its own order

C. Only if the tables have different sizes

D. Only in MySQL

ANSWER KEY · JOINING THREE OR MORE TABLES

1. B The left join keeps the student with NULL enrolment columns, but `c.id = NULL` is unknown, so the following inner join drops the row. Everything joined through an optional table must be left-joined too.

2. B Orders and payments are independent children, so the joins produce 3 × 2 = 6 rows: each order is repeated twice (600) and each payment three times (750). Aggregate each child separately before joining.

3. B A relational column holds one value, so the pairs live in their own table with one foreign key to each side. A `student_id` in `courses` would allow only one student per course; a cross join pairs everyone with everything.

4. B Inner joins are associative and commutative, so the same rows come back (in an unspecified order unless there is an `ORDER BY`). Outer joins are different: their order decides which rows are preserved.

LESSON 6 · 15 MIN

Semi joins and anti joins — "has a match" and "has none"

Two questions come up constantly and look like joins but are not quite: "which customers have placed *at least one* order?" and "which customers have placed *no* orders?". The answer lists customers only — no order columns, and each customer once. The first is a **semi join**, the second an **anti join**. SQL has no keyword for either; you write them with `EXISTS`, `IN`, or a left join, and the choice matters for duplicates and, in the anti-join case, for correctness when NULLs are around. Anti joins in particular are among the most asked SQL interview questions.

Why a plain join is not a semi join

```
SELECT c.id, c.name
FROM customers c
JOIN orders o ON o.customer_id = c.id
ORDER BY c.id;
```

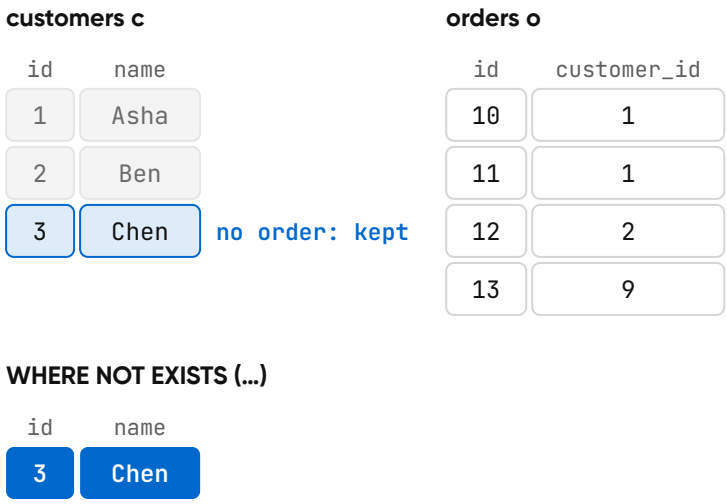
id	name
1	Asha
1	Asha
2	Ben

Asha has two orders, so she appears twice. The join answers "list the (customer, order) pairs", and the question was "list the customers". `SELECT DISTINCT c.id, c.name` removes the repeats, but it does so by building all the pairs first and then sorting or hashing them away — wasted work when a customer has thousands of orders.

Semi join with EXISTS

```sql
SELECT c.id, c.name
FROM customers c
WHERE EXISTS (
  SELECT 1 FROM orders o WHERE o.customer_id = c.id
)
ORDER BY c.id;
```

Semi and anti joins: does a match exist?



1. A plain JOIN answers "list the (customer, order) pairs": Asha has 2 orders, so she appears 2 times. The question was "list the customers".
2. EXISTS asks only whether one matching row exists. For Asha the search stops at order 10; order 11 is never needed.
3. So each customer appears at most once: Asha and Ben. Chen, with no order, is left out. That is a semi join.
4. NOT EXISTS keeps exactly the customers the semi join left out: Chen. That is the anti join, and unlike NOT IN it is safe when orders.customer_id holds a NULL.

`EXISTS (subquery)` is true if the subquery returns at least one row. The subquery refers to `c.id` from the outer query — it is **correlated**, evaluated for each customer (the subqueries module covers correlation in depth). The `SELECT 1` is a convention: `EXISTS` looks only at whether a row exists, so the select list is ignored; `SELECT *` or `SELECT o.id` would do the same. The database can stop at the first matching order — it never needs the rest.

Extra conditions on the matching row go inside:

```sql
-- customers with at least one order of 300 or more
SELECT c.id, c.name
FROM customers c
WHERE EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.id AND o.amount
ORDER BY c.id;
```

Semi join with IN

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

Same result. `IN` compares one value against the set the subquery returns; duplicates in that set do not duplicate the output. `IN` reads naturally when the link is a single column and the subquery stands alone. `EXISTS` is more flexible: it can correlate on several columns and add any condition. MySQL 8 turns both into a semi-join plan, so choose by readability.

Anti join, three ways

Customers with no orders.

NOT EXISTS — the recommended form:

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

LEFT JOIN ... IS NULL — keep every customer, then keep only those whose order side stayed empty:

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

The `WHERE` clause here is the deliberate exception to the outer-joins lesson's rule: it filters on the right table precisely to keep the NULL-filled rows. Test a column that is never NULL in a real match — the right table's primary key, or the join column itself. Testing `o.amount IS NULL` would also return customers who have an order whose amount happens to be NULL.

NOT IN — the one to be careful with:

```sql
SELECT c.id, c.name
FROM customers c
WHERE c.id NOT IN (SELECT o.customer_id FROM orders o)
ORDER BY c.id;
```

This returns the right answer only while `orders.customer_id` contains no NULL. Insert a single order with `customer_id` NULL — a guest checkout, say — and the query returns **no rows at all**. The reason is three-valued logic: `3 NOT IN (1, 2, NULL)` expands to `3 <> 1 AND 3 <> 2 AND 3 <> NULL`, and `3 <> NULL` is unknown, which makes the whole `AND` unknown, and `WHERE` keeps only true. The final lesson of the subqueries module is devoted to this trap. For now: prefer `NOT EXISTS` for anti joins, and if you write `NOT IN`, add `WHERE o.customer_id IS NOT NULL` inside.

Anti join with a condition

"Customers with no order in 2024" — the date condition is part of what counts as a match, so it goes **inside** the `NOT EXISTS`, or into the `ON` of the left join:

```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;

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

Putting the date range in the outer `WHERE` of the left-join version would be contradictory — `o.order_date` is NULL on exactly the rows `o.id IS NULL` keeps — and the query would return nothing.

How MySQL runs them

MySQL 8 recognises `IN` and `EXISTS` subqueries as semi joins and, since 8.0.17, `NOT IN` and `NOT EXISTS` as antijoins, choosing among strategies (first match, materialisation, duplicate weedout) by cost. An index on the inner join column — `orders(customer_id)` — is what makes any of them fast. PostgreSQL and SQL Server likewise plan `NOT EXISTS` and `LEFT JOIN … IS NULL` as anti joins; `NOT IN` is often planned worse, precisely because of its NULL semantics. In interviews, say you would write `NOT EXISTS` and explain why `NOT IN` is fragile.

Pitfalls

- A plain join where a semi join was meant, then `DISTINCT` to hide the duplicates.
- `NOT IN` against a column that can contain NULL.
- Testing a nullable right-side column in the `LEFT JOIN … IS NULL` pattern.
- An anti join's extra condition placed in the outer `WHERE` instead of inside the subquery or the `ON`.

- Selecting columns of the inner table in a semi join — there is no single matching row to take them from.

Key takeaways

- Semi join: rows with at least one match, each once — `EXISTS` or `IN`.
- Anti join: rows with no match — `NOT EXISTS`, or `LEFT JOIN … WHERE right_key IS NULL`.
- `NOT IN` returns nothing as soon as the subquery yields a NULL.
- Conditions on what counts as a match go inside the subquery or into `ON`.
- MySQL 8 plans all of these as semi- and antijoins; index the inner join column.

EXERCISE 1

Customers with a big order

List the customers who placed **at least one** order of 300 or more. Each customer appears once, however many big orders they placed.

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	300
12	1	500
13	2	120
14	4	300
15	9	900

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

```
id | name
1 | Asha
4 | Dina
```

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, 300), (12, 1, 500), (13, 2,
120), (14, 4, 300), (15, 9, 900);
```

OUTPUT 1

```
id | name
1 | Asha
4 | Dina
```

REFERENCE SOLUTION

```sql
SELECT c.id, c.name
FROM customers c
WHERE EXISTS (
  SELECT 1 FROM orders o WHERE o.customer_id = c.id AND o.amount ⩾ 300
)
ORDER BY c.id;
```

EXERCISE 2

Products never sold

List the products that appear in **no** order item. Beware: some order items have a NULL `product_id` (a deleted product).

Tables: `products(id, name)` and `order_items(order_id, product_id, qty)`.

products.id	name
1	Pen
2	Ink
3	Paper
4	Stapler

order_id	product_id	qty
1	1	2
1	3	1
2	1	5

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

```
id | name
2 | Ink
4 | Stapler
```

INPUT 1

```
CREATE TABLE products (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE order_items (order_id INT,
product_id INT, qty INT);
INSERT INTO products VALUES (1, 'Pen'),
(2, 'Ink'), (3, 'Paper'), (4,
'Stapler');
INSERT INTO order_items VALUES (1, 1,
2), (1, 3, 1), (2, 1, 5);
```

OUTPUT 1

```
id | name
2 | Ink
4 | Stapler
```

REFERENCE SOLUTION

```sql
SELECT p.id, p.name
FROM products p
WHERE NOT EXISTS (
  SELECT 1 FROM order_items i WHERE i.product_id = p.id
)
ORDER BY p.id;
```

QUIZ

1. `orders.customer_id` contains the values 1, 2 and NULL. What does this return?

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

A. The customers whose id is not 1 or 2

B. No rows

C. Every customer

D. An error

2. Why is `WHERE EXISTS (…)` better than a plain join for "customers who have placed an order"?

A. A join cannot reach the orders table

B. A join returns a customer once per order, so it needs `DISTINCT`; `EXISTS` returns each customer once and can stop at the first match

C. `EXISTS` returns the order columns too

D. Joins are not allowed in `WHERE`

3. In `SELECT c.* FROM customers c LEFT JOIN shipments s ON s.customer_id = c.id WHERE ___ IS NULL`, which column should fill the blank to find customers with no shipment? `shipped_on` is NULL for shipments not yet sent.

A. `s.shipped_on`

B. `s.id` (the shipments primary key)

C. `c.id`

D. Any column of `s`

4. What does the select list inside `EXISTS (SELECT 1 FROM orders WHERE …)` do?

A. It decides which value is compared

B. Nothing — `EXISTS` only checks whether any row is returned

C. It must be `*`

D. It limits the subquery to one row

ANSWER KEY · SEMI JOINS AND ANTI JOINS — "HAS A MATCH" AND "HAS NONE"

1. B For customer 3 the condition is `3 <> 1 AND 3 <> 2 AND 3 <> NULL`; the last part is unknown, so the whole condition is unknown and the row is not kept. That happens for every customer. `NOT EXISTS` gives the intended answer.

2. B A semi join's answer is the customers, not the pairs. `EXISTS` (or `IN`) states that directly and never multiplies rows. The subquery's columns are not available outside it.

3. B A real shipment always has its primary key, so `s.id IS NULL` means "no match". `s.shipped_on IS NULL` would also keep customers whose shipment exists but is unsent. `c.id` is never NULL here.

4. B `EXISTS` is true when the subquery returns at least one row, whatever its columns. `SELECT 1`, `SELECT *` and `SELECT o.id` behave the same; `SELECT 1` is the convention that signals this.

CHECKPOINT · 25 MIN

Checkpoint — Joins

This checkpoint covers the whole module: inner joins and how rows multiply in one-to-many relationships, left and right joins with counting and the ON-versus-WHERE rule, the full outer join and its MySQL emulation, cross joins for grids, self joins for hierarchies, pairs and day-to-day comparisons, chains of joins through junction tables with the fan-out problem, and semi and anti joins.

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 happens to a row with no partner in an inner join, a left join and a full outer join?
- Why is `COUNT(o.id)` right and `COUNT(*)` wrong after a left join?
- Where does a condition on the optional table go, and what happens if it is in `WHERE`?
- How do you write a full outer join in MySQL, and why `UNION ALL` rather than `UNION`?
- In a self join for pairs, what does `a.id < b.id` prevent?
- Why does an inner join after a left join undo it?
- What is fan-out, and how do you avoid double counting across two child tables?
- Why is `NOT EXISTS` safer than `NOT IN` for an anti join?

The three queries are a department report that keeps empty departments, a self join pairing employees in the same department, and an anti join for courses with no enrolments in one term.

EXERCISE 1

Department report

Report **every** department with its number of employees and its highest salary. A department with no employees shows `0` and `NULL`. Employees with no department are ignored.

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

departments.id	name
1	Engineering
2	Sales
3	Legal

employees.id	name	dept_id	salary
1	Anu	1	120000
2	Ben	1	95000
3	Chen	2	70000
4	Dev	NULL	50000
5	Esha	2	82000

Output: columns `department`, `headcount`, `top_salary`, ordered by `headcount` descending, then `department`.

```
department | headcount | top_salary
Engineering | 2 | 120000
Sales | 2 | 82000
Legal | 0 | NULL
```

INPUT 1

```
CREATE TABLE departments (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), dept_id INT,
salary INT);
INSERT INTO departments VALUES (1,
'Engineering'), (2, 'Sales'), (3,
'Legal');
INSERT INTO employees VALUES (1, 'Anu',
1, 120000), (2, 'Ben', 1, 95000), (3,
'Chen', 2, 70000), (4, 'Dev', NULL,
50000), (5, 'Esha', 2, 82000);
```

OUTPUT 1

```
department | headcount | top_salary
Engineering | 2 | 120000
Sales | 2 | 82000
Legal | 0 | NULL
```

REFERENCE SOLUTION

```sql
SELECT d.name AS department, COUNT(e.id) AS headcount, MAX(e.salary) AS top_salary
FROM departments d
LEFT JOIN employees e ON e.dept_id = d.id
GROUP BY d.id, d.name
ORDER BY headcount DESC, department;
```

EXERCISE 2

Colleagues in the same department

List every pair of employees who work in the same department, with the department's name. Each pair appears once: `employee_a` is the one with the **smaller id**. Employees with no department are never paired.

Tables: `departments(id, name)` and `employees(id, name, dept_id)`.

departments.id	name
1	Engineering
2	Sales
3	Legal

employees.id	name	dept_id
1	Anu	1
2	Ben	1
3	Chen	2
4	Dev	NULL
5	Esha	2
6	Farid	1
7	Gia	3

Output: columns `department`, `employee_a`, `employee_b`, ordered by `department`, `employee_a`, `employee_b`.

```
department | employee_a | employee_b
Engineering | Anu | Ben
Engineering | Anu | Farid
Engineering | Ben | Farid
Sales | Chen | Esha
```

INPUT 1

```
CREATE TABLE departments (id INT PRIMARY
KEY, name VARCHAR(40));
CREATE TABLE employees (id INT PRIMARY
KEY, name VARCHAR(40), dept_id INT);
INSERT INTO departments VALUES (1,
'Engineering'), (2, 'Sales'), (3,
'Legal');
INSERT INTO employees VALUES (1, 'Anu',
1), (2, 'Ben', 1), (3, 'Chen', 2), (4,
'Dev', NULL), (5, 'Esha', 2), (6,
'Farid', 1), (7, 'Gia', 3);
```

OUTPUT 1

```
department | employee_a | employee_b
Engineering | Anu | Ben
Engineering | Anu | Farid
Engineering | Ben | Farid
Sales | Chen | Esha
```

REFERENCE SOLUTION

```sql
SELECT d.name AS department, a.name AS employee_a, b.name AS employee_b
FROM employees a
JOIN employees b ON b.dept_id = a.dept_id AND a.id < b.id
JOIN departments d ON d.id = a.dept_id
ORDER BY department, employee_a, employee_b;
```

EXERCISE 3

Courses with no enrolment this term

List the courses that have **no** enrolment in the term `'2024-S2'` (they may have enrolments in other terms). Some enrolment rows have a NULL `course_id`.

Tables: `courses(id, title)` and `enrollments(student_id, course_id, term)`.

courses.id	title
101	Databases
102	Networks
103	Statistics
104	Compilers

student_id	course_id	term
1	101	2024-S2
2	101	2024-S1
3	102	2024-S1
1	103	2024-S2
2	NULL	2024-S2

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

```
id | title
102 | Networks
104 | Compilers
```

INPUT 1

```
CREATE TABLE courses (id INT PRIMARY
KEY, title VARCHAR(40));
CREATE TABLE enrollments (student_id
INT, course_id INT, term VARCHAR(10));
INSERT INTO courses VALUES (101,
'Databases'), (102, 'Networks'), (103,
'Statistics'), (104, 'Compilers');
INSERT INTO enrollments VALUES (1, 101,
'2024-S2'), (2, 101, '2024-S1'), (3,
102, '2024-S1'), (1, 103, '2024-S2'),
(2, NULL, '2024-S2');
```

OUTPUT 1

```
id | title
102 | Networks
104 | Compilers
```

REFERENCE SOLUTION

```sql
SELECT c.id, c.title
FROM courses c
WHERE NOT EXISTS (
  SELECT 1 FROM enrollments e
  WHERE e.course_id = c.id AND e.term = '2024-S2'
)
ORDER BY c.id;
```

CHECKPOINT QUESTIONS

1. Which join keeps the unmatched rows of **both** tables?

A. INNER JOIN

B. LEFT JOIN

C. FULL OUTER JOIN

D. CROSS JOIN

2. `a` has 5 rows and `b` has 0 rows. How many rows does `a LEFT JOIN b ON b.a_id = a.id` return?

A. 0

B. 5

C. An error

D. 1

3. Every customer should appear with their 2024 order count. Which is correct?

A. `LEFT JOIN orders o ON o.customer_id = c.id WHERE YEAR(o.order_date) = 2024`

B. `LEFT JOIN orders o ON o.customer_id = c.id AND o.order_date ⩾ '2024-01-01' AND o.order_date < '2025-01-01'`

C. `JOIN orders o ON o.customer_id = c.id AND YEAR(o.order_date) = 2024`

D. `RIGHT JOIN orders o ON o.customer_id = c.id`

4. What does `COALESCE(SUM(o.amount), 0)` fix after a left join?

A. Duplicate rows

B. `SUM` returning NULL for a group whose only row is the NULL-filled one

C. The ambiguous column error

D. Fan-out double counting

5. Which is the correct MySQL emulation of `bank FULL OUTER JOIN books ON books.ref = bank.ref`?

A. `bank LEFT JOIN books … UNION ALL books LEFT JOIN bank … WHERE bank.ref IS NULL`

B. `bank LEFT JOIN books … UNION ALL bank RIGHT JOIN books …`

C. `bank CROSS JOIN books`

D. `bank JOIN books … UNION ALL books JOIN bank …`

6. A cross join of 3 regions and 12 months, left-joined to `sales` and grouped by region and month, returns how many rows?

A. Only the combinations that had sales

B. 36

C. 15

D. 12

7. Why does a pair query use `a.id < b.id` rather than `a.id <> b.id`?

A. `<>` is not valid in a join

B. `<>` returns every pair twice, as (A, B) and (B, A); `<` keeps one ordering

C. `<` is faster but returns the same rows

D. `<>` pairs rows with themselves

8. `employees.manager_id` is NULL for the CEO. What happens to the CEO in `employees e JOIN employees m ON m.id = e.manager_id`?

A. Appears with a NULL manager

B. Does not appear as `e`, because `m.id = NULL` matches nothing

C. Appears as their own manager

D. Causes an error

9. A query joins `customers` to both `orders` and `payments` and sums both amounts. What is the safest fix for the inflated totals?

A. `SUM(DISTINCT o.amount)` and `SUM(DISTINCT p.amount)`

B. Aggregate orders and payments separately per customer (derived tables), then join the totals

C. Change the joins to `RIGHT JOIN`

D. Add `GROUP BY` on more columns

10. Which pattern is an anti join?

A. `WHERE EXISTS (…)`

B. `LEFT JOIN … WHERE right.id IS NULL`

C. `JOIN … ON …`

D. `CROSS JOIN`

11. `order_items.product_id` contains a NULL. Which query still lists unsold products correctly?

A. `WHERE p.id NOT IN (SELECT product_id FROM order_items)`

B. `WHERE NOT EXISTS (SELECT 1 FROM order_items i WHERE i.product_id = p.id)`

C. `WHERE p.id <> ALL (SELECT product_id FROM order_items)`

D. `JOIN order_items i ON i.product_id <> p.id`

12. What does `students s LEFT JOIN enrollments e ON … JOIN courses c ON c.id = e.course_id` do to students with no enrolments?

A. Keeps them with NULL course columns

B. Removes them, because the inner join to courses cannot match their NULL `course_id`

C. Repeats them once per course

D. Raises an error

ANSWER KEY · CHECKPOINT — JOINS

1. C A full outer join keeps matched pairs plus the unmatched rows of each side. A left join keeps only the left side's unmatched rows; an inner join keeps none; a cross join does no matching at all.

2. B Every left row appears at least once; with no partners, each appears once with NULLs for `b`'s columns. The same query as an inner join, or `a CROSS JOIN b`, returns 0.

3. B The year condition decides which orders count as a match, so it belongs in `ON`. In `WHERE` it removes the NULL rows of customers without 2024 orders; the inner join drops them outright; the right join keeps orders, not customers.

4. B `SUM` ignores NULLs and returns NULL when there is nothing else, so a customer with no orders would show NULL; `COALESCE` replaces it with 0. It does nothing about duplicates or fan-out.

5. A The first half keeps every bank row; the second adds only the books rows with no bank match. Option 2 without a filter lists every matched row twice; option 4 has no unmatched rows at all; a cross join is every pair.

6. B The cross join creates all 3 × 12 = 36 cells; the left join attaches sales without removing any cell, and grouping by the cell keeps 36 groups. That is the point of the grid pattern.

7. B Both exclude self-pairs, but `<>` keeps both orderings of each pair. `<` keeps exactly one.

8. B A NULL key never satisfies an equality. Use `LEFT JOIN` to keep the CEO with a NULL manager. (The CEO still appears as `m` for their direct reports.)

9. B Each customer then meets one order-total row and one payment-total row, so nothing multiplies. `SUM(DISTINCT …)` merges equal amounts from different rows and gives wrong totals; the other options leave the fan-out in place.

10. B Keeping only the left rows whose right side stayed NULL returns the rows with no match — an anti join, like `NOT EXISTS`. `EXISTS` is a semi join.

11. B `NOT EXISTS` asks only whether a matching row exists, and NULL never matches. `NOT IN` and `<> ALL` are the same comparison and both become unknown because of the NULL. The `<>` join pairs each product with every other item.

12. B The inner join after the left join filters out the NULL-filled rows. Make the second join a `LEFT JOIN` to keep them.
