Joins deep-dive

- INNER / LEFT / RIGHT / FULL (and how to simulate FULL in MySQL)
- CROSS / SELF joins
- Semi-join / Anti-join patterns (EXISTS / NOT EXISTS)
- Join filtering pitfalls (WHERE vs ON)
- Multiplicity & fan-out (why totals get duplicated)
- NULL behavior
- Non-equi / range joins
- Join with aggregation
- “Top-N per group” join cases
- Performance + indexing rules for joins
- Practice queries with expected outcomes (based on your data)

0) Your schema relationships (think in graphs)

HR side

- `departments (1) -- (many) employees`
- `employees (1 manager) -- (many reports)` via `employees.manager_id -> employees.emp_id` (self-reference)

Sales side

- `customers (1) -- (many) orders`
- `orders (1) -- (many) order_items`
- `products (1) -- (many) order_items`

So the canonical path for invoice-like outputs is:

`customers → orders → order_items → products`

---

1) Join basics: what JOIN actually does

Mental model (important)

A JOIN combines rows from two tables based on a **match condition**.

- Think of it as generating **pairs of rows**.
- For every row in left table, find matching rows in right table.
- Output row count depends on relationship:
    - 1-to-1 → same count
    - 1-to-many → **expands** rows (fan-out)
    - many-to-many → **explodes** rows

---

2) INNER JOIN (most common)

Definition

Returns only rows where the join condition matches on both sides.

Example 1: Employees with department name

```
SELECT e.emp_id, e.first_name, d.dept_name
FROM employees e
INNER JOIN departments d
  ON e.dept_id = d.dept_id;
```

Result (based on your data): all employees appear, because every employee has valid `dept_id` (1/2/3 exist).

Example 2: Orders with customer name

```
SELECT o.order_id, c.customer_name, o.total_amount, o.status
FROM orders o
JOIN customers c
  ON o.customer_id = c.customer_id;
```

Result: 4 orders → 4 rows.

---

3) LEFT JOIN (most important after INNER)

Definition

Returns:

- all rows from the **left table**
- matching rows from right table
- if no match: right-side columns become **NULL**

Example 1: List all customers and their orders (including customers with no orders)

```
SELECT c.customer_id, c.customer_name, o.order_id, o.status
FROM customers c
LEFT JOIN orders o
  ON c.customer_id = o.customer_id
ORDER BY c.customer_id, o.order_id;
```

Expected using your data

- Rahul (customer_id=1) → two orders (1 and 3)
- Sneha (2) → one order (2)
- Arjun (3) → one order (4)
- Kiran (4) → **no order** → **order_id NULL**

This query is the standard pattern for “show all X even if no Y exists”.

Example 2: Departments and employees (including empty departments)

Right now all departments have employees? Actually:

- Engineering (1): Aarav, Isha
- Sales (2): Rohan, Meera
- HR (3): Anjali
- Finance (4): **no employees**

So:

```
SELECT d.dept_name, e.emp_id, e.first_name
FROM departments d
LEFT JOIN employees e
  ON d.dept_id = e.dept_id
ORDER BY d.dept_id, e.emp_id;
```

Finance will appear with `NULL` employee columns.

---

4) RIGHT JOIN

Definition

Same as LEFT JOIN but reversed: keeps all rows from the **right table**.

In practice: avoid RIGHT JOIN for readability; just swap table order and use LEFT JOIN.

Example:

```
SELECT d.dept_name, e.first_name
FROM employees e
RIGHT JOIN departments d
  ON e.dept_id = d.dept_id;
```

Equivalent (preferred):

```
SELECT d.dept_name, e.first_name
FROM departments d
LEFT JOIN employees e
  ON e.dept_id = d.dept_id;
```

---

5) FULL OUTER JOIN (not supported directly in MySQL)

Full outer join returns:

- all matching rows (like inner join)
- plus left-only rows
- plus right-only rows

In MySQL, simulate using `UNION` of LEFT and RIGHT joins:

```
SELECT d.dept_id, d.dept_name, e.emp_id, e.first_name
FROM departments d
LEFT JOIN employees e
  ON d.dept_id = e.dept_id

UNION

SELECT d.dept_id, d.dept_name, e.emp_id, e.first_name
FROM departments d
RIGHT JOIN employees e
  ON d.dept_id = e.dept_id;
```

(Use `UNION ALL` carefully; it may duplicate matched rows.)

---

6) CROSS JOIN (Cartesian product)

Definition

Every row from left pairs with every row from right.

```
SELECT c.customer_name, p.product_name
FROM customers c
CROSS JOIN products p;
```

With your data: 4 customers × 4 products = **16 rows**.

Real-life use

- generating combinations
- calendars (dates × entities)
- pricing matrices

Danger
Accidentally missing ON condition in joins can create a cross join implicitly and explode results.

---

7) SELF JOIN (employees → managers)

Because `employees.manager_id` references `employees.emp_id`.

Example: show each employee with their manager

```
SELECT
  e.emp_id,
  CONCAT(e.first_name, ' ', e.last_name) AS employee,
  CONCAT(m.first_name, ' ', m.last_name) AS manager
FROM employees e
LEFT JOIN employees m
  ON e.manager_id = m.emp_id
ORDER BY e.emp_id;
```

Expected

- Aarav has NULL manager
- Isha manager = Aarav

- Rohan NULL
- Meera manager = Rohan
- Anjali NULL

---

8) Multi-table joins (invoice style)

Example: order details (customer + items + product)

```
SELECT
  o.order_id,
  c.customer_name,
  o.status,
  oi.quantity,
  oi.price AS item_price,
  p.product_name
FROM orders o
JOIN customers c   ON c.customer_id = o.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
JOIN products p    ON p.product_id = oi.product_id
ORDER BY o.order_id;
```

Fan-out warning
Orders can have multiple items → this query returns **one row per order item** (not one per order).

Order 1 has 2 items → you'll see order 1 twice.

---

9) Join filtering pitfall: WHERE vs ON (critical)

This causes real bugs.

Problem: “show all customers, but only completed orders”

Many people write:

```
SELECT c.customer_name, o.order_id, o.status
FROM customers c
LEFT JOIN orders o
  ON c.customer_id = o.customer_id
WHERE o.status = 'Completed';
```

This is WRONG if you want customers with no orders.

Why wrong?

`WHERE o.status='Completed'` filters after the join.
For customers with no order, `o.status` is NULL, so they get removed.
Your LEFT JOIN effectively becomes an INNER JOIN.

✅ Correct version (put filter in ON):

```
SELECT c.customer_name, o.order_id, o.status
FROM customers c
LEFT JOIN orders o
  ON c.customer_id = o.customer_id
 AND o.status = 'Completed'
ORDER BY c.customer_id;
```

Now Kiran still appears with NULL order columns.

---

10) Semi-joins and Anti-joins (advanced patterns)

A) Customers who have at least one order (semi-join)

Use `EXISTS` (often best):

```
SELECT c.*
```

```
FROM customers c
WHERE EXISTS (
  SELECT 1
  FROM orders o
  WHERE o.customer_id = c.customer_id
);
```

This returns Rahul, Sneha, Arjun (not Kiran).

B) Customers who have no orders (anti-join)

```
SELECT c.*
FROM customers c
WHERE NOT EXISTS (
  SELECT 1
  FROM orders o
  WHERE o.customer_id = c.customer_id
);
```

Returns Kiran.

Alternative using LEFT JOIN:

```
SELECT c.*
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.customer_id
WHERE o.order_id IS NULL;
```

---

11) Non-equi joins (theta joins) / range joins

Not all joins are equality-based.

Example concept: suppose you want to bucket employees into salary bands (you'd need a salary_band table). In real systems, "range joins" exist, but they can be expensive because indexes are harder to use.

A simple example without extra tables:

- Find pairs of employees within same department where salary difference <= 5000:

```
SELECT
  e1.emp_id AS emp1, e1.salary AS sal1,
  e2.emp_id AS emp2, e2.salary AS sal2,
  e1.dept_id
FROM employees e1
JOIN employees e2
  ON e1.dept_id = e2.dept_id
 AND e1.emp_id < e2.emp_id
 AND ABS(e1.salary - e2.salary) <= 5000
ORDER BY e1.dept_id;
```

This is a valid join condition (not just equality).

---

12) Aggregation + joins (avoid double counting)

Example: total spent per customer (correct)

```
SELECT c.customer_name, SUM(o.total_amount) AS total_spent
FROM customers c
JOIN orders o ON o.customer_id = c.customer_id
GROUP BY c.customer_id, c.customer_name
ORDER BY total_spent DESC;
```

Common mistake: joining order_items and then summing order total

If you do this:

```
SELECT c.customer_name, SUM(o.total_amount)
FROM customers c
JOIN orders o ON o.customer_id = c.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
GROUP BY c.customer_id;
```

You will multiply `o.total_amount` by number of items in each order (Order 1 has 2 items → it doubles).

✅ Fix options:

1. Sum order totals without joining items
2. Or compute total from items (`SUM(oi.quantity * oi.price)`), not from `orders.total_amount`

Correct "from items":

```
SELECT c.customer_name,
       SUM(oi.quantity * oi.price) AS total_from_items
FROM customers c
JOIN orders o ON o.customer_id = c.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
GROUP BY c.customer_id, c.customer_name;
```

---

13) "Top N per group" join cases (advanced)

Example: highest paid employee per department.

Approach 1: correlated subquery

```
SELECT e.*
FROM employees e
WHERE e.salary = (
  SELECT MAX(e2.salary)
  FROM employees e2
  WHERE e2.dept_id = e.dept_id
);
```

Approach 2: window function (MySQL 8+ best)

```
SELECT *
FROM (
  SELECT e.*,
```

```
        ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary
DESC) AS rn
  FROM employees e
) x
WHERE rn = 1;
```

---

14) Join performance rules (practical)

Indexes that matter for joins

1. The join column(s) should be indexed on at least one side:

- `employees.dept_id`
- `orders.customer_id`
- `order_items.order_id`
- `order_items.product_id`

Foreign keys *often* create indexes in MySQL automatically depending on version/config, but do not assume—verify.

Check indexes:

```
SHOW INDEX FROM orders;
SHOW INDEX FROM order_items;
```

Cardinality & join order

- Optimizer usually starts with the table that reduces rows earliest (high selectivity).
- If you filter by `dept_name='Engineering'`, index `departments.dept_name` helps start from departments then join employees.

Add index:

```
CREATE INDEX idx_dept_name ON departments(dept_name);
```

15) Practice set: Join exercises (increasing difficulty)

1) Employees with department name

```
SELECT e.emp_id, e.first_name, d.dept_name
FROM employees e
JOIN departments d ON d.dept_id = e.dept_id;
```

2) Departments with employees (include empty departments)

```
SELECT d.dept_name, e.first_name
FROM departments d
LEFT JOIN employees e ON e.dept_id = d.dept_id;
```

3) Customers with orders (include customers with no orders)

```
SELECT c.customer_name, o.order_id, o.status
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.customer_id;
```

4) Customers with completed orders only BUT still include customers with no orders

```
SELECT c.customer_name, o.order_id, o.status
FROM customers c
LEFT JOIN orders o
  ON o.customer_id = c.customer_id
 AND o.status = 'Completed';
```

5) Full invoice view (order + customer + items + product)

```
SELECT o.order_id, c.customer_name, p.product_name, oi.quantity,
oi.price
FROM orders o
JOIN customers c ON c.customer_id = o.customer_id
JOIN order_items oi ON oi.order_id = o.order_id
JOIN products p ON p.product_id = oi.product_id
```

```
ORDER BY o.order_id;
```

6) Employee → manager mapping (self join)

```
SELECT CONCAT(e.first_name,' ',e.last_name) AS employee,
       CONCAT(m.first_name,' ',m.last_name) AS manager
FROM employees e
LEFT JOIN employees m ON m.emp_id = e.manager_id;
```

7) Customers who have no orders (anti-join)

```
SELECT c.customer_name
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.customer_id
WHERE o.order_id IS NULL;
```

8) Product revenue (join + aggregation)

```
SELECT p.product_name,
       SUM(oi.quantity * oi.price) AS revenue
FROM products p
JOIN order_items oi ON oi.product_id = p.product_id
GROUP BY p.product_id, p.product_name
ORDER BY revenue DESC;
```

---

16) Common join interview traps (you should master)

1. **LEFT JOIN + WHERE on right table** turns into INNER JOIN (unless you handle NULLs)
2. Fan-out causes duplicated totals
3. Tree of joins: `A → B → C` can multiply rows unexpectedly
4. TreeMap-like behavior doesn't exist in joins; joins don't "deduplicate" automatically
5. `DISTINCT` is not a real fix for wrong joins; it hides bugs and can be expensive

17) Your “mastery checklist” for joins

You’re strong when you can answer/do all of these:

- Pick correct join type for “include missing” scenarios
- Predict row count growth (fan-out) before running query
- Place filters correctly (ON vs WHERE)
- Build invoice-like multi-joins without double-counting
- Use EXISTS / NOT EXISTS for semi/anti joins
- Use self join for hierarchies
- Understand FULL join simulation in MySQL
- Know which indexes matter for join performance