1️⃣ How do you find the third highest salary without using TOP or LIMIT?

SELECT DISTINCT salary

FROM employees e1

WHERE 2 = (

SELECT COUNT(DISTINCT salary)

FROM employees e2

WHERE e2.salary > e1.salary

);

| id | name | salary |

| -- | ----- | ------ |

| 1 | John | 5000 |

| 2 | Alice | 7000 |

| 3 | Bob | 6000 |

| 4 | Mike | 8000 |

| 5 | Lisa | 7000 |

 Distinct salaries: 8000, 7000, 6000, 5000

 Third highest = 6000 → Two salaries are greater (8000, 7000)

SELECT salary

FROM (

SELECT salary, DENSE\_RANK() OVER (ORDER BY salary DESC) AS rnk

FROM employees

) AS ranked

WHERE rnk = 3;

2️⃣ How do you remove duplicate rows based on multiple columns?

DELETE FROM employees

WHERE id NOT IN (

SELECT id FROM (

SELECT MIN(id) AS id

FROM employees

GROUP BY first\_name, last\_name, department

) AS keepers

);

🔹 **Explanation**:

* Groups by multiple columns (first\_name, last\_name, department)
* Keeps the row with the **minimum id** per group
* Deletes all others (i.e., duplicates)

✅ Option 2: Use ROW\_NUMBER() with CTE (MySQL 8.0+)

WITH ranked AS (

SELECT \*,

ROW\_NUMBER() OVER (PARTITION BY first\_name, last\_name, department ORDER BY id) AS rn

FROM employees

)

DELETE FROM employees

WHERE id IN (

SELECT id FROM ranked WHERE rn > 1

);

🔹 Keeps the **first row** in each duplicate group (based on id) and deletes the rest.

| id | first\\_name | last\\_name | department |

| -- | ----------- | ---------- | ---------- |

| 1 | John | Smith | HR |

| 2 | John | Smith | HR |

| 3 | Alice | Jones | IT |

| 4 | John | Smith | HR |

3️⃣ What’s the use of the PARTITION BY clause in SQL?

The PARTITION BY clause is used **within window functions** to **divide the result set into groups (partitions)** before applying a function like ROW\_NUMBER(), RANK(), SUM(), AVG(), etc.

4️⃣ How do you calculate percentage contribution of each row to a total?

✅ Using SUM() OVER()

SELECT

employee\_id,

department\_id,

salary,

ROUND(100.0 \* salary / SUM(salary) OVER (), 2) AS pct\_of\_total\_salary

FROM employees;

**🔍 What it does:**

* SUM(salary) OVER (): computes the total salary **once for all rows**
* Each row gets a salary / total\_salary \* 100
* ROUND(..., 2): rounds to 2 decimal places

5️⃣ How do you join three or more tables efficiently?

**✅ How to Join Three or More Tables Efficiently in SQL**

Joining multiple tables efficiently means:

1. **Using correct join types** (INNER, LEFT, etc.)
2. **Joining on indexed keys** (like foreign keys)
3. **Filtering early** (use WHERE or ON filters as soon as possible)
4. **Joining logically** (based on data relationships, not arbitrarily)

🔧 Syntax: Joining 3 Tables

SELECT e.employee\_id, e.name, d.department\_name, l.location\_name

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id

JOIN locations l ON d.location\_id = l.location\_id;

**🧠 Example Schema:**

* employees(employee\_id, name, department\_id)
* departments(department\_id, department\_name, location\_id)
* locations(location\_id, location\_name)

**🔍 Tips for Efficiency:**

1. **Use INNER JOIN** if all joined data must match (faster).
2. **Use LEFT JOIN** only when you need unmatched rows.
3. **Filter early** to reduce the number of rows before joins:

SELECT ...

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id

WHERE e.status = 'active'

🛠 Alternative: Use CTEs for Readability

WITH emp\_dept AS (

SELECT e.employee\_id, e.name, d.department\_name, d.location\_id

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id

)

SELECT ed.\*, l.location\_name

FROM emp\_dept ed

JOIN locations l ON ed.location\_id = l.location\_id;

**⚡ Pro Tip: EXPLAIN Your Query**

Use EXPLAIN (in MySQL) to see how the query is executed and optimize if needed:

EXPLAIN SELECT ...

6️⃣ How can you retrieve only the latest record per user based on a timestamp?

✅ Option 1: Using ROW\_NUMBER()

SELECT user\_id, action, action\_time

FROM (

SELECT \*,

ROW\_NUMBER() OVER (PARTITION BY user\_id ORDER BY action\_time DESC) AS rn

FROM user\_actions

) AS ranked

WHERE rn = 1;

**🔍 What it does:**

* PARTITION BY user\_id: groups rows by user
* ORDER BY action\_time DESC: gives the latest record first
* ROW\_NUMBER() assigns rank starting from 1
* Outer query filters to only the **latest record per user**

✅ Option 2: Using a Subquery

SELECT ua.\*

FROM user\_actions ua

JOIN (

SELECT user\_id, MAX(action\_time) AS latest\_time

FROM user\_actions

GROUP BY user\_id

) AS latest

ON ua.user\_id = latest.user\_id AND ua.action\_time = latest.latest\_time;

🧠 Example Table:

| user\\_id | action | action\\_time |

| -------- | ------ | ------------------- |

| 1 | login | 2024-01-01 08:00:00 |

| 1 | logout | 2024-01-01 10:00:00 |

| 2 | login | 2024-01-01 09:00:00 |

➡️ Result:

| user\\_id | action | action\\_time |

| -------- | ------ | ------------------- |

| 1 | logout | 2024-01-01 10:00:00 |

| 2 | login | 2024-01-01 09:00:00 |

7️⃣ What’s the difference between CHAR, VARCHAR, and TEXT?

**✅ 1. CHAR(n) — Fixed-length string**

* **Allocates exactly n characters**, padding with spaces if needed.
* Best for **uniform-length** data like country codes (CHAR(2) for 'US') or status flags (CHAR(1) for 'Y'/'N').

Example:

name CHAR(10) -- always takes 10 characters, even if the name is 'Amy'

**| Input | Storage |**

**| ----- | ------------------------- |**

**| 'Amy' | `'Amy       '` (7 spaces) |**

**✅ 2. VARCHAR(n) — Variable-length string**

* Stores only the actual content + 1 or 2 bytes for length.
* Best for most string fields like names, emails, etc.
* More **space-efficient** than CHAR for variable-length data.

**Example:**

**email VARCHAR(50)**

**| Input | Storage |**

**| ----------------------------------------------------------------- | -------- |**

**| 'a@b.com ' | 7 bytes |**

**| 'averylongemail@example.com' | 27 bytes |**

8️⃣ How do you handle NULLs in aggregations like AVG, COUNT, or SUM?

✅ Here's how SQL handles NULL values in common aggregations:

| Function | Ignores NULLs? | Behavior |

| --------------- | -------------- | --------------------------------------------- |

| `COUNT(\*)` | ❌ No | Counts all rows, including `NULL`s |

| `COUNT(column)` | ✅ Yes | Counts only \*\*non-NULL\*\* values in the column |

| `SUM(column)` | ✅ Yes | Adds only \*\*non-NULL\*\* values |

| `AVG(column)` | ✅ Yes | Averages only \*\*non-NULL\*\* values |

| `MAX(column)` | ✅ Yes | Ignores `NULL`s, returns max of non-NULLs |

| `MIN(column)` | ✅ Yes | Ignores `NULL`s, returns min of non-NULLs |

🧠 Example Table:

| id | salary |

| -- | ------ |

| 1 | 1000 |

| 2 | NULL |

| 3 | 2000 |

**Results:**

* COUNT(\*) → 3 (counts all rows)
* COUNT(salary) → 2 (ignores NULL)
* SUM(salary) → 3000
* AVG(salary) → 1500 (3000 / 2)

**🛠️ To handle NULLs explicitly:**

Use COALESCE() to **replace NULLs with 0** or another default value.

SELECT SUM(COALESCE(salary, 0)) AS total\_salary

FROM employees;

✅ Pro Tip: Prevent divide-by-zero with NULLs

SELECT

SUM(salary) / NULLIF(COUNT(salary), 0) AS safe\_avg

FROM employees;

NULLIF(count, 0) returns NULL if count = 0, preventing a divide-by-zero error.

9️⃣ How do you generate a calendar or date dimension using SQL?

Generating a **calendar or date dimension** is a common task in data warehousing and analytics — it provides a ready-to-use table with dates and their attributes (day, month, year, weekday, etc.).

WITH RECURSIVE calendar AS (

SELECT DATE('2024-01-01') AS dt

UNION ALL

SELECT dt + INTERVAL 1 DAY

FROM calendar

WHERE dt < '2024-12-31'

)

SELECT

dt,

DAY(dt) AS day,

MONTH(dt) AS month,

YEAR(dt) AS year,

DAYNAME(dt) AS weekday\_name,

WEEKDAY(dt) AS weekday\_num,

WEEK(dt) AS week\_num,

QUARTER(dt) AS quarter,

CASE WHEN WEEKDAY(dt) < 5 THEN 'Weekday' ELSE 'Weekend' END AS day\_type

FROM calendar;

**📅 What This Does:**

* Builds a list of dates from **2024-01-01 to 2024-12-31**
* Adds useful columns:
  + day, month, year
  + weekday\_name, week\_num, quarter
  + day\_type (Weekday vs Weekend)

**⚠️ Notes:**

* RECURSIVE CTE works in MySQL 8.0+ only.
* MySQL allows recursion up to 1000 rows by default — you can extend using:

SET max\_execution\_time = 0;

🔟 What’s the impact of indexing on SELECT vs INSERT performance?  
1️⃣1️⃣ How do you retrieve every alternate row from a table?  
1️⃣2️⃣ How can you implement IF-ELSE logic inside a SQL query?  
1️⃣3️⃣ What is normalization and when should you denormalize data instead?  
1️⃣4️⃣ How do you create a running average in SQL?  
1️⃣5️⃣ How do you pivot a table dynamically based on values in a column?  
1️⃣6️⃣ What are the pros and cons of using stored procedures?  
1️⃣7️⃣ How do you filter rows using a value from the previous row in SQL?  
1️⃣8️⃣ What is the difference between a primary key and a unique constraint?  
1️⃣9️⃣ How do you perform a case-insensitive search in SQL?  
2️⃣0️⃣ What techniques do you use to troubleshoot slow-running queries?

## Q - **Get IDs That Only Taught Mathematics**

| id | subject |

| -- | ------- |

| 1 | M |

| 2 | M |

| 3 | C |

| 2 | C |

| 4 | M |

| 5 | P |

| 5 | M |

SELECT id

FROM your\_table

GROUP BY id

HAVING COUNT(DISTINCT subject) = 1

AND MAX(subject) = 'M';

✅ **Expected Output:**

| id |

| -- |

| 1 |

| 4 |

✅ Result of COUNT(DISTINCT subject):

| id | unique\\_subjects | |

| -- | ---------------- | --------- |

| 1 | 1 | -- Only M |

| 2 | 2 | -- M, C |

| 3 | 1 | -- Only C |

| 4 | 1 | -- Only M |

| 5 | 2 | -- M, P |