
🧠 10 Hard Theoretical SQL Questions with Answers

1. What is the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN?

Answer:

- **INNER JOIN** → Returns rows that have matching values in both tables.
- **LEFT JOIN** → Returns all rows from the left table and matching rows from the right table; unmatched right rows → NULL.
- **RIGHT JOIN** → Opposite of LEFT JOIN.
- **FULL OUTER JOIN** → Returns all rows when there's a match in either table; unmatched rows get NULL.

---

2. What are window functions in SQL, and how are they different from aggregate functions?

Answer:

- **Window functions** perform calculations across a set of table rows that are somehow related to the current row.
- Unlike aggregate functions, **they don't collapse rows**.
  Example:

```
SELECT name, salary, RANK() OVER (ORDER BY salary DESC) AS salary_rank

FROM employees;
```

---

3. Explain the difference between WHERE and HAVING clause.

Answer:

- WHERE → filters rows **before aggregation**.
- HAVING → filters **after aggregation** (used with GROUP BY).
  Example:

```
SELECT dept, COUNT(*)

FROM employees

WHERE salary > 30000

GROUP BY dept

HAVING COUNT(*) > 5;
```

---

4. What is normalization? Explain up to 3NF.

Answer:

- **Normalization** = process of organizing data to minimize redundancy.
  - **1NF:** Atomic values (no repeating groups).
  - **2NF:** 1NF + no partial dependency.

    - **3NF:** 2NF + no transitive dependency.

---

5. What is the difference between UNION and UNION ALL?

Answer:

- UNION → combines results and removes duplicates.
- UNION ALL → includes duplicates.
  UNION ALL is faster because it skips duplicate checks.

---

6. What are ACID properties in SQL transactions?

Answer:

- **A**tomicity → all or none
- **C**onsistency → maintains database rules
- **I**solation → concurrent transactions don't affect each other
- **D**urability → once committed, data persists

---

7. Explain the concept of indexing. What are clustered and non-clustered indexes?

Answer:

- **Index** speeds up retrieval but slows down insertion/updating.
- **Clustered Index:** Alters the physical order of data (one per table).
- **Non-Clustered Index:** Logical order; stores pointers to data.

  n Perfect, Gaurav 👌 — let's **see it in action step-by-step** with a **real example** to visualize exactly how **clustered and non-clustered indexes** behave inside a database.

  I'll show you how to:
  1️⃣ Create a table
  2️⃣ Insert data
  3️⃣ Create a **clustered index**
  4️⃣ Create a **non-clustered index**
  5️⃣ Understand **how the data looks** internally (conceptually)

  🧩 **Step 1: Create a Sample Table**

```
CREATE TABLE Employees (

   EmpID INT PRIMARY KEY,

   EmpName VARCHAR(50),

   Department VARCHAR(30),

   Salary INT
```

);

➡️ Here, EmpID is the **PRIMARY KEY**, which **automatically creates a clustered index**.

🧩 Step 2: Insert Sample Data

INSERT INTO Employees (EmpID, EmpName, Department, Salary) VALUES

(103, 'Ravi Kumar', 'HR', 45000),

(101, 'Gaurav Sahu', 'IT', 60000),

(105, 'Neha Singh', 'Finance', 70000),

(102, 'Aman Verma', 'IT', 55000),

(104, 'Priya Das', 'HR', 50000);

🧩 Step 3: Clustered Index in Action

Since the **primary key (EmpID)** already has a **clustered index**, the data will be **physically sorted** by EmpID.

So even though you inserted rows in random order,
the **table data is stored like this internally** 👇

| 📦 Physical Storage Order (Clustered Index on EmpID) |

EmpID	EmpName	Department	Salary
101	Gaurav Sahu	IT	60000
102	Aman Verma	IT	55000
103	Ravi Kumar	HR	45000
104	Priya Das	HR	50000
105	Neha Singh	Finance	70000

✅ **Notice:** The data is now **physically sorted** by EmpID.

📚 This means if you query:

SELECT * FROM Employees WHERE EmpID = 104;

SQL directly navigates the **clustered index tree** → finds 104 → retrieves the record immediately.
No full table scan needed.

🧩 Step 4: Create a Non-Clustered Index

Now suppose we often search by employee name.

CREATE NONCLUSTERED INDEX idx_emp_name

ON Employees(EmpName);

This creates a **separate index structure** that looks like this internally 👇

🔍 Non-Clustered Index (idx_emp_name)	➡️ Points To (Clustered Key)
Aman Verma	102
Gaurav Sahu	101
Neha Singh	105
Priya Das	104
Ravi Kumar	103

✅ This index is **stored separately** from the main table.
It **does not reorder the table** — it only keeps **sorted EmpName + pointer (EmpID)**.

🧩 Step 5: Non-Clustered Index in Action

When you query:

SELECT * FROM Employees WHERE EmpName = 'Neha Singh';

The database engine performs:

1. Looks into the **non-clustered index (idx_emp_name)**
   → finds 'Neha Singh'
   → gets pointer EmpID = 105
2. Goes to the **clustered index (on EmpID)**
   → fetches full record of EmpID 105.

So, 2 lookups:
Index → Clustered Table → Data

Still **much faster** than scanning all rows!

⚙️ Step 6: Confirm the Indexes (Optional)

In SQL Server or MySQL, you can view indexes using:

```
-- SQL Server

EXEC sp_helpindex 'Employees';



-- MySQL

SHOW INDEX FROM Employees;
```

You'll see something like:

Table	Non_unique	Key_name	Column_name	Index_type
Employees	0	PRIMARY	EmpID	CLUSTERED
Employees	1	idx_emp_name	EmpName	NONCLUSTERED

📘 Summary

Feature	Clustered Index	Non-Clustered Index
Storage	Actual table data sorted	Separate structure
Sorting	Physically reorders rows	Logically orders key values
Count	Only 1 per table	Multiple allowed
Lookup Speed	1 step (direct access)	2 steps (index → data)
Example	PRIMARY KEY (EmpID)	INDEX (EmpName)

🧠 Visual Representation

📦 Table Data (Clustered by EmpID)

```
---------------------------------
EmpID | EmpName     | Dept      | Salary
101   | Gaurav Sahu | IT        | 60000
102   | Aman Verma  | IT        | 55000
103   | Ravi Kumar  | HR        | 45000
104   | Priya Das   | HR        | 50000
105   | Neha Singh  | Finance   | 70000
```

🔍 Non-Clustered Index (on EmpName)

```
----------------------------------
EmpName     → Points To EmpID
Aman Verma  → 102
Gaurav Sahu → 101
Neha Singh  → 105
Priya Das   → 104
Ravi Kumar  → 103
```

8. What is a CTE (Common Table Expression)?

Answer:

- A **temporary result set** defined within a query for readability and reuse.
  Example:

```
WITH SalesCTE AS (
  SELECT emp_id, SUM(amount) AS total_sales
  FROM sales
  GROUP BY emp_id
)
SELECT * FROM SalesCTE WHERE total_sales > 10000;
```

---

9. What is a subquery and how does it differ from a JOIN?

Answer:

- **Subquery:** A query inside another query.
- **JOIN:** Combines data horizontally from multiple tables.
  Subqueries can be used in WHERE, SELECT, or FROM clauses.

  Excellent, Gaurav 👏 — here's a **concise, professional, and interview-ready explanation** of
  "What is a Subquery and how it differs from a JOIN", formatted like a crisp **interview answer + comparison table**.

  You can literally **speak or write this directly** in interviews 💼 👇

  💬 **Interview Answer:**

  A **subquery** is a query **nested inside another query** to perform an operation where the result of the inner query is used by the outer query.
  In contrast, a **JOIN** is used to **combine data from two or more tables** based on a related column.

  The main difference is that a **subquery executes first** and passes its result to the outer query, while a **JOIN processes multiple tables together** to produce a single result set.

  🧠 **Subquery vs JOIN — Interview Comparison Table**

Feature / Point	Subquery	JOIN
Definition	A query inside another query.	Combines rows from two or more tables using related columns.
Execution	Inner query executes first, then outer query uses its result.	All tables are processed together by the database engine.
Purpose	To filter, compare, or compute values based on another query's result.	To retrieve related data from multiple tables.
Data Involvement	Usually involves **one main table**, but can refer to others inside.	Always involves **two or more tables** directly.

Feature / Point	**Subquery**	**JOIN**
Performance	Can be **slower** (especially correlated subqueries).	Generally **faster** due to optimized join algorithms.
Readability	Easier for **simple filtering logic**.	Easier for **complex data relationships**.
Reusability	Result of subquery is **temporary** (not reusable).	Result can be used or joined with other queries easily.
Dependency	May depend on the outer query (correlated subquery).	Works **independently** with clear join conditions.
Common Use Case	Filtering: find data **based on a calculated result** (e.g., above average salary).	Combining: show **related columns** from multiple tables.
Output	Returns values used by the outer query (not necessarily all columns).	Returns **combined dataset** with columns from all joined tables.

⚙️ Example 1 – Subquery

```
SELECT emp_name, salary
FROM employees
WHERE salary > (
  SELECT AVG(salary)
  FROM employees
);
```

👉 Finds employees earning **more than the average salary**.
The inner query computes the average; the outer query uses that result.

⚙️ Example 2 – JOIN

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

👉 Combines employee and department data
to show **each employee's department name**.

✅ Interview Summary Points

- Subquery = **Query inside another query**.
- JOIN = **Combines multiple tables**.

- Subquery = **Sequential** execution.
- JOIN = **Parallel** table processing.
- Subquery = Better for **filtering or comparison**.
- JOIN = Better for **combining related data** efficiently.

---

10. Explain referential integrity and how it’s enforced.

Answer:

- **Referential Integrity** ensures that foreign key values in one table match primary key values in another.
- Enforced using **FOREIGN KEY constraints**.

💻 10 Hard Practical SQL Questions with Answers

---

1. Get the 2nd highest salary from the employees table.

```
SELECT MAX(salary)
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

---

2. Find employees who earn more than the average salary of their department.

```
SELECT e.emp_name, e.salary, e.dept_id
FROM employees e
WHERE e.salary > (
  SELECT AVG(salary)
  FROM employees
  WHERE dept_id = e.dept_id
);
```

---

3. Get departments that have more than 5 employees.

```
SELECT dept_id, COUNT(*) AS total_employees
FROM employees
GROUP BY dept_id
```

```
HAVING COUNT(*) > 5;
```

---

4. Write a query to display duplicate records from a table.

```
SELECT name, COUNT(*)
FROM employees
GROUP BY name
HAVING COUNT(*) > 1;
```

---

5. Delete duplicate rows but keep one copy.

```
DELETE FROM employees
WHERE id NOT IN (
  SELECT MIN(id)
  FROM employees
  GROUP BY name, dept_id, salary
);
```

```
DELETE s1
FROM students s1
JOIN students s2
ON s1.name = s2.name
AND s1.email = s2.email
AND s1.id > s2.id;
```

---

6. Get top 3 highest-paid employees from each department.

```
SELECT *
FROM (
  SELECT e.*,
         ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS rnk
  FROM employees e
) ranked
WHERE rnk <= 3;
```

---

7. Find departments where all employees earn more than ₹30,000.

```
SELECT dept_id
FROM employees
GROUP BY dept_id
HAVING MIN(salary) > 30000;
```

---

8. Find employees who don’t belong to any department.

```
SELECT e.emp_name
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.dept_id
WHERE d.dept_id IS NULL;
```

---

9. Display employee name and their manager’s name (self join).

```
SELECT e.emp_name AS Employee, m.emp_name AS Manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.emp_id;
```

---

10. Retrieve employees hired in the last 90 days.

```
SELECT emp_name, hire_date
FROM employees
WHERE hire_date >= CURRENT_DATE - INTERVAL '90' DAY;
```
