

SQL interview questions

29 questions · 11 easy · 13 medium · 5 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

In what order does SQL logically execute a **SELECT** query?

Easy



#basics #select

You write **SELECT ... FROM ... WHERE ...**, but the database logically evaluates it in this order:

- **FROM** and **JOIN**: build the working set of rows
- **WHERE**: filter individual rows
- **GROUP BY**: collapse rows into groups
- **HAVING**: filter groups
- **SELECT**: compute columns and aliases
- **DISTINCT**, then **ORDER BY**, then **LIMIT** / **OFFSET**

This explains common errors: a **SELECT** alias does not exist yet in **WHERE**, so you cannot filter on it there, but you can use it in **ORDER BY**. The optimiser may physically run things differently, but the result must match this logical order.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

-- the alias works in ORDER BY (runs after SELECT) but not in WHERE
SELECT name, salary * 12 AS yearly
FROM emp
WHERE salary * 12 > 1000000
ORDER BY yearly DESC;
```

🔗 **Likely follow-up:** Why can MySQL let you use an alias in **HAVING** when the standard says no?

2

What is the difference between **WHERE** and **HAVING** ?

Easy



#aggregation #basics

WHERE filters **rows** before grouping. **HAVING** filters **groups** after **GROUP BY** has run.

So aggregates like **COUNT(*)** or **AVG(salary)** can only appear in **HAVING**, because they do not exist until the groups are built.

- Put every condition that does not need an aggregate in **WHERE**: it **shrinks the data before grouping**, which is faster.
- **HAVING** without **GROUP BY** treats the whole table as one group.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

-- departments with 2+ people earning over 60k
SELECT dept_id, COUNT(*) AS people, AVG(salary) AS avg_salary
FROM emp
WHERE salary > 60000          -- row filter
GROUP BY dept_id
HAVING COUNT(*) >= 2;        -- group filter
```

3

Explain the different types of joins.

Easy 🔥🔥 #joins

- **INNER JOIN**: only rows that match in both tables.
- **LEFT JOIN**: every row from the left table, with `NULL`s where the right has no match.
- **RIGHT JOIN**: the mirror image; usually rewritten as a LEFT JOIN.
- **FULL OUTER JOIN**: all rows from both sides, matched where possible (not in MySQL; emulate with `LEFT JOIN UNION RIGHT JOIN`).
- **CROSS JOIN**: every combination (a Cartesian product).
- **SELF JOIN**: a table joined to itself with aliases.

A classic trick is the **anti-join**: `LEFT JOIN` and then `WHERE right.id IS NULL` finds rows with no match. Also remember that a filter on the right table placed in `WHERE` turns a LEFT JOIN back into an inner join; put it in the `ON` clause instead.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT e.name, d.name AS dept FROM emp e
INNER JOIN dept d ON d.id = e.dept_id;      -- Farah missing

SELECT e.name, d.name AS dept FROM emp e
LEFT JOIN dept d ON d.id = e.dept_id;      -- Farah with NULL

SELECT d.name FROM dept d
LEFT JOIN emp e ON e.dept_id = d.id
WHERE e.id IS NULL;                        -- anti-join: HR has nobody
```

🔗 Likely follow-up: What happens to a LEFT JOIN if you add `WHERE d.name = 'Sales'`?

4

What is a self join? Find each employee with their manager's name.

Easy



#joins #self join

A **self join** joins a table to itself, using two aliases so each copy can play a different role. It is the standard way to query hierarchies stored in one table, like an `emp.manager_id` that points at another `emp.id`.

Use a `LEFT JOIN` if you want to keep the top boss, who has no manager. With an `INNER JOIN` the CEO would disappear.

For more than one level (the whole management chain), you need a **recursive CTE** instead.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT e.name AS employee, m.name AS manager
FROM emp e
LEFT JOIN emp m ON m.id = e.manager_id;
```

5

Find employees who earn more than their manager.

Easy



#self join #classic

Self join the employee table to itself on the manager key, then compare the two salary columns:

`e.manager_id = m.id` links each row to its manager row, and `e.salary > m.salary` keeps only those who out-earn them.

An `INNER JOIN` is correct here, because someone with no manager has nothing to compare against. The same pattern answers related questions: employees in a different department than their manager, or managers with more than N direct reports (group by `m.id`).

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);
UPDATE emp SET salary = 95000 WHERE id = 5;  -- Eva now out-earns Dina

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

6

Explain primary, foreign, unique and composite keys.

Easy



#keys #constraints

- **Primary key:** uniquely identifies each row; **unique and never NULL**; one per table. Usually a surrogate (**id**) column.
- **Foreign key:** a column that must match a key in another table, enforcing **referential integrity**. **ON DELETE CASCADE** / **SET NULL** define what happens when the parent row goes.
- **Unique key:** no duplicate values, but (in most databases) it allows **NULL**s, and a table can have many.
- **Composite key:** a key made of several columns, such as (**student_id, course_id**) in a join table.
- **Candidate key:** any minimal set of columns that could be the primary key; a **natural key** comes from real data (email), a **surrogate key** is generated.

Example

```
DROP TABLE IF EXISTS enrolment;
DROP TABLE IF EXISTS student;
CREATE TABLE student (
  id INTEGER PRIMARY KEY,
  email TEXT UNIQUE NOT NULL
);
CREATE TABLE enrolment (
  student_id INTEGER REFERENCES student(id) ON DELETE CASCADE,
  course TEXT,
  PRIMARY KEY (student_id, course)           -- composite key
);
INSERT INTO student VALUES (1, 'asha@x.io');
INSERT INTO enrolment VALUES (1, 'SQL'), (1, 'Git');
SELECT * FROM enrolment;
```

? Likely follow-up: Should you use a natural key or a surrogate key?

7

Find the second (or Nth) highest salary.

Medium 🔥🔥 #classic #window functions

Ask first: should ties count once? Usually yes, so work on **distinct** salaries.

- **LIMIT/OFFSET:** `SELECT DISTINCT salary ... ORDER BY salary DESC LIMIT 1 OFFSET N-1`. Simple, but returns no row if N is too big.
- **Subquery:** `MAX(salary)` where salary is below the overall `MAX`. Works everywhere, but only nicely for N = 2.
- **DENSE_RANK:** rank salaries and pick rank N. **The most flexible:** handles ties and extends to "Nth highest per department" with `PARTITION BY`.
- **Correlated subquery:** count how many distinct salaries are higher; that count equals N-1.

Use `DENSE_RANK`, not `RANK` or `ROW_NUMBER`, so tied salaries share a position.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT DISTINCT salary FROM emp ORDER BY salary DESC LIMIT 1 OFFSET 1;

SELECT MAX(salary) FROM emp WHERE salary < (SELECT MAX(salary) FROM emp);

SELECT DISTINCT salary FROM (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk FROM emp
) WHERE rnk = 2;

SELECT DISTINCT salary FROM emp e1
WHERE 1 = (SELECT COUNT(DISTINCT salary) FROM emp e2 WHERE e2.salary >
e1.salary);
```

 **Likely follow-up:** Now return the top 2 earners in every department.

8

How do you find and delete duplicate rows?

Medium 🔥🔥 #classic #duplicates

Find them by grouping on the columns that define a duplicate and keeping groups with more than one row: `GROUP BY email HAVING COUNT(*) > 1`.

Delete them while keeping one copy of each:

- keep the lowest `id` per group: delete rows whose `id` is not the `MIN(id)` for their group
- or number each group with `ROW_NUMBER() OVER (PARTITION BY email ORDER BY id)` and delete every row numbered above 1

Afterwards add a `UNIQUE` constraint so duplicates cannot come back. If the rows are identical with no id at all, copy `SELECT DISTINCT` into a new table and swap it in.

Example

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (id INTEGER PRIMARY KEY, email TEXT);
INSERT INTO users (email) VALUES ('a@x.io'), ('b@x.io'), ('a@x.io'),
('a@x.io'), ('b@x.io');

SELECT email, COUNT(*) AS copies FROM users GROUP BY email HAVING
COUNT(*) > 1;

DELETE FROM users WHERE id IN (
  SELECT id FROM (
    SELECT id, ROW_NUMBER() OVER (PARTITION BY email ORDER BY id) AS rn
  FROM users
  ) WHERE rn > 1
);
SELECT * FROM users;
```

9

How does `GROUP BY` work with aggregate functions?

Easy 🔥🔥 #aggregation

`GROUP BY` collapses rows that share the same values into one row per group. Aggregates then summarise each group: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`.

Rule: every column in `SELECT` must either be in the `GROUP BY` or be wrapped in an aggregate. Otherwise the database does not know `which row's value to show` (strict databases throw an error; old MySQL and SQLite pick one silently).

NULL details:

- `COUNT(*)` counts rows; `COUNT(col)` skips `NULL`s
- `AVG` and `SUM` ignore `NULL`s
- all `NULL` keys fall into one group

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT d.name, COUNT(e.id) AS people, SUM(e.salary) AS payroll,
MAX(e.salary) AS top
FROM dept d
LEFT JOIN emp e ON e.dept_id = d.id
GROUP BY d.id, d.name
ORDER BY payroll DESC;
```

10

How does SQL handle **NULL** ?

Medium 🔥🔥 #null #basics

NULL means **unknown**, not zero or empty. Any comparison with it, even **NULL = NULL**, gives **unknown**, which **WHERE** treats as false.

- test with **IS NULL** / **IS NOT NULL**, never **= NULL**
- **COALESCE(a, b)** returns the first non-null value; **NULLIF(a, b)** returns **NULL** when they are equal (handy to avoid divide by zero)
- arithmetic with **NULL** returns **NULL**
- aggregates skip **NULL**s, except **COUNT(*)**

The famous trap: **WHERE id NOT IN (subquery)** returns **no rows** if the subquery contains a **NULL**. Use **NOT EXISTS** instead.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT name, COALESCE(dept_id, 0) AS dept FROM emp;
SELECT COUNT(*) AS all_rows, COUNT(dept_id) AS with_dept FROM emp;

-- NOT IN with a NULL in the list: zero rows
SELECT name FROM dept WHERE id NOT IN (SELECT dept_id FROM emp);
-- NOT EXISTS does the right thing: HR
SELECT name FROM dept d WHERE NOT EXISTS (SELECT 1 FROM emp e WHERE
e.dept_id = d.id);
```

11

Subquery vs join: when do you use which?

Medium



#subqueries #joins

Many queries can be written either way, and modern optimisers often turn `IN` / `EXISTS` subqueries into joins internally.

Join when you need columns from both tables in the result, or you are combining large sets. It is usually the clearest and fastest choice.

Subquery when you only need to test something: `EXISTS`, `IN`, or compare against an aggregate such as "salary above the average". It often reads more naturally.

Watch out:

- a join to a one-to-many table can **multiply rows**, so you then need **DISTINCT**; **EXISTS** avoids that
- a correlated subquery may run once per outer row
- check **EXPLAIN** rather than guessing

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

-- departments that have at least one employee
SELECT d.name FROM dept d WHERE EXISTS (SELECT 1 FROM emp e WHERE
e.dept_id = d.id);
SELECT DISTINCT d.name FROM dept d JOIN emp e ON e.dept_id = d.id;
```

12

What is a correlated subquery?

Medium 🔥🔥 #subqueries

A **correlated subquery** refers to a column of the outer query, so logically it **runs once for every outer row**. A normal (non-correlated) subquery runs once on its own.

Typical example: employees who earn more than the average **of their own department**. The inner **AVG** depends on the outer row's **dept_id**.

They are expressive, but can be slow on big tables. Alternatives are a join to a pre-aggregated derived table or CTE, or a window function like **AVG(salary) OVER (PARTITION BY dept_id)**.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT e.name, e.salary, e.dept_id
FROM emp e
WHERE e.salary > (
  SELECT AVG(i.salary) FROM emp i WHERE i.dept_id = e.dept_id
);
```

? Likely follow-up: Rewrite it with a window function.

13

What are window functions? Compare `ROW_NUMBER`, `RANK` and `DENSE_RANK`.

Medium 🔥🔥 #window functions

A **window function** computes a value over a set of related rows without collapsing them the way `GROUP BY` does. You define the window with `OVER (PARTITION BY ... ORDER BY ...)`.

For salaries 150, 120, 120, 90:

- `ROW_NUMBER`: 1, 2, 3, 4 (always unique, ties broken arbitrarily)
- `RANK`: 1, 2, 2, 4 (ties share, then a gap)
- `DENSE_RANK`: 1, 2, 2, 3 (ties share, no gap)

Use `ROW_NUMBER` for de-duplication and "latest row per group", `DENSE_RANK` for "Nth highest". You cannot filter on a window result in `WHERE`; wrap it in a subquery or CTE.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT name, salary,
  ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,
  RANK() OVER (ORDER BY salary DESC) AS rnk,
  DENSE_RANK() OVER (ORDER BY salary DESC) AS dense,
  RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS
rank_in_dept
FROM emp;
```

 **Likely follow-up:** How would you get the highest-paid employee in each department?

14

How do you compute running totals and compare with the previous row?

Medium 🔥🔥 #window functions

Both are window functions.

Running total: `SUM(amount) OVER (ORDER BY day)`. With an `ORDER BY` the default frame is "from the start up to the current row". Add `PARTITION BY` to restart per customer, or `ROWS BETWEEN 6 PRECEDING AND CURRENT ROW` for a 7-day moving sum.

Previous / next row: `LAG(col)` reads the previous row's value and `LEAD(col)` the next one, so day-over-day change is `amount - LAG(amount) OVER (ORDER BY day)`. Both accept an offset and a default: `LAG(amount, 1, 0)`.

Before window functions this needed a self join or correlated subquery, which is slower and harder to read.

Example

```
DROP TABLE IF EXISTS sales;
CREATE TABLE sales (day TEXT, amount INTEGER);
INSERT INTO sales VALUES ('2024-01-01', 100), ('2024-01-02', 150),
('2024-01-03', 120), ('2024-01-04', 200);

SELECT day, amount,
       SUM(amount) OVER (ORDER BY day)           AS running_total,
       amount - LAG(amount) OVER (ORDER BY day) AS change,
       LEAD(amount) OVER (ORDER BY day)         AS next_day,
       AVG(amount) OVER (ORDER BY day ROWS BETWEEN 1 PRECEDING AND CURRENT
ROW) AS moving_avg
FROM sales;
```

15

What is a CTE? How does a recursive CTE work?

Hard



#cte #recursion

A **CTE** (Common Table Expression) is a named, temporary result defined with `WITH name AS (...)` and used by the query that follows. It makes long queries readable, and one CTE can be referenced several times.

A **recursive CTE** has two parts joined by `UNION ALL`:

- the **anchor**: the starting rows (the CEO, with `manager_id IS NULL`)
- the **recursive member**: joins the table to the CTE itself to get the next level

It repeats until a step returns no new rows. Uses: org charts, category trees, bill of materials, generating date series. Guard against cycles with a depth limit.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

WITH RECURSIVE chain AS (
  SELECT id, name, manager_id, 0 AS level FROM emp WHERE manager_id IS
  NULL
  UNION ALL
  SELECT e.id, e.name, e.manager_id, c.level + 1
  FROM emp e JOIN chain c ON e.manager_id = c.id
)
SELECT level, name FROM chain ORDER BY level, name;
```

? Likely follow-up: CTE vs subquery vs temp table: which would you pick and why?

16

UNION vs UNION ALL ?

Easy 🔥🔥 #set operations

Both stack the results of two queries with the same number of columns and compatible types.

- UNION removes duplicate rows, which needs a sort or hash step.
- UNION ALL keeps everything and is faster.

Use UNION ALL unless you actually need de-duplication, or when you know the sets cannot overlap.

Related set operators: **INTERSECT** (rows in both) and **EXCEPT** (**MINUS** in Oracle: rows in the first but not the second). Column names come from the first query, and **ORDER BY** applies to the combined result.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);

SELECT dept_id FROM emp WHERE salary > 100000
UNION
SELECT dept_id FROM emp WHERE dept_id = 1;      -- 1 row

SELECT dept_id FROM emp WHERE salary > 100000
UNION ALL
SELECT dept_id FROM emp WHERE dept_id = 1;      -- 6 rows
```

17

What is the difference between **DELETE**, **TRUNCATE** and **DROP**?

Easy 🔥🔥 #ddl #dml

- **DELETE** (DML) removes **rows**, optionally with a **WHERE**. It logs each row, fires triggers, can be rolled back, and keeps auto-increment counters.
- **TRUNCATE** (DDL) removes **all rows** at once by deallocating pages. Much faster, no **WHERE**, usually no row triggers, resets identity counters, and is blocked by referencing foreign keys.
- **DROP** removes the **table itself**: data, structure, indexes and constraints.

Rollback: **DELETE** always can inside a transaction. **TRUNCATE** can in PostgreSQL and SQL Server but auto-commits in MySQL and Oracle. SQLite has no **TRUNCATE**; a **DELETE** with no **WHERE** is optimised to do the same job.

18

What is an index, how does it work, and when can it hurt?

Medium 🔥🔥 #indexes #performance

An **index** is a separate sorted structure, usually a **B-tree** (B+ tree), that maps column values to rows. A lookup becomes $O(\log n)$ instead of a full table scan, and range queries and **ORDER BY** can read it in order.

- **Clustered** index: the table rows are stored in that order (one per table; the primary key in InnoDB). **Non-clustered**: a separate structure pointing to rows.
- **Hash** indexes: equality only, no ranges.

Indexes hurt when:

- writes are heavy: every **INSERT** / **UPDATE** / **DELETE** must update each index
- the column has low selectivity (a boolean), so a scan is cheaper
- the query wraps the column in a function (**WHERE LOWER(email) = ...**) or starts a **LIKE** with **%**
- they take disk and memory

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);
CREATE INDEX idx_emp_salary ON emp(salary);

EXPLAIN QUERY PLAN SELECT name FROM emp WHERE salary > 100000; -- uses
the index
EXPLAIN QUERY PLAN SELECT name FROM emp WHERE salary + 0 > 100000; --
expression: full scan
```

? Likely follow-up: Why might the database ignore an index you created?

19

What are composite and covering indexes? Why does column order matter?

Hard



#indexes #performance

A **composite index** covers several columns, like `(dept_id, salary)`. It is sorted by the first column, then by the second within it, like a phone book sorted by surname then first name.

That gives the **leftmost prefix rule**: it helps queries filtering on `dept_id`, or on `dept_id` and `salary`, but not on `salary` alone. Put equality columns first and range or sort columns after.

A **covering index** contains every column a query needs, so the database answers from the index alone without touching the table (an "index-only scan"). Some databases let you add non-key columns with `INCLUDE`.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);
CREATE INDEX idx_dept_salary ON emp(dept_id, salary);

EXPLAIN QUERY PLAN SELECT salary FROM emp WHERE dept_id = 1 AND salary > 100000; -- covering index
EXPLAIN QUERY PLAN SELECT name FROM emp WHERE salary > 100000;
-- not leftmost: scan
```

20

How do you find out why a query is slow?

Medium 🔥🔥 #performance #explain

Start with the **execution plan**: `EXPLAIN` shows what the optimiser intends to do, and `EXPLAIN ANALYZE` (PostgreSQL, MySQL 8) actually runs the query and shows real row counts and timings.

Look for:

- **full table scans** (`Seq Scan`, `type: ALL`) on big tables
- a big gap between `estimated` and `actual rows` (stale statistics: run `ANALYZE`)
- expensive sorts, temp tables, or nested loops over large sets

Then fix: add or adjust an index, make the filter sargable (no functions on the indexed column), select only needed columns, paginate with keyset instead of a large `OFFSET`, or rewrite the query. Also check the slow query log to find which queries matter most.

21

Explain normalisation: 1NF, 2NF, 3NF and BCNF. When do you denormalise?

Medium



#design #normalisation

Normalisation organises tables to remove redundancy, so each fact is stored once and update anomalies disappear.

- **1NF**: atomic values, no repeating groups or comma-separated lists.
- **2NF**: 1NF, and every non-key column depends on the **whole** key (matters for composite keys).
- **3NF**: 2NF, and no **transitive** dependency: non-key columns depend only on the key, not on another non-key column (`dept_name` belongs in `dept`, not `emp`).
- **BCNF**: every determinant is a candidate key; a stricter 3NF.

Summary: **the key, the whole key, and nothing but the key.**

Denormalise deliberately for read-heavy workloads, reporting, or to avoid expensive joins: store a cached total or duplicate a name. You trade write complexity and consistency risk for read speed.

? **Likely follow-up**: Give an example of an update anomaly.

22

What does ACID mean?

Easy



#transactions

ACID is the set of guarantees a transaction gives:

- **Atomicity**: all statements succeed or none do. A failed bank transfer never debits one account without crediting the other.
- **Consistency**: a transaction takes the database from one valid state to another; constraints, keys and checks always hold.
- **Isolation**: concurrent transactions do not see each other's half-finished work (how strictly depends on the isolation level).
- **Durability**: once committed, data **survives a crash**, thanks to the write-ahead log.

In SQL: `BEGIN`, then your statements, then `COMMIT`, or `ROLLBACK` to undo everything.

Example

```
DROP TABLE IF EXISTS account;
CREATE TABLE account (id INTEGER PRIMARY KEY, balance INTEGER CHECK
(balance >= 0));
INSERT INTO account VALUES (1, 100), (2, 50);

BEGIN;
UPDATE account SET balance = balance - 30 WHERE id = 1;
UPDATE account SET balance = balance + 30 WHERE id = 2;
COMMIT;

BEGIN;
UPDATE account SET balance = 0 WHERE id = 2;
ROLLBACK;                                -- undo: balance is still 80
SELECT * FROM account;
```

23

What are transaction isolation levels and the anomalies they prevent?

Hard 🔥🔥 #transactions #concurrency

The anomalies:

- **Dirty read:** reading another transaction's uncommitted change.
- **Non-repeatable read:** reading the same row twice and getting different values because someone committed in between.
- **Phantom read:** re-running a range query and getting new rows.

The levels, weakest to strongest:

- **Read Uncommitted:** allows all three.
- **Read Committed:** no dirty reads. Default in PostgreSQL, Oracle, SQL Server.
- **Repeatable Read:** also no non-repeatable reads. Default in MySQL InnoDB (which also blocks most phantoms).
- **Serializable:** behaves as if transactions ran one at a time.

Higher isolation means **more correctness but less concurrency** (more locking, waits or retries). Most databases implement this with MVCC: readers see a snapshot and do not block writers.

🔗 **Likely follow-up:** What is a lost update, and how do you prevent it?

24

What is a deadlock and how do you avoid it?

Hard



#concurrency #locks

Databases use **locks** to protect data: **shared** locks for reading, **exclusive** locks for writing, at row, page or table level. `SELECT ... FOR UPDATE` takes row locks explicitly (pessimistic locking).

A **deadlock** happens when transaction A holds a lock B needs, while B holds a lock A needs. Neither can continue, so the database detects the cycle and **kills one as the victim**, which must retry.

Avoiding them:

- always lock rows and tables in the same order
- keep transactions short; no user input or network calls inside
- index the columns you filter on, so fewer rows are locked
- use **optimistic locking** (a `version` column checked in the `UPDATE`) where conflicts are rare
- add retry logic for deadlock errors

25

What is a view? How is a materialised view different?

Medium



#views

A **view** is a saved query that behaves like a virtual table. It stores no data; the query runs every time you select from it. Uses: hiding complex joins, exposing only some columns for security, giving a stable interface while tables change.

A **materialised view** **stores the query's result physically**. Reads are as fast as a table and it can be indexed, but the data is stale until you refresh it (`REFRESH MATERIALIZED VIEW` in PostgreSQL, or scheduled).

Use materialised views for expensive aggregations and dashboards where slightly old data is acceptable. MySQL and SQLite do not have them; you emulate them with a summary table updated by a job or trigger.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);
DROP VIEW IF EXISTS dept_payroll;

CREATE VIEW dept_payroll AS
SELECT d.name, COUNT(e.id) AS people, SUM(e.salary) AS payroll
FROM dept d LEFT JOIN emp e ON e.dept_id = d.id
GROUP BY d.id;

SELECT * FROM dept_payroll WHERE people > 0;
```

26

What are stored procedures, functions and triggers?

Medium



#procedures #triggers

- A **stored procedure** is precompiled SQL logic stored in the database and called with `CALL` / `EXEC`. It can take parameters, run several statements and control transactions.
- A **function** returns a value (or table) and can be used inside a query, like `SELECT tax(salary)`; it usually cannot change data.
- A **trigger** runs automatically **before or after an** `INSERT`, `UPDATE` or `DELETE`, for audit logs or derived columns.

Pros: fewer round trips, shared logic across apps, permissions on the procedure instead of tables. Cons: business logic hidden in the database, harder to version, test and debug, and vendor lock-in. Most modern web apps keep logic in the application and use procedures sparingly.

Example

```
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
CREATE TABLE dept (id INTEGER PRIMARY KEY, name TEXT NOT NULL);
CREATE TABLE emp (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  salary INTEGER NOT NULL,
  dept_id INTEGER REFERENCES dept(id),
  manager_id INTEGER REFERENCES emp(id)
);
INSERT INTO dept VALUES (1, 'Engineering'), (2, 'Sales'), (3, 'HR');
INSERT INTO emp VALUES
  (1, 'Asha', 150000, 1, NULL),
  (2, 'Ben', 120000, 1, 1),
  (3, 'Chen', 120000, 1, 1),
  (4, 'Dina', 90000, 2, 1),
  (5, 'Eva', 70000, 2, 4),
  (6, 'Farah', 65000, NULL, 4);
DROP TABLE IF EXISTS salary_log;
CREATE TABLE salary_log (emp_id INTEGER, old INTEGER, new INTEGER);

CREATE TRIGGER log_salary AFTER UPDATE OF salary ON emp
BEGIN
  INSERT INTO salary_log VALUES (OLD.id, OLD.salary, NEW.salary);
END;

UPDATE emp SET salary = salary + 5000 WHERE dept_id = 2;
SELECT * FROM salary_log;
```

27

What is the N+1 query problem?

Medium



#performance #orm

You run **1** query to load a list, then **N** more queries, one per item, to load related data. For example, fetch 100 posts and then the author of each post separately: 101 round trips.

It is usually caused by ORMs lazy-loading relations inside a loop, and it **looks fine in development** but is slow with real data.

Fixes:

- a single **JOIN**
- eager loading in the ORM (**include** in Prisma/Sequelize, **select_related** in Django)

- batching: collect the ids and run one `WHERE id IN (...)` query (the DataLoader pattern in GraphQL)

Spot it by logging queries per request.

28

SQL vs NoSQL: how do you choose?

Easy



#design #nosql

SQL (relational): fixed schema, tables linked by keys, powerful joins, strong ACID transactions. Scales mainly vertically, plus read replicas. Great for structured, related data where correctness matters: payments, orders, inventory.

NoSQL: several families (document like MongoDB, key-value like Redis, wide-column like Cassandra, graph like Neo4j). Flexible schema, designed to **scale horizontally**, often trading strict consistency for availability and speed.

Choose by access patterns, not hype:

- many relationships, ad-hoc queries, transactions: SQL
- huge write volume, evolving or nested data, simple key lookups: NoSQL

Many systems use both: Postgres as the source of truth, Redis for cache.

? Likely follow-up: What does the CAP theorem say?

29

What is the difference between replication and sharding?

Hard



#scaling #architecture

Replication copies the **same data** to several servers. A primary takes writes and replicas serve reads, which gives read scaling and failover. Replication is often asynchronous, so replicas can lag (read-your-own-writes issues).

Sharding (horizontal partitioning) **splits the data** across servers by a **shard key**, such as `user_id` ranges or a hash. Each shard holds a subset, so writes and storage scale too.

Sharding costs: cross-shard joins and transactions become hard, a bad key creates hot spots, and rebalancing is painful. Try indexes, caching, replicas and vertical scaling first. Do not confuse it with **partitioning**, which splits a table within one server.