

SQL & Relational Databases

Joins, aggregation, window functions, indexes, transactions, normalisation

1

Query order and basics

Definition

A relational database stores data in **tables with a fixed schema**, linked by keys, queried with SQL, and protected by ACID transactions (MySQL, PostgreSQL, SQL Server, SQLite).

FROM / JOIN

→

WHERE

→

GROUP BY

→

HAVING

→

SELECT

→

ORDER BY

→

LIMIT

Logical order SQL runs in – why WHERE can't use a SELECT alias

Sample tables used below (run me first)

```
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 * FROM emp;
```

-  **Primary key:** unique, not null row id. **Foreign key:** references another table's key.
-  **WHERE** filters rows before grouping; **HAVING** filters groups after.
-  **NULL** means unknown: use **IS NULL**, never **= NULL**; **COUNT(col)** skips NULLs, **COUNT(*)** doesn't.

<i>Join</i>	<i>Returns</i>
INNER JOIN	only matching rows in both tables
LEFT JOIN	all rows from the left table, NULLs where there's no match
RIGHT JOIN	all rows from the right table
FULL OUTER JOIN	all rows from both
CROSS JOIN	every combination
SELF JOIN	a table joined to itself (employee → manager)

LEFT JOIN and SELF JOIN

```
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);

-- every employee with their department (Farah has none)
SELECT e.name, d.name AS dept
FROM emp e
LEFT JOIN dept d ON d.id = e.dept_id;

-- employees with their manager's name
SELECT e.name AS employee, m.name AS manager
FROM emp e
LEFT JOIN emp m ON m.id = e.manager_id;

-- departments with no employees (anti-join)
SELECT d.name FROM dept d
LEFT JOIN emp e ON e.dept_id = d.id
WHERE e.id IS NULL;
```

GROUP BY, HAVING, subquery

```
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, AVG(e.salary) AS avg_salary
FROM dept d
JOIN emp e ON e.dept_id = d.id
GROUP BY d.id
HAVING COUNT(e.id) >= 2
ORDER BY avg_salary DESC;

-- earning more than the company average
SELECT name, salary FROM emp
WHERE salary > (SELECT AVG(salary) FROM emp);
```

4

Window functions (top interview topic)

Definition

Window functions compute **across related rows without collapsing them** (unlike GROUP BY): `fn()`
`OVER (PARTITION BY ... ORDER BY ...)`.

Second-highest salary and top earner per department

```
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);

-- ROW_NUMBER vs RANK vs DENSE_RANK (Ben and Chen tie)
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
FROM emp;

-- 2nd highest distinct salary
SELECT DISTINCT salary FROM emp ORDER BY salary DESC LIMIT 1 OFFSET 1;

-- highest paid per department
WITH ranked AS (
  SELECT name, dept_id, salary,
    DENSE_RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS r
  FROM emp WHERE dept_id IS NOT NULL
)
SELECT name, dept_id, salary FROM ranked WHERE r = 1;
```

 **LAG()** / **LEAD()** look at the previous / next row (month-over-month change).

 Running total: **SUM(amount) OVER (ORDER BY day)**.

 CTEs (**WITH ...**) name subqueries; recursive CTEs walk hierarchies.

Indexes and performance

Definition

An **index** (usually a B-tree) lets the database find rows without scanning the whole table – faster reads, slightly slower writes and extra storage.

Composite index and EXPLAIN

```
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_dept_salary ON emp (dept_id, salary);

EXPLAIN QUERY PLAN
SELECT name FROM emp WHERE dept_id = 1 AND salary > 100000;
```

- Index columns used in WHERE, JOIN and ORDER BY; high-selectivity columns first.
- Composite index (a, b) helps `WHERE a = ?` and `WHERE a = ? AND b > ?`, but not `WHERE b = ?` alone (leftmost-prefix rule).
- Functions on a column (`WHERE LOWER(email) = ...`) skip a normal index.
- Use `EXPLAIN` / `EXPLAIN ANALYZE` to see scans vs index use; avoid `SELECT *`; fix N+1 queries with joins or batching.

6

Transactions and isolation

 **ACID:** Atomic (all or nothing), Consistent (constraints hold), Isolated (concurrent transactions don't interfere), Durable (committed survives crashes).

Isolation level	Prevents
Read uncommitted	nothing (dirty reads possible)
Read committed (Postgres default)	dirty reads
Repeatable read (MySQL InnoDB default)	+ non-repeatable reads
Serializable	+ phantom reads (safest, slowest)

Atomic transfer

```
DROP TABLE IF EXISTS account;
CREATE TABLE account (id INTEGER PRIMARY KEY, balance INTEGER NOT NULL CHECK
(balance >= 0));
INSERT INTO account VALUES (1, 1000), (2, 200);

BEGIN;
UPDATE account SET balance = balance - 500 WHERE id = 1;
UPDATE account SET balance = balance + 500 WHERE id = 2;
COMMIT;

SELECT * FROM account;
```

Schema design and normalisation

Normal form	Rule
1NF	atomic values, no repeating groups (no "tags: a,b,c" column)
2NF	1NF + every column depends on the whole primary key
3NF	2NF + no column depends on another non-key column

- Many-to-many needs a **junction table** (`student_course(student_id, course_id)`).
- Denormalise deliberately for read-heavy reporting (with care to keep copies in sync).
- Constraints keep data correct: `NOT NULL`, `UNIQUE`, `CHECK`, `FOREIGN KEY ... ON DELETE CASCADE/RESTRICT`.
- Soft delete (`deleted_at`) when you need history or undo.

Revision

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY. LEFT JOIN + IS NULL for "missing" rows. Window functions for rankings. Index WHERE/JOIN columns (leftmost prefix). ACID + isolation levels. Normalise to 3NF, then denormalise on purpose.

Q. WHERE vs HAVING?

A. WHERE filters rows before grouping; HAVING filters groups after aggregation.

Q. INNER vs LEFT JOIN?

A. INNER returns only matching rows; LEFT returns every left-table row, with NULLs where there is no match.

Q. How do you find the second-highest salary?

A. `SELECT DISTINCT salary ... ORDER BY salary DESC LIMIT 1 OFFSET 1`, or `DENSE_RANK() = 2`.

Q. ROW_NUMBER vs RANK vs DENSE_RANK?

A. ROW_NUMBER is always unique; RANK gives ties the same rank and skips the next numbers; DENSE_RANK gives ties the same rank without gaps.

Q. What is an index and its trade-off?

A. A data structure (usually a B-tree) that speeds up lookups; it costs storage and makes writes slower.

Q. What is the leftmost-prefix rule?

A. A composite index (a, b, c) is used for filters on a, a+b, or a+b+c — not for b or c alone.

Q. Explain ACID.

A. Atomicity, Consistency, Isolation, Durability — the guarantees of a database transaction.

Q. What is normalisation?

A. Organising tables to remove duplication and update anomalies: 1NF atomic values, 2NF full key dependency, 3NF no transitive dependency.

Q. DELETE vs TRUNCATE vs DROP?

A. DELETE removes chosen rows (logged, can roll back); TRUNCATE quickly removes all rows; DROP removes the table itself.

Q. What is the N+1 query problem?

A. One query for a list, then one more per item. Fix with a JOIN, an IN (...) batch, or eager loading.

Q. UNION vs UNION ALL?

A. UNION removes duplicates (extra work); UNION ALL keeps all rows.