

The SQL Interview Cheat Sheet

Joins, aggregation, subqueries, window functions and the 15 query patterns interviewers test — every query run on a real sample database (SQLite 3.45), with its actual output.

By **Priyabrat Dalbehera** · GenAI engineer

11 pages

8 parts

59 queries with real output

15 interview problems

What's inside

1 Basics p. 3
SELECT, WHERE, NULL traps, CASE, text & date functions, pagination

2 Aggregation p. 4
COUNT/SUM/AVG, GROUP BY, HAVING, conditional aggregation, UNION

3 Joins p. 5
inner, left, right, full, cross, self, anti- and semi-joins, pitfalls

4 Subqueries & CTEs p. 6
scalar, IN, EXISTS, correlated, NOT IN trap, recursive CTEs

5 Window functions p. 7
ROW_NUMBER, RANK, DENSE_RANK, LAG/LEAD, running totals, frames

6 Changing data p. 8
INSERT, UPDATE, UPSERT, constraints, indexes, transactions, ACID

7 15 classic interview problems p. 9–10
Nth salary, duplicates, streaks, median, pivot, MoM change...

8 Concepts & answers p. 11
keys, normalisation, performance, dialect table, quick answers

How to read it

Each query is followed by the table it really returns. The sample database is on page 2.

Tip & Watch out

Green boxes are good habits. Amber boxes are traps that fail real queries and interviews.

Dialects

Each example is labelled with where it runs unchanged; MySQL, PostgreSQL and SQL Server equivalents are on page 11.

0.1 How to use this sheet

Every query below was **run on SQLite 3.45** against this small company database; the table under each query is its **real output**. Salaries are in ₹ thousands.

The label beside each title says where the query runs **unchanged**.

Standard SQL = SQLite 3.25+, PostgreSQL, MySQL 8+ and SQL Server.
Equivalents for other databases are in the dialect table on page 11.

Watch out: SQL Server returns an **integer** from `AVG()` or `/` on integer columns — write `AVG(salary * 1.0)` to get decimals like the outputs here.

Order a query is **written**: `SELECT` → `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `ORDER BY` → `LIMIT`

Order it **runs**: `FROM/JOIN` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `DISTINCT` → `ORDER BY` → `LIMIT`

Tip: that's why a column alias from `SELECT` can't be used in `WHERE`, but can be used in `ORDER BY`.

0.6 logins

logins

user_id	login_date
1	2026-03-01
1	2026-03-02
1	2026-03-03
1	2026-03-05
2	2026-03-01
2	2026-03-03
3	2026-03-02
3	2026-03-03
3	2026-03-04
3	2026-03-05

0.2 departments

departments

id	name
1	Engineering
2	Data
3	Sales
4	HR

0.3 employees

employees

id	name	dept_id	manager_id	salary	hired
1	Asha	1	NULL	150	2019-03-01
2	Ravi	1	1	120	2020-07-15
3	Meera	2	1	130	2021-01-10
4	Kabir	2	3	135	2022-05-20
5	Zoya	3	1	90	2021-11-01
6	Dev	3	5	90	2023-02-14
7	Isha	NULL	1	70	2024-06-03

`manager_id` points to another employee. Isha has no department; Kabir earns more than his manager.

0.4 customers

customers

id	name	city	email
1	Anil	Pune	anil@mail.com
2	Bela	Delhi	bela@mail.com
3	Chen	Pune	chen@mail.com
4	Dina	Mumbai	dina@mail.com
5	Anil K	Pune	anil@mail.com

Anil K repeats Anil's email; Dina and Anil K never ordered.

0.5 orders

orders

id	customer_id	order_date	amount	status
101	1	2026-01-05	250	paid
102	1	2026-01-20	400	paid
103	2	2026-01-21	150	refunded
104	3	2026-02-02	300	paid
105	2	2026-02-14	500	paid
106	1	2026-02-28	100	paid
107	3	2026-03-03	300	paid

1.1 Select, filter, sort, limit

SQLite · PostgreSQL · MySQL

```
SELECT name, salary
FROM employees
WHERE salary >= 100      -- filter rows
ORDER BY salary DESC, name -- sort: desc, then name asc
LIMIT 3;                -- TOP 3 in SQL Server
```

name	salary
Asha	150
Kabir	135
Meera	130

1.2 Filtering operators

Standard SQL

```
SELECT name, hired FROM employees
WHERE hired BETWEEN '2021-01-01' AND '2022-12-31' -- inclusive
      AND name LIKE '%a%' -- % = any text, _ = one char
      AND dept_id IN (1, 2, 3)
ORDER BY hired;
```

name	hired
Meera	2021-01-10
Zoya	2021-11-01
Kabir	2022-05-20

Also: NOT IN, NOT LIKE, <> / !=. ILIKE (PostgreSQL) = case-insensitive LIKE.

1.3 NULL — the most-asked trap

SQLite · PostgreSQL · MySQL

```
SELECT name, dept_id,
      dept_id = NULL AS eq_null, -- NULL, never true!
      dept_id IS NULL AS is_null,
      COALESCE(dept_id, 0) AS dept_or_0 -- first non-NULL
FROM employees WHERE id IN (6, 7);
```

name	dept_id	eq_null	is_null	dept_or_0
Dev	3	NULL	0	3
Isha	NULL	NULL	1	0

Watch out: x = NULL is never true — use IS NULL. NOT IN with a NULL in the list returns **no rows** (see 4.4). COUNT(col) skips NULLs.

1.4 DISTINCT

Standard SQL

```
SELECT DISTINCT city FROM customers ORDER BY city;
```

city
Delhi
Mumbai
Pune

1.5 CASE — if/else inside a query

SQLite · PostgreSQL · MySQL

```
SELECT name, salary,
      CASE
        WHEN salary >= 130 THEN 'senior'
        WHEN salary >= 100 THEN 'mid'
        ELSE 'junior'
      END AS band
FROM employees ORDER BY salary DESC LIMIT 4;
```

name	salary	band
Asha	150	senior
Kabir	135	senior
Meera	130	senior
Ravi	120	mid

1.6 Text & date functions

SQLite

```
SELECT name,
      UPPER(name) AS upper_name,
      LENGTH(name) AS len,
      SUBSTR(name, 1, 3) AS first3,
      name || ' (' || id || ')' AS label,
      strftime('%Y', hired) AS year_hired
FROM employees WHERE id <= 3;
```

name	upper_name	len	first3	label	year_hired
Asha	ASHA	4	Ash	Asha (1)	2019
Ravi	RAVI	4	Rav	Ravi (2)	2020
Meera	MEERA	5	Mee	Meera (3)	2021

Concatenate: || (Postgres, SQLite, Oracle) · CONCAT() (MySQL, SQL Server). Year: EXTRACT(YEAR FROM d) (Postgres) · YEAR(d) (MySQL / SQL Server).

1.7 Pagination

SQLite · PostgreSQL · MySQL

```
SELECT id, amount FROM orders
ORDER BY id
LIMIT 2 OFFSET 2; -- page 2 with 2 rows per page
```

id	amount
103	150
104	300

Tip: for deep pages, use keyset pagination: WHERE id > :last_seen_id
ORDER BY id LIMIT 20 — OFFSET gets slower as it grows.

2.1 Aggregate functions

Standard SQL

```
SELECT COUNT(*)      AS all_rows,    -- counts every row
       COUNT(dept_id) AS with_dept,   -- skips NULLs
       COUNT(DISTINCT dept_id) AS depts,
       SUM(salary)    AS total,
       ROUND(AVG(salary), 1) AS avg_sal,
       MIN(salary)    AS min_sal,
       MAX(salary)    AS max_sal
FROM employees;
```

all_rows	with_dept	depts	total	avg_sal	min_sal	max_sal
7	6	3	785	112.1	70	150

2.2 GROUP BY — one row per group

Standard SQL

```
SELECT dept_id,
       COUNT(*)      AS headcount,
       SUM(salary)    AS payroll,
       MAX(salary)    AS top_salary
FROM employees
GROUP BY dept_id
ORDER BY payroll DESC;
```

dept_id	headcount	payroll	top_salary
1	2	270	150
2	2	265	135
3	2	180	90
NULL	1	70	70

Watch out: every selected column must be in GROUP BY or inside an aggregate. Note the NULL group (Isha).

2.3 WHERE vs HAVING

Standard SQL

```
SELECT dept_id, ROUND(AVG(salary), 1) AS avg_sal
FROM employees
WHERE hired >= '2020-01-01' -- filters ROWS, before grouping
GROUP BY dept_id
HAVING AVG(salary) > 100    -- filters GROUPS, after grouping
ORDER BY avg_sal DESC;
```

dept_id	avg_sal
2	132.5
1	120.0

2.4 Conditional aggregation

Standard SQL

```
SELECT customer_id,
       COUNT(*)      AS orders,
       SUM(CASE WHEN status = 'paid' THEN amount END) AS paid,
       SUM(CASE WHEN status = 'refunded' THEN 1 ELSE 0 END) AS refunds
FROM orders
GROUP BY customer_id;
```

customer_id	orders	paid	refunds
1	3	750	0
2	2	500	1
3	2	600	0

PostgreSQL also has COUNT(*) FILTER (WHERE status = 'paid').

2.5 Group by an expression (per month)

SQLite

```
SELECT strftime('%Y-%m', order_date) AS month,
       COUNT(*)      AS orders,
       SUM(amount)    AS revenue
FROM orders
WHERE status = 'paid'
GROUP BY month
ORDER BY month;
```

month	orders	revenue
2026-01	2	650
2026-02	3	900
2026-03	1	300

Month bucket: DATE_TRUNC('month', d) (Postgres) · DATE_FORMAT(d, '%Y-%m') (MySQL) · FORMAT(d, 'yyyy-MM') (SQL Server).

2.6 Combine row sets

Standard SQL

```
SELECT city FROM customers WHERE id <= 2
UNION
-- removes duplicates (UNION ALL keeps them)
SELECT city FROM customers WHERE id >= 3
ORDER BY city;
```

city
Delhi
Mumbai
Pune

Also INTERSECT (in both) and EXCEPT (in first, not second; MINUS in Oracle). UNION ALL is faster — use it when duplicates are impossible or wanted.

3.1 Which join?

Join	Returns
INNER JOIN	only rows that match in both tables
LEFT JOIN	all left rows; right side NULL when no match
RIGHT JOIN	all right rows (= LEFT JOIN with tables swapped)
FULL JOIN	all rows from both, NULLs where no match (not in MySQL)
CROSS JOIN	every combination (rows × rows)
self join	a table joined to itself (e.g. employee → manager)

3.2 INNER JOIN

Standard SQL

```
SELECT e.name, d.name AS dept
FROM employees e
JOIN departments d ON d.id = e.dept_id -- JOIN = INNER JOIN
ORDER BY e.id;
```

name	dept
Asha	Engineering
Ravi	Engineering
Meera	Data
Kabir	Data
Zoya	Sales
Dev	Sales

Isha (no department) and HR (no employees) disappear.

3.3 LEFT JOIN — keep everything on the left

Standard SQL

```
SELECT d.name AS dept, COUNT(e.id) AS headcount
FROM departments d
LEFT JOIN employees e ON e.dept_id = d.id
GROUP BY d.id, d.name
ORDER BY d.id;
```

dept	headcount
Engineering	2
Data	2
Sales	2
HR	0

Tip: count a column from the right table — `COUNT(e.id)` — so empty groups show 0, not 1.

3.4 FULL JOIN — unmatched rows from both sides

Standard · not MySQL

```
SELECT e.name AS employee, d.name AS dept
FROM employees e
FULL JOIN departments d ON d.id = e.dept_id
WHERE e.id IS NULL OR d.id IS NULL;
```

employee	dept
Isha	NULL
NULL	HR

3.5 Self join — employee and manager

Standard SQL

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

employee	manager
Asha	NULL
Ravi	Asha
Meera	Asha
Kabir	Meera
Zoya	Asha
Dev	Zoya
Isha	Asha

3.6 Anti-join — rows with NO match

Standard SQL

```
SELECT c.name
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
WHERE o.id IS NULL; -- customers who never ordered
```

name
Dina
Anil K

3.7 Semi-join — rows with AT LEAST one match

Standard SQL

```
SELECT c.name FROM customers c
WHERE EXISTS (SELECT 1 FROM orders o
              WHERE o.customer_id = c.id
                  AND o.amount >= 400);
```

name
Anil
Bela

Tip: EXISTS stops at the first match and never duplicates rows — better than JOIN + DISTINCT.

3.8 CROSS JOIN

Standard SQL

```
SELECT d.name, s.status
FROM departments d
CROSS JOIN (SELECT 'open' AS status UNION ALL SELECT 'closed') s
WHERE d.id <= 2;
```

name	status
Engineering	open
Engineering	closed
Data	open
Data	closed

3.9 Join pitfalls

- A condition on the right table in **WHERE** turns a LEFT JOIN into an INNER JOIN — put it in **ON**.
- Joining two one-to-many tables multiplies rows (fan-out) — aggregate each first.
- Always use table aliases and qualify columns: `e.name`, not `name`.

4.1 Scalar subquery — one value

Standard SQL

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees)
ORDER BY salary DESC;
```

name	salary
Asha	150
Kabir	135
Meera	130
Ravi	120

4.2 IN with a subquery

Standard SQL

```
SELECT name FROM customers
WHERE id IN (SELECT customer_id FROM orders
            WHERE status = 'refunded');
```

name
Bela

4.3 Correlated subquery — runs per row

Standard SQL

```
SELECT e.name, e.dept_id, e.salary
FROM employees e
WHERE e.salary = (SELECT MAX(salary) FROM employees x
                  WHERE x.dept_id = e.dept_id)
ORDER BY e.dept_id;
```

name	dept_id	salary
Asha	1	150
Kabir	2	135
Zoya	3	90
Dev	3	90

Highest earner in each department (ties all returned).

4.4 The NOT IN + NULL trap

Standard SQL

```
SELECT name FROM departments
WHERE id NOT IN (SELECT dept_id FROM employees);
-- returns NO rows: the list contains NULL (Isha)
```

name
(no rows)

Tip: use NOT EXISTS instead — it handles NULLs correctly:

```
SELECT name FROM departments d WHERE NOT EXISTS (
  SELECT 1 FROM employees e WHERE e.dept_id = d.id);
```

name
HR

4.5 CTE — WITH name AS (...)

Standard SQL

```
WITH dept_pay AS (
  SELECT dept_id, SUM(salary) AS payroll
  FROM employees
  WHERE dept_id IS NOT NULL
  GROUP BY dept_id
)
SELECT d.name, p.payroll
FROM dept_pay p
JOIN departments d ON d.id = p.dept_id
WHERE p.payroll > 200;
```

name	payroll
Engineering	270
Data	265

Tip: CTEs make long queries readable — one named step at a time, like variables.

4.6 Recursive CTE — walk a hierarchy

SQLite · PostgreSQL · MySQL 8+

```
WITH RECURSIVE chain AS (
  SELECT id, name, manager_id, 1 AS level
  FROM employees WHERE manager_id IS NULL      -- start: the CEO
  UNION ALL
  SELECT e.id, e.name, e.manager_id, c.level + 1
  FROM employees e
  JOIN chain c ON e.manager_id = c.id          -- next Level down
)
SELECT name, level FROM chain ORDER BY level, name;
```

name	level
Asha	1
Isha	2
Meera	2
Ravi	2
Zoya	2
Dev	3
Kabir	3

4.7 Generate a series with recursion

SQLite

```
WITH RECURSIVE days(d) AS (
  SELECT DATE('2026-03-01')
  UNION ALL
  SELECT DATE(d, '+1 day') FROM days WHERE d < '2026-03-04'
)
SELECT d FROM days;
```

d
2026-03-01
2026-03-02
2026-03-03
2026-03-04

Useful to fill gaps (days with no sales). PostgreSQL: generate_series().

5.1 What a window function does

It computes a value for **every row** using related rows — **without collapsing** them like GROUP BY.

`function() OVER (PARTITION BY ... ORDER BY ... ROWS BETWEEN ...)`

- **PARTITION BY** — restart for each group
- **ORDER BY** — order inside the group
- **frame** — which rows count (e.g. last 3)

5.2 ROW_NUMBER vs RANK vs DENSE_RANK

Standard SQL

```
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 employees;
```

name	salary	row_num	rnk	dense
Asha	150	1	1	1
Kabir	135	2	2	2
Meera	130	3	3	3
Ravi	120	4	4	4
Zoya	90	5	5	5
Dev	90	6	5	5
Isha	70	7	7	6

Ties (Zoya and Dev both earn 90): RANK gives 5, 5 then 7 (skips 6); DENSE_RANK gives 5, 5 then 6; ROW_NUMBER is always unique.

5.3 PARTITION BY — rank inside each group

Standard SQL

```
SELECT dept_id, name, salary,
       RANK() OVER (PARTITION BY dept_id
                   ORDER BY salary DESC) AS rank_in_dept
FROM employees
WHERE dept_id IS NOT NULL
ORDER BY dept_id, rank_in_dept;
```

dept_id	name	salary	rank_in_dept
1	Asha	150	1
1	Ravi	120	2
2	Kabir	135	1
2	Meera	130	2
3	Zoya	90	1
3	Dev	90	1

5.4 Aggregates as windows

Standard SQL

```
SELECT name, dept_id, salary,
       SUM(salary) OVER (PARTITION BY dept_id) AS dept_total,
       ROUND(100.0 * salary
            / SUM(salary) OVER (PARTITION BY dept_id), 1) AS pct
FROM employees
WHERE dept_id IS NOT NULL ORDER BY dept_id;
```

name	dept_id	salary	dept_total	pct
Asha	1	150	270	55.6
Ravi	1	120	270	44.4
Meera	2	130	265	49.1
Kabir	2	135	265	50.9
Zoya	3	90	180	50.0
Dev	3	90	180	50.0

5.5 Running total & moving average

Standard SQL

```
SELECT order_date, amount,
       SUM(amount) OVER (ORDER BY order_date) AS running,
       ROUND(AVG(amount) OVER (
           ORDER BY order_date
           ROWS BETWEEN 2 PRECEDING AND CURRENT ROW), 1) AS avg3
FROM orders WHERE status = 'paid';
```

order_date	amount	running	avg3
2026-01-05	250	250	250.0
2026-01-20	400	650	325.0
2026-02-02	300	950	316.7
2026-02-14	500	1450	400.0
2026-02-28	100	1550	300.0
2026-03-03	300	1850	300.0

5.6 LAG / LEAD — previous and next row

Standard SQL

```
SELECT order_date, amount,
       LAG(amount) OVER (ORDER BY order_date) AS prev_amt,
       amount - LAG(amount) OVER (ORDER BY order_date) AS change,
       LEAD(order_date) OVER (ORDER BY order_date) AS next_date
FROM orders WHERE customer_id = 1;
```

order_date	amount	prev_amt	change	next_date
2026-01-05	250	NULL	NULL	2026-01-20
2026-01-20	400	250	150	2026-02-28
2026-02-28	100	400	-300	NULL

5.7 FIRST_VALUE, NTILE

Standard SQL

```
SELECT name, salary,
       FIRST_VALUE(name) OVER (ORDER BY salary DESC) AS top_earner,
       NTILE(3) OVER (ORDER BY salary DESC) AS bucket
FROM employees;
```

name	salary	top_earner	bucket
Asha	150	Asha	1
Kabir	135	Asha	1
Meera	130	Asha	1
Ravi	120	Asha	2
Zoya	90	Asha	2
Dev	90	Asha	3
Isha	70	Asha	3

Also LAST_VALUE (needs a full frame: ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING), NTH_VALUE, PERCENT_RANK, CUME_DIST.

5.8 Filter on a window result → wrap it

Standard SQL

```
SELECT * FROM (
    SELECT name, salary,
           DENSE_RANK() OVER (ORDER BY salary DESC) AS r
    FROM employees
) t
WHERE r <= 2;
```

name	salary	r
Asha	150	1
Kabir	135	2

Watch out: window functions can't be used in WHERE (they run after it) — use a subquery/CTE, or QUALIFY in Snowflake/BigQuery.

6.1 INSERT

SQLite · PostgreSQL · MySQL

```
INSERT INTO departments (id, name) VALUES (5, 'Legal');
INSERT INTO departments (name)           -- id auto-assigned
SELECT 'Ops';                             -- insert from a query
SELECT * FROM departments WHERE id >= 4;
```

id	name
4	HR
5	Legal
6	Ops

6.2 UPDATE — always with WHERE

Standard SQL

```
UPDATE employees
SET salary = CAST(ROUND(salary * 1.10) AS INTEGER) -- +10%
WHERE dept_id = 3;
SELECT name, salary FROM employees WHERE dept_id = 3;
```

name	salary
Zoya	99
Dev	99

Watch out: an UPDATE/DELETE without WHERE changes **every row**. Run the WHERE as a SELECT first.

Without ROUND, $90 * 1.10$ stores 99.00000000000001 (floating point). For money, use DECIMAL / NUMERIC columns.

6.3 UPDATE using another table

Standard SQL

```
UPDATE employees
SET salary = salary + 5
WHERE dept_id IN (SELECT id FROM departments
                  WHERE name = 'Data');
SELECT name, salary FROM employees WHERE dept_id = 2;
```

name	salary
Meera	135
Kabir	140

6.4 DELETE vs TRUNCATE vs DROP

DELETE FROM t WHERE ...	removes chosen rows; can roll back; fires triggers
TRUNCATE TABLE t	removes all rows fast (not in SQLite — use DELETE FROM t); identity reset varies by database
DROP TABLE t	removes the table itself

6.5 UPSERT — insert or update

SQLite · PostgreSQL

```
INSERT INTO departments (id, name) VALUES (2, 'Data Science')
ON CONFLICT (id) DO UPDATE SET name = excluded.name;
SELECT * FROM departments WHERE id = 2;
```

id	name
2	Data Science

MySQL: ON DUPLICATE KEY UPDATE · SQL Server / Oracle: MERGE.

6.6 Create a table with constraints

SQLite

```
CREATE TABLE projects (
  id          INTEGER PRIMARY KEY,
  name        TEXT      NOT NULL UNIQUE,
  dept_id     INTEGER NOT NULL REFERENCES departments(id),
  budget      INTEGER CHECK (budget >= 0),
  status      TEXT      DEFAULT 'planned',
  created_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

6.7 Indexes

SQLite

6.8 Transactions & ACID

SQLite · PostgreSQL · MySQL

```
BEGIN;
UPDATE orders SET status = 'refunded' WHERE id = 104;
UPDATE customers SET city = 'Goa' WHERE id = 3;
COMMIT;           -- or ROLLBACK; to undo both
```

Atomic	all statements succeed, or none do
Consistent	constraints always hold
Isolated	concurrent transactions don't see half-done work
Durable	committed data survives a crash

Isolation levels (weakest → strongest): READ UNCOMMITTED · READ COMMITTED (Postgres default) · REPEATABLE READ (MySQL default) · SERIALIZABLE.

Q1 Second-highest salary

Standard SQL

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

```
second_highest
135
```

Nth highest with ties handled: `DENSE_RANK()` ... `WHERE r = N` (see 5.8).

Q2 Top 2 earners in each department

Standard SQL

```
SELECT dept_id, name, salary FROM (
  SELECT *, DENSE_RANK() OVER (
    PARTITION BY dept_id ORDER BY salary DESC) AS r
  FROM employees WHERE dept_id IS NOT NULL
) t
WHERE r <= 2
ORDER BY dept_id, salary DESC;
```

dept_id	name	salary
1	Asha	150
1	Ravi	120
2	Kabir	135
2	Meera	130
3	Zoya	90
3	Dev	90

Q3 Employees who earn more than their manager

Standard SQL

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

employee	salary	manager	manager_salary
Kabir	135	Meera	130

Q4 Departments with no employees

Standard SQL

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

```
name
HR
```

Q5 Find duplicate emails

Standard SQL

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

email	copies
anil@mail.com	2

Q6 Delete duplicates — keep the lowest id

SQLite · PostgreSQL · SQL Server

```
DELETE FROM customers
WHERE id NOT IN (SELECT MIN(id) FROM customers
                GROUP BY email);
SELECT id, name, email FROM customers;
```

id	name	email
1	Anil	anil@mail.com
2	Bela	bela@mail.com
3	Chen	chen@mail.com
4	Dina	dina@mail.com

MySQL can't read the table it deletes from: wrap the subquery — `... NOT IN (SELECT id FROM (SELECT MIN(id) id FROM customers GROUP BY email) k)`.

Q7 Running total of revenue

Standard SQL

```
SELECT order_date, amount,
       SUM(amount) OVER (ORDER BY order_date, id) AS running_total
FROM orders
WHERE status = 'paid';
```

order_date	amount	running_total
2026-01-05	250	250
2026-01-20	400	650
2026-02-02	300	950
2026-02-14	500	1450
2026-02-28	100	1550
2026-03-03	300	1850

Q8 Month-over-month revenue change

SQLite

```
WITH monthly AS (
  SELECT strftime('%Y-%m', order_date) AS month,
         SUM(amount) AS revenue
  FROM orders WHERE status = 'paid'
  GROUP BY month
)
SELECT month, revenue,
       revenue - LAG(revenue) OVER (ORDER BY month) AS change,
       ROUND(100.0 * (revenue - LAG(revenue) OVER (ORDER BY month))
            / LAG(revenue) OVER (ORDER BY month), 1) AS pct
FROM monthly;
```

month	revenue	change	pct
2026-01	650	NULL	NULL
2026-02	900	250	38.5
2026-03	300	-600	-66.7

Q9 Customers who never ordered

Standard SQL

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

name
Dina
Anil K

Q10 Top customer (by spend) in each city

Standard SQL

```
WITH spend AS (
  SELECT c.city, c.name, SUM(o.amount) AS total,
         RANK() OVER (PARTITION BY c.city
                     ORDER BY SUM(o.amount) DESC) AS r
  FROM customers c
  JOIN orders o ON o.customer_id = c.id AND o.status = 'paid'
  GROUP BY c.city, c.name
)
SELECT city, name, total FROM spend WHERE r = 1;
```

city	name	total
Delhi	Bela	500
Pune	Anil	750

Q11 Users who logged in 3+ days in a row (gaps & islands)

SQLite

```
WITH numbered AS (
  SELECT user_id, login_date,
         julianday(login_date) - ROW_NUMBER() OVER (
           PARTITION BY user_id ORDER BY login_date) AS grp
  FROM logins
)
SELECT user_id, MIN(login_date) AS streak_start,
       COUNT(*) AS days
FROM numbered
GROUP BY user_id, grp
HAVING COUNT(*) >= 3;
```

user_id	streak_start	days
1	2026-03-01	3
3	2026-03-02	4

Consecutive dates minus their row number give the same value → one group per streak. PostgreSQL: `login_date - ROW_NUMBER() OVER (...)::int`; SQL Server: `DATEADD(day, -ROW_NUMBER() OVER (...), login_date)`.

Q12 Each department's share of total payroll

Standard SQL

```
SELECT d.name, SUM(e.salary) AS payroll,
       ROUND(100.0 * SUM(e.salary) / SUM(SUM(e.salary)) OVER (), 1)
       AS pct_of_total
FROM employees e
JOIN departments d ON d.id = e.dept_id
GROUP BY d.name
ORDER BY payroll DESC;
```

name	payroll	pct_of_total
Engineering	270	37.8
Data	265	37.1
Sales	180	25.2

Q13 Median salary

SQLite · PostgreSQL

```
SELECT AVG(salary) AS median FROM (
  SELECT salary,
         ROW_NUMBER() OVER (ORDER BY salary) AS rn,
         COUNT(*) OVER () AS n
  FROM employees
) t
WHERE rn IN ((n + 1) / 2, (n + 2) / 2);
```

median
120.0

Relies on integer division in `(n + 1) / 2`: in MySQL use `DIV`, in SQL Server integers already divide this way. PostgreSQL: `PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary)`.

Q14 Pivot — revenue per customer per month

SQLite

```
WITH paid AS (
  SELECT customer_id, amount, strftime('%m', order_date) AS mon
  FROM orders WHERE status = 'paid'
)
SELECT customer_id,
       SUM(CASE WHEN mon = '01' THEN amount ELSE 0 END) AS jan,
       SUM(CASE WHEN mon = '02' THEN amount ELSE 0 END) AS feb,
       SUM(CASE WHEN mon = '03' THEN amount ELSE 0 END) AS mar
FROM paid
GROUP BY customer_id;
```

customer_id	jan	feb	mar
1	650	100	0
2	0	500	0
3	0	300	300

PostgreSQL also has `crosstab()`; SQL Server has `PIVOT`.

Q15 First order of each customer

Standard SQL

```
SELECT customer_id, id AS order_id, order_date, amount
FROM (
  SELECT *, ROW_NUMBER() OVER (
    PARTITION BY customer_id ORDER BY order_date) AS rn
  FROM orders
) t
WHERE rn = 1;
```

customer_id	order_id	order_date	amount
1	101	2026-01-05	250
2	103	2026-01-21	150
3	104	2026-02-02	300

8.1 Keys & relationships

Primary key	unique, not NULL, identifies a row
Foreign key	points to a primary key in another table
Unique key	no duplicate values; NULLs allowed (SQL Server: only one NULL)
Composite key	a key made of several columns
Surrogate vs natural	auto-id vs a real-world value (email)
1:1 · 1:N · M:N	M:N needs a junction table (student_courses)

8.2 Normalisation (1NF → 3NF)

1NF	one value per cell, no repeating groups
2NF	1NF + every column depends on the whole key
3NF	2NF + no column depends on another non-key column

Normalise to avoid update anomalies; **denormalise** on purpose for read-heavy analytics (star schema: fact + dimension tables).

8.3 Query performance checklist

- Read the plan: EXPLAIN / EXPLAIN ANALYZE — look for full table scans.
- Index columns in WHERE / JOIN / ORDER BY; avoid functions on indexed columns.
- Select only the columns you need — never SELECT * in production.
- Filter early; aggregate before joining big tables.
- EXISTS over IN for big correlated checks; UNION ALL over UNION.
- Paginate with keyset (WHERE id > ?), not huge OFFSETs.
- Use parameters (? / %s) — faster plan reuse and no SQL injection.

8.4 Dialect cheat table

Need	PostgreSQL	MySQL	SQL Server
first N rows	LIMIT n	LIMIT n	TOP n / OFFSET... FETCH
concatenate	a b	CONCAT(a, b)	a + b / CONCAT
NULL fallback	COALESCE	IFNULL / COALESCE	ISNULL / COALESCE
today	CURRENT_DATE	CURDATE()	CAST(GETDATE() AS date)
month bucket	DATE_TRUNC('month', d)	DATE_FORMAT(d,'%Y- %m')	FORMAT(d,'yyyy- MM')
auto id	GENERATED ... AS IDENTITY	AUTO_INCREMENT	IDENTITY(1,1)
upsert	ON CONFLICT	ON DUPLICATE KEY	MERGE
string agg	STRING_AGG	GROUP_CONCAT	STRING_AGG

8.5 Quick interview answers

WHERE vs HAVING	rows before grouping vs groups after
UNION vs UNION ALL	removes duplicates (slower) vs keeps all
DELETE vs TRUNCATE	chosen rows, logged vs all rows, fast
RANK vs DENSE_RANK	gaps after ties vs no gaps
Clustered index	the table itself is stored in index order (one per table)
View	a saved query; materialised view stores the result
CTE vs subquery	same power; CTE is named, reusable, can recurse
OLTP vs OLAP	many small writes vs big analytical reads
Why NOT NULL?	simpler queries, fewer NULL traps, better plans

8.6 Interview revision checklist

- ☐ Say the logical order: FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT
- ☐ NULL: IS NULL, COALESCE, COUNT(col) vs COUNT(*), the NOT IN trap
- ☐ Pick the join and explain what happens to unmatched rows
- ☐ Top-N per group with ROW_NUMBER / DENSE_RANK + PARTITION BY
- ☐ Running totals, LAG/LEAD, and gaps & islands
- ☐ Duplicates: find with GROUP BY ... HAVING, delete keeping one
- ☐ Explain one index you'd add and why — and how you'd check the plan
- ☐ Know your dialect: LIMIT vs TOP, upsert syntax, date bucketing

8.7 Keep learning — free

- Python Cheat Sheet (17 pages, free)
- Notes, tutorials and architecture guides
- Build a RAG chatbot step by step

learn.aiwithpriyabrat.com