

INTERVIEW PREPARATION GUIDE | 2026 EDITION

100 SQL Interview Questions and Answers

For freshers, intermediate and experienced developers.
66 query questions with real outputs, plus the sample database.

40

Freshers questions

35

Intermediate questions

25

Experienced questions

● **Code and Debug**

codeanddebug.in | [YouTube: @codeanddebug](https://www.youtube.com/@codeanddebug) | Free to share for learning

How to use this guide

The 100 questions are grouped by level: **Freshers** (1 to 40), **Intermediate** (41 to 75) and **Experienced** (76 to 100). Freshers should master the first section completely and read the second. Experienced candidates are expected to answer all three.

Do not memorise answers word for word. Interviewers follow up with "why?" and "what happens if...?". Run every code example yourself, change it, and break it on purpose.

Every query in this guide was executed on the sample database shown on the next page, and the output shown is the real output. Queries marked as syntax examples show MySQL or general syntax and are not run. Where MySQL, PostgreSQL or SQL Server differ, the answer says so.

The interactive version of this guide, with a live SQL playground where every query runs in your browser, flashcards, practice mode and a predict-the-result game, is at codeanddebug.in/blog/sql-interview-questions.

The sample database

All query questions use these four tables. They include deliberate traps: an employee with no department (NULL), a department with no employees, two employees tied on salary, a duplicate customer, and an order from a customer who does not exist.

```
CREATE TABLE departments (
  dept_id INTEGER PRIMARY KEY,
  dept_name TEXT NOT NULL,
  location TEXT
);
INSERT INTO departments VALUES
(1, 'Engineering', 'Bengaluru'),
(2, 'Sales', 'Mumbai'),
(3, 'HR', 'Pune'),
(4, 'Finance', 'Mumbai'),
(5, 'Marketing', 'Delhi');

CREATE TABLE employees (
  emp_id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  dept_id INTEGER REFERENCES departments(dept_id),
  manager_id INTEGER REFERENCES employees(emp_id),
  salary INTEGER,
  hire_date TEXT,
  city TEXT
);
INSERT INTO employees VALUES
(1, 'Rajesh Kumar', 1, NULL, 250000, '2015-04-01', 'Bengaluru'),
(2, 'Priya Sharma', 1, 1, 150000, '2017-06-15', 'Bengaluru'),
(3, 'Amit Verma', 1, 2, 95000, '2019-01-10', 'Hyderabad'),
(4, 'Sneha Iyer', 1, 2, 95000, '2020-03-22', 'Bengaluru'),
(5, 'Karan Mehta', 2, 1, 120000, '2016-09-05', 'Mumbai'),
(6, 'Neha Gupta', 2, 5, 70000, '2021-07-19', 'Mumbai'),
(7, 'Rohit Singh', 2, 5, 65000, '2022-02-14', 'Delhi'),
(8, 'Ananya Rao', 3, 1, 85000, '2018-11-30', 'Pune'),
(9, 'Vikram Joshi', 3, 8, 55000, '2023-05-08', 'Pune'),
(10, 'Pooja Nair', 4, 1, 110000, '2017-01-25', 'Mumbai'),
(11, 'Arjun Patel', 4, 10, 75000, '2020-08-17', 'Ahmedabad'),
(12, 'Kavya Reddy', 1, 2, 120000, '2018-04-12', 'Hyderabad'),
(13, 'Manish Tiwari', NULL, 5, 48000, '2024-01-15', 'Lucknow'),
(14, 'Divya Menon', 2, 6, 72000, '2024-06-01', 'Mumbai');

CREATE TABLE customers (
  customer_id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  email TEXT,
  city TEXT,
  signup_date TEXT
);
INSERT INTO customers VALUES
(101, 'Aarav Shah', 'aarav@mail.com', 'Surat', '2025-01-12'),
(102, 'Isha Kapoor', 'isha@mail.com', 'Delhi', '2025-02-03'),
(103, 'Yash Malhotra', 'yash@mail.com', 'Mumbai', '2025-02-20'),
(104, 'Meera Pillai', 'meera@mail.com', 'Kochi', '2025-03-15'),
(105, 'Dev Choudhary', 'dev@mail.com', 'Jaipur', '2025-04-01'),
(106, 'Riya Sen', 'riya@mail.com', 'Kolkata', '2025-05-09'),
(107, 'Aarav Shah', 'aarav@mail.com', 'Surat', '2025-06-21'),
(108, 'Tara Bose', NULL, 'Kolkata', '2025-07-30');

CREATE TABLE orders (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  order_date TEXT,
  amount INTEGER,
  status TEXT
);
INSERT INTO orders VALUES
(1, 101, '2025-01-15', 2500, 'Delivered'),
(2, 102, '2025-02-05', 1200, 'Delivered'),
(3, 101, '2025-02-18', 3400, 'Delivered'),
(4, 103, '2025-03-02', 800, 'Cancelled'),
(5, 104, '2025-03-20', 5600, 'Delivered'),
(6, 102, '2025-03-25', 2100, 'Delivered'),
(7, 105, '2025-04-10', 950, 'Pending'),
(8, 101, '2025-04-22', 1800, 'Delivered'),
(9, 106, '2025-05-11', 4300, 'Delivered'),
(10, 103, '2025-05-28', 2700, 'Delivered'),
(11, 104, '2025-06-05', 1500, 'Cancelled'),
(12, 102, '2025-06-19', 3900, 'Delivered'),
(13, 107, '2025-06-25', 600, 'Delivered'),
(14, 106, '2025-07-08', 2200, 'Pending'),
(15, 999, '2025-07-12', 1000, 'Delivered'),
(16, 101, '2025-07-30', 4100, 'Delivered');
```

Contents

Freshers (1 to 40)

1. What is SQL?
2. What is the difference between a DBMS and an RDBMS?
3. What are the types of SQL commands (DDL, DML, DQL, DCL, TCL)?
4. What is a primary key?
5. What is a foreign key?
6. What is the difference between a primary key and a unique key?
7. What are constraints in SQL?
8. What is the difference between DELETE, TRUNCATE and DROP?
9. What is the difference between WHERE and HAVING?
10. What is NULL, and why does = NULL not work?
11. What is the difference between CHAR and VARCHAR?
12. What is a JOIN, and what are the different types?
13. Write an INNER JOIN to show each employee with their department name.
14. What does a LEFT JOIN return? Show an example.
15. What is normalization? Explain 1NF, 2NF and 3NF.
16. What is denormalization and when would you use it?
17. What is an index, and how does it speed up queries?
18. What is a view?
19. What is the difference between UNION and UNION ALL?
20. What are aggregate functions?
21. How does GROUP BY work?
22. How do you get the top N rows? (LIMIT, TOP, FETCH FIRST)
23. What does DISTINCT do?
24. How do LIKE and wildcards work?
25. How are BETWEEN and IN used?
26. What are aliases in SQL?
27. What is the difference between COUNT(*), COUNT(column) and COUNT(DISTINCT column)?
28. What is a subquery? Give an example.
29. What is a transaction? Explain ACID properties.
30. How do COMMIT, ROLLBACK and SAVEPOINT work?
31. What is a schema in SQL?
32. What is the difference between OLTP and OLAP?
33. How do you use CASE in SQL?
34. Find the second highest salary.
35. Find employees who joined in a particular year.
36. Find cities that have more than one employee.
37. How do you find duplicate records?
38. Find customers who have never placed an order.
39. Find the total amount spent by each customer, highest first.
40. Delete duplicate rows but keep one copy.

Intermediate (41 to 75)

41. How does a RIGHT JOIN differ from a LEFT JOIN?
42. What is a FULL OUTER JOIN?
43. What is a SELF JOIN? Show each employee with their manager.
44. What is a CROSS JOIN, and when is it useful?
45. What is a correlated subquery? Find employees earning more than their department's average.
46. What is the difference between IN and EXISTS? What is the NOT IN NULL trap?
47. What is a CTE (Common Table Expression)?
48. What are window functions, and how are they different from GROUP BY?
49. What is the difference between ROW_NUMBER, RANK and DENSE_RANK?
50. Find the Nth highest salary (for example, the 3rd highest).
51. Find the top 2 earners in each department.
52. Calculate a running total of revenue by order date.
53. How do LAG and LEAD work? Show month-over-month revenue change.
54. What do COALESCE, IFNULL and NULLIF do?
55. Which string functions are commonly asked?
56. Show monthly revenue for 2025. How do date functions differ across databases?
57. How do you create a pivot table with conditional aggregation?
58. Find employees who earn more than their managers.
59. Find departments that have no employees.

60. Find the highest-paid employee in each department, with their name.
61. Find customers with more than 2 delivered orders and total spend above 5000.
62. Show each department's share of the total salary bill as a percentage.
63. For each customer, find the first and last order date and the days between them.
64. What is a stored procedure?
65. What is the difference between a function and a stored procedure?
66. What is a trigger? Show an example.
67. What is the difference between a clustered and a non-clustered index?
68. What is a composite index, and what is the leftmost prefix rule?
69. What are super keys, candidate keys, alternate keys and composite keys?
70. What does ON DELETE CASCADE do?
71. What types of relationships exist between tables?
72. What do INTERSECT and EXCEPT (MINUS) do?
73. What is the logical order of execution of a SQL query?
74. What is the difference between JOIN and UNION?
75. How do you find the median salary?

Experienced (76 to 100)

76. What are transaction isolation levels and the read phenomena they prevent?
77. What is a deadlock, and how do you prevent one?
78. How do you optimize a slow SQL query?
79. How do you read an execution plan? Show the effect of an index.
80. When can an index hurt performance? What is a covering index?
81. Use a recursive CTE to show the full reporting hierarchy.
82. Generate a calendar of months and show revenue including months with zero sales.
83. Gaps and islands: find streaks of consecutive login days.
84. Calculate a 3-order moving average using a window frame.
85. Find each customer's first order.
86. Find customers who ordered in two consecutive months.
87. Calculate month-over-month revenue growth as a percentage.
88. Show a running total of spend that restarts for each customer.
89. Delete duplicates using ROW_NUMBER().
90. Swap two values in a column with a single UPDATE.
91. Find the Nth highest salary without LIMIT, TOP or window functions.
92. What is BCNF, and when should you denormalize?
93. What are fact and dimension tables? Star vs snowflake schema?
94. What is the difference between partitioning and sharding?
95. When would you choose SQL over NoSQL, and vice versa?
96. What is MVCC? What is optimistic vs pessimistic locking?
97. What is SQL injection, and how do you prevent it?
98. What is a materialized view, and how is it different from a view?
99. Split customers into spending quartiles with NTILE.
100. Find customers whose order amount increased with every order.

Freshers level

SQL fundamentals, keys, NULLs, joins, aggregation and the classic beginner queries asked in campus and entry-level interviews.

Q1. What is SQL?

FRESHERS · CONCEPT · Basics

SQL (Structured Query Language) is the standard language for storing, reading and changing data in relational databases such as MySQL, PostgreSQL, Oracle, SQL Server and SQLite. You describe *what* data you want, and the database engine works out *how* to fetch it, which is why SQL is called a **declarative** language.

With SQL you can create tables, insert, update and delete rows, query and combine data across tables, and control access and transactions. Each database adds its own extensions (MySQL, T-SQL in SQL Server, PL/SQL in Oracle), but the core syntax is standardised by ANSI/ISO and works almost everywhere.

SQL

```
SELECT name, salary
FROM employees
WHERE salary > 100000
ORDER BY salary DESC;
```

OUTPUT

```
name          | salary
-----+-----
Rajesh Kumar  | 250000
Priya Sharma  | 150000
Karan Mehta   | 120000
Kavya Reddy   | 120000
Pooja Nair    | 110000
(5 rows)
```

Q2. What is the difference between a DBMS and an RDBMS?

FRESHERS · CONCEPT · Basics

A **DBMS** (Database Management System) is any software that stores and manages data, which could be files, documents or key-value pairs. An **RDBMS** (Relational DBMS) stores data in **tables** of rows and columns, links tables through keys, enforces integrity rules, and is queried with SQL.

	DBMS	RDBMS
Data storage	Files, documents, hierarchies	Related tables
Relationships	Not enforced	Primary and foreign keys
Normalisation	Not required	Supported
ACID transactions	Not always	Yes
Examples	File systems, early XML stores	MySQL, PostgreSQL, Oracle, SQL Server

Q3. What are the types of SQL commands (DDL, DML, DQL, DCL, TCL)?

FRESHERS · CONCEPT · Basics

Type	Purpose	Commands
DDL (Data Definition)	Define or change structure	CREATE , ALTER , DROP , TRUNCATE , RENAME
DML (Data Manipulation)	Change the data	INSERT , UPDATE , DELETE , MERGE
DQL (Data Query)	Read data	SELECT
DCL (Data Control)	Permissions	GRANT , REVOKE
TCL (Transaction Control)	Manage transactions	COMMIT , ROLLBACK , SAVEPOINT

Some textbooks put [SELECT](#) under DML. Either answer is accepted if you explain it.

Q4. What is a primary key?

FRESHERS · CONCEPT · Keys & constraints

A primary key is a column, or set of columns, that **uniquely identifies each row** in a table. It cannot contain duplicates and cannot be NULL, and a table can have only one primary key. Most databases automatically create an index on it, so lookups by primary key are fast. In our sample database, [emp_id](#) is the primary key of [employees](#), so inserting a second row with an existing id fails:

SQL

```
INSERT INTO employees (emp_id, name, salary)
VALUES (1, 'Duplicate Person', 50000);
```

OUTPUT

```
Error: UNIQUE constraint failed: employees.emp_id
```

Q5. What is a foreign key?

FRESHERS · CONCEPT · Keys & constraints

A foreign key is a column in one table that refers to the primary key of another table. It enforces **referential integrity**: you cannot insert a value that does not exist in the parent table, or delete a parent row that children still point to (unless you specify an action like [ON DELETE CASCADE](#)). Here [employees.dept_id](#) references [departments.dept_id](#), so department 42 is rejected.

Note: SQLite only enforces foreign keys after [PRAGMA foreign_keys = ON](#). MySQL (InnoDB), PostgreSQL and SQL Server enforce them by default.

SQL

```
PRAGMA foreign_keys = ON;
INSERT INTO employees (emp_id, name, dept_id, salary)
VALUES (50, 'New Joiner', 42, 60000);
```

OUTPUT

```
Error: FOREIGN KEY constraint failed
```

Q6. What is the difference between a primary key and a unique key?

FRESHERS · CONCEPT · Keys & constraints

	Primary key	Unique key
Purpose	Identifies each row	Prevents duplicate values in a column
NULL allowed?	No	Yes (usually one NULL, or several depending on the database)
How many per table	Only one	Many
Default index	Clustered in SQL Server and MySQL InnoDB	Non-clustered

Example: `emp_id` is the primary key, while `email` or `pan_number` would be unique keys.

Q7. What are constraints in SQL?

FRESHERS · CONCEPT · Keys & constraints

Constraints are rules on columns that the database enforces automatically, so bad data never gets in:

- **NOT NULL**: a value is required.
- **UNIQUE**: no duplicates.
- **PRIMARY KEY**: **NOT NULL** plus **UNIQUE**, identifies each row.
- **FOREIGN KEY**: value must exist in the referenced table.
- **CHECK**: value must satisfy a condition.
- **DEFAULT**: value used when none is given.

SQL

```
CREATE TABLE accounts (  
  id      INTEGER PRIMARY KEY,  
  holder  TEXT NOT NULL,  
  balance INTEGER DEFAULT 0 CHECK (balance >= 0)  
);  
INSERT INTO accounts (id, holder) VALUES (1, 'Riya');  
INSERT INTO accounts (id, holder, balance) VALUES (2, 'Kabir', -500);
```

OUTPUT

```
Error: CHECK constraint failed: balance >= 0
```

Q8. What is the difference between DELETE, TRUNCATE and DROP?

FRESHERS · CONCEPT · Basics

	DELETE	TRUNCATE	DROP
What it removes	DML	DDL	DDL
WHERE clause	Rows (all or filtered with WHERE)	All rows	The whole table: data, structure, indexes
Speed	Yes	No	No
Triggers fire	Slower, logs each row	Very fast, deallocates pages	Fast
Rollback	Yes	No	No
Transaction / auto-increment	Yes, inside a transaction	In PostgreSQL and SQL Server yes; in MySQL and Oracle no (auto-commit)	Same as TRUNCATE
	Not reset	Reset	Gone with the table

SQLite has no `TRUNCATE`; `DELETE FROM table` without a `WHERE` clause is optimised to do the same job.

SQL (SYNTAX EXAMPLE)

```
DELETE FROM orders WHERE status = 'Cancelled';    -- removes 2 rows
TRUNCATE TABLE orders;                          -- removes all rows (MySQL / PostgreSQL)
DROP TABLE orders;                              -- removes the table itself
```

Q9. What is the difference between WHERE and HAVING?

FRESHERS · QUERY · Filtering

`WHERE` filters **individual rows before grouping**, so it cannot use aggregate functions. `HAVING` filters **groups after GROUP BY**, so it can use aggregates like `COUNT()` or `AVG()`. You can use both in one query: here `WHERE` keeps only employees earning above 60,000, then `HAVING` keeps only departments that still have at least 3 of them.

SQL

```
SELECT dept_id, COUNT(*) AS headcount, AVG(salary) AS avg_salary
FROM employees
WHERE salary > 60000
GROUP BY dept_id
HAVING COUNT(*) >= 3;
```

OUTPUT

dept_id	headcount	avg_salary
1	5	142000
2	4	81750

(2 rows)

Q10. What is NULL, and why does = NULL not work?

FRESHERS · QUERY · Filtering

`NULL` means "unknown" or "missing". It is not zero and not an empty string. Any comparison with `NULL`, including `NULL = NULL`, gives **UNKNOWN**, which `WHERE` treats as false. That is why `WHERE dept_id = NULL` returns nothing. Always use `IS NULL` or `IS NOT NULL`.

SQL

```
SELECT COUNT(*) AS using_equals FROM employees WHERE dept_id = NULL;
SELECT name, dept_id FROM employees WHERE dept_id IS NULL;
```

OUTPUT

```
using_equals
-----
0
(1 row)

name          | dept_id
-----+-----
Manish Tiwari | NULL
(1 row)
```

Q11. What is the difference between CHAR and VARCHAR?

FRESHERS · CONCEPT · Basics

CHAR(n) is **fixed length**: every value takes exactly n characters, padded with spaces. It suits codes of a fixed size, like country codes ('IN') or PIN codes. **VARCHAR(n)** is **variable length**: it stores only the characters you use (up to n) plus a small length prefix, so it saves space for names, emails and addresses.

CHAR can be slightly faster for truly fixed-size data, but VARCHAR is the right default. In Oracle the type is **VARCHAR2**, and PostgreSQL also offers **TEXT** with no length limit.

Q12. What is a JOIN, and what are the different types?

FRESHERS · CONCEPT · Joins

A JOIN combines rows from two or more tables based on a related column, usually a foreign key matching a primary key.

- **INNER JOIN**: only rows that match in both tables.
- **LEFT JOIN**: all rows from the left table, plus matches from the right (NULL where none).
- **RIGHT JOIN**: all rows from the right table, plus matches from the left.
- **FULL OUTER JOIN**: all rows from both tables, matched where possible.
- **CROSS JOIN**: every combination of rows (Cartesian product).
- **SELF JOIN**: a table joined to itself, such as employees to their managers.

Each one has its own question below with a query you can run.

Q13. Write an INNER JOIN to show each employee with their department name.

FRESHERS · QUERY · Joins

INNER JOIN keeps only rows where the join condition matches in both tables. Manish Tiwari has no department (**dept_id** is NULL), so he disappears from the result, and so does the Marketing department, which has no employees. Table aliases (**e**, **d**) keep the query short.

SQL

```
SELECT e.name, d.dept_name, e.salary
FROM employees e
INNER JOIN departments d ON e.dept_id = d.dept_id
ORDER BY d.dept_name, e.salary DESC;
```

OUTPUT

name	dept_name	salary
Rajesh Kumar	Engineering	250000
Priya Sharma	Engineering	150000
Kavya Reddy	Engineering	120000
Amit Verma	Engineering	95000
Sneha Iyer	Engineering	95000
Pooja Nair	Finance	110000
Arjun Patel	Finance	75000
Ananya Rao	HR	85000
Vikram Joshi	HR	55000
Karan Mehta	Sales	120000
Divya Menon	Sales	72000
Neha Gupta	Sales	70000
Rohit Singh	Sales	65000

(13 rows)

Q14. What does a LEFT JOIN return? Show an example.

FRESHERS · QUERY · Joins

LEFT JOIN returns **every row from the left table**, and fills the right table's columns with NULL when there is no match. Here all 5 departments appear, including Marketing with zero employees. Note that `COUNT(e.emp_id)` counts only non-NULL values, which is why Marketing shows 0 instead of 1.

SQL

```
SELECT d.dept_name, COUNT(e.emp_id) AS employees
FROM departments d
LEFT JOIN employees e ON e.dept_id = d.dept_id
GROUP BY d.dept_name
ORDER BY employees DESC;
```

OUTPUT

dept_name	employees
Engineering	5
Sales	4
HR	2
Finance	2
Marketing	0

(5 rows)

Q15. What is normalization? Explain 1NF, 2NF and 3NF.

FRESHERS · CONCEPT · Database design

Normalization organises tables to **reduce duplicate data and avoid update anomalies** (for example, a customer's city stored in 50 order rows and updated in only 49).

- **1NF:** each column holds atomic (single) values, there are no repeating groups, and each row is unique. A `phone_numbers` column holding "98xx, 99xx" breaks 1NF.
- **2NF:** 1NF plus no *partial dependency*: every non-key column depends on the whole primary key, not just part of a composite key.
- **3NF:** 2NF plus no *transitive dependency*: non-key columns depend only on the key, not on other non-key columns. Storing `dept_name` in `employees` breaks 3NF, because it depends on `dept_id`, so we keep it in a separate `departments` table.

BCNF, 4NF and 5NF go further, but 3NF is the usual target for transactional systems.

Q16. What is denormalization and when would you use it?

FRESHERS · CONCEPT · Database design

Denormalization deliberately adds redundant data, for example storing `dept_name` directly in a reporting table, to **avoid expensive joins and speed up reads**. The cost is extra storage and the risk of inconsistent data, because the same fact lives in several places and must be kept in sync.

It is common in reporting databases, data warehouses (star schemas), caches and read-heavy dashboards, where fast reads matter more than write efficiency. Transactional systems (OLTP) usually stay normalized.

Q17. What is an index, and how does it speed up queries?

FRESHERS · CONCEPT · Indexes

An index is a separate data structure, usually a **B-tree**, that keeps a column's values sorted with pointers to the matching rows, like the index at the back of a book. Without one, the database scans every row (a full table scan). With one, it can jump to the matching rows in $O(\log n)$.

The trade-off: indexes use extra storage and slow down **INSERT**, **UPDATE** and **DELETE**, because every index must be updated too. Index the columns you filter, join and sort on often. **EXPLAIN QUERY PLAN** (SQLite) or **EXPLAIN** (MySQL, PostgreSQL) shows whether an index is used:

SQL

```
CREATE INDEX idx_emp_city ON employees(city);
EXPLAIN QUERY PLAN
SELECT name FROM employees WHERE city = 'Mumbai';
```

OUTPUT

```
id | parent | notused | detail
---+-----+-----+-----
3  | 0      | 0       | SEARCH employees USING INDEX idx_emp_city (city=?)
(1 row)
```

Q18. What is a view?

FRESHERS · QUERY · Views

A view is a **saved SELECT query that behaves like a virtual table**. It stores no data itself (unless it is a materialized view); every time you query it, the underlying query runs. Views are used to simplify complex joins, to give users access to only some columns (hiding salary, for example), and to keep a stable interface while underlying tables change.

SQL

```
CREATE VIEW employee_directory AS
SELECT e.name, d.dept_name, e.city
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.dept_id;

SELECT * FROM employee_directory WHERE city = 'Mumbai';
```

OUTPUT

```
name          | dept_name | city
-----+-----+-----
Karan Mehta   | Sales     | Mumbai
Neha Gupta    | Sales     | Mumbai
Pooja Nair    | Finance   | Mumbai
Divya Menon   | Sales     | Mumbai
(4 rows)
```

Q19. What is the difference between UNION and UNION ALL?

FRESHERS · QUERY · Set operations

Both stack the results of two SELECT queries with the same number of columns and compatible types. **UNION removes duplicate rows**, which requires a sort or hash step. **UNION ALL** keeps every row, so it is faster. Use UNION ALL unless you actually need de-duplication. Below, UNION returns each city once, while UNION ALL keeps all 8 rows, duplicates included.

SQL

```
SELECT city FROM employees WHERE city IN ('Mumbai', 'Pune')
UNION
SELECT location FROM departments WHERE location = 'Mumbai';

SELECT COUNT(*) AS union_all_rows FROM (
  SELECT city FROM employees WHERE city IN ('Mumbai', 'Pune')
  UNION ALL
  SELECT location FROM departments WHERE location = 'Mumbai'
);
```

OUTPUT

```
city
-----
Mumbai
Pune
(2 rows)

union_all_rows
-----
8
(1 row)
```

Q20. What are aggregate functions?

FRESHERS · QUERY · Aggregation

Aggregate functions calculate a single value from many rows: [COUNT\(\)](#), [SUM\(\)](#), [AVG\(\)](#), [MIN\(\)](#) and [MAX\(\)](#). They **ignore NULL values** (except [COUNT\(*\)](#)), and they are usually combined with [GROUP BY](#) to get one result per group.

SQL

```
SELECT COUNT(*)      AS total_orders,
       SUM(amount)   AS revenue,
       AVG(amount)   AS avg_order,
       MIN(amount)   AS smallest,
       MAX(amount)   AS largest
FROM orders
WHERE status = 'Delivered';
```

OUTPUT

```
total_orders | revenue | avg_order | smallest | largest
-----+-----+-----+-----+-----
12           | 33200  | 2766.6667 | 600      | 5600
(1 row)
```

Q21. How does GROUP BY work?

FRESHERS · QUERY · Aggregation

[GROUP BY](#) splits rows into groups that share the same value in the listed columns, and aggregate functions then run once per group. Every column in the SELECT list must either be in the GROUP BY or be inside an aggregate function. MySQL can relax this when [ONLY_FULL_GROUP_BY](#) is off, but relying on that gives unpredictable results.

SQL

```
SELECT dept_id,
       COUNT(*)           AS headcount,
       ROUND(AVG(salary)) AS avg_salary,
       MAX(salary)        AS max_salary
FROM employees
WHERE dept_id IS NOT NULL
GROUP BY dept_id
ORDER BY avg_salary DESC;
```

OUTPUT

dept_id	headcount	avg_salary	max_salary
1	5	142000	250000
4	2	92500	110000
2	4	81750	120000
3	2	70000	85000

(4 rows)

Q22. How do you get the top N rows? (LIMIT, TOP, FETCH FIRST)

FRESHERS · QUERY · Filtering

Sort with **ORDER BY**, then limit the rows. The syntax differs by database:

- MySQL, PostgreSQL, SQLite: **LIMIT 3** (with **OFFSET** for paging)
- SQL Server: **SELECT TOP 3 ...**
- Oracle 12c+ and standard SQL: **FETCH FIRST 3 ROWS ONLY**

Without **ORDER BY**, the database may return any 3 rows. Also note that two employees tie at 120000, so "top 3" can be ambiguous; window functions (later questions) handle ties properly.

SQL

```
SELECT name, salary
FROM employees
ORDER BY salary DESC
LIMIT 3;
```

OUTPUT

name	salary
Rajesh Kumar	250000
Priya Sharma	150000
Karan Mehta	120000

(3 rows)

Q23. What does DISTINCT do?

FRESHERS · QUERY · Filtering

DISTINCT removes duplicate rows from the result. With several columns, it applies to the **combination** of those columns. It is often used with COUNT: **COUNT(DISTINCT city)**. Do not use **DISTINCT** to hide duplicate rows caused by a wrong join; fix the join instead.

SQL

```
SELECT DISTINCT city FROM employees ORDER BY city;
SELECT COUNT(DISTINCT city) AS unique_cities FROM employees;
```

OUTPUT

```
city
-----
Ahmedabad
Bengaluru
Delhi
Hyderabad
Lucknow
Mumbai
Pune
(7 rows)

unique_cities
-----
7
(1 row)
```

Q24. How do LIKE and wildcards work?

FRESHERS · QUERY · Filtering

LIKE matches text patterns: `%` means any number of characters and `_` means exactly one character. `'A%'` starts with A, `'%a'` ends with a, and `'%an%'` contains an.

Case sensitivity varies: MySQL and SQLite (for ASCII letters) are case-insensitive by default, PostgreSQL is case-sensitive and offers **ILIKE**. A pattern starting with `%` cannot use a normal index, so it is slow on large tables.

SQL

```
SELECT name FROM employees WHERE name LIKE 'A%';
SELECT name FROM employees WHERE name LIKE '_a%';
```

OUTPUT

```
name
-----
Amit Verma
Ananya Rao
Arjun Patel
(3 rows)

name
-----
Rajesh Kumar
Karan Mehta
Kavya Reddy
Manish Tiwari
(4 rows)
```

Q25. How are BETWEEN and IN used?

FRESHERS · QUERY · Filtering

BETWEEN a AND b is **inclusive** on both ends and works for numbers, dates and text. **IN (...)** is a cleaner way to write several OR conditions. Both have NOT versions. With dates stored as datetimes, be careful: **BETWEEN '2025-01-01' AND '2025-01-31'** misses times later on 31 January, so prefer **>= start AND < next_day**.

SQL

```
SELECT name, salary FROM employees
WHERE salary BETWEEN 70000 AND 95000
AND city IN ('Mumbai', 'Bengaluru', 'Hyderabad')
ORDER BY salary;
```

OUTPUT

name	salary
Neha Gupta	70000
Divya Menon	72000
Amit Verma	95000
Sneha Iyer	95000

(4 rows)

Q26. What are aliases in SQL?

FRESHERS · QUERY · Basics

An alias is a temporary name for a column or table within one query, written with **AS** (optional for tables, and not allowed before table aliases in Oracle). Column aliases make output readable; table aliases shorten joins and are *required* for self joins. In standard SQL you cannot use a column alias in WHERE, because WHERE runs before SELECT, but you can use it in ORDER BY.

SQL

```
SELECT e.name AS employee,  
       e.salary * 12 AS annual_ctc  
FROM employees AS e  
ORDER BY annual_ctc DESC  
LIMIT 3;
```

OUTPUT

employee	annual_ctc
Rajesh Kumar	3000000
Priya Sharma	1800000
Karan Mehta	1440000

(3 rows)

Q27. What is the difference between COUNT(*), COUNT(column) and COUNT(DISTINCT column)?

FRESHERS · QUERY · Aggregation

- **COUNT(*)** counts all rows, including those with NULLs.
- **COUNT(column)** counts rows where that column is **not NULL**.
- **COUNT(DISTINCT column)** counts unique non-NULL values.

Manish Tiwari has a NULL `dept_id`, so the counts differ:

SQL

```
SELECT COUNT(*)           AS all_rows,  
       COUNT(dept_id)     AS with_dept,  
       COUNT(DISTINCT dept_id) AS unique_depts  
FROM employees;
```

OUTPUT

all_rows	with_dept	unique_depts
14	13	4

(1 row)

Q28. What is a subquery? Give an example.

FRESHERS · QUERY · Subqueries

A subquery is a query nested inside another query. It can appear in WHERE (the most common), in FROM (as a derived table), or in SELECT (as a scalar value). Here the inner query computes the company-wide average salary once, and the outer query keeps employees above it.

SQL

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

OUTPUT

name	salary
Rajesh Kumar	250000
Priya Sharma	150000
Karan Mehta	120000
Kavya Reddy	120000
Pooja Nair	110000

(5 rows)

Q29. What is a transaction? Explain ACID properties.

FRESHERS · CONCEPT · Transactions

A transaction is a group of SQL statements that must succeed or fail **as one unit**, like debiting one bank account and crediting another. ACID describes the guarantees:

- **Atomicity:** all statements succeed, or none do.
- **Consistency:** the database moves from one valid state to another, with all constraints satisfied.
- **Isolation:** concurrent transactions do not see each other's half-finished work (the degree depends on the isolation level).
- **Durability:** once committed, changes survive crashes, because they are written to a log on disk.

Q30. How do COMMIT, ROLLBACK and SAVEPOINT work?

FRESHERS · QUERY · Transactions

BEGIN (or **START TRANSACTION** in MySQL) starts a transaction. **COMMIT** makes its changes permanent. **ROLLBACK** undoes everything since **BEGIN**. **SAVEPOINT name** marks a point you can roll back to without undoing the whole transaction. Below, the salary hike for Neha is undone, so her salary is unchanged.

SQL

```
BEGIN;
UPDATE employees SET salary = salary * 2 WHERE emp_id = 6;
SELECT name, salary FROM employees WHERE emp_id = 6;
ROLLBACK;
SELECT name, salary FROM employees WHERE emp_id = 6;
```

OUTPUT

```
name      | salary
-----+-----
Neha Gupta | 140000
(1 row)
```

```
name      | salary
-----+-----
Neha Gupta | 70000
(1 row)
```

Q31. What is a schema in SQL?

FRESHERS · CONCEPT · Basics

A schema is a **named container for database objects** such as tables, views, indexes and procedures, and it also describes their structure. Schemas group related objects and control permissions, for example `sales.orders` and `hr.employees` in the same database.

Meaning differs slightly by product: in MySQL, "schema" and "database" are synonyms; in PostgreSQL and SQL Server a database contains many schemas (`public` and `dbo` are the defaults); in Oracle each user owns a schema.

Q32. What is the difference between OLTP and OLAP?

FRESHERS · CONCEPT · Database design

	OLTP	OLAP
Purpose	Run day-to-day transactions	Analyse data for decisions
Examples	UPI payments, order placement, ticket booking	Sales dashboards, yearly trends
Queries	Many short reads and writes	Few complex reads over huge data
Design	Normalized (3NF)	Denormalized (star or snowflake schema)
Systems	MySQL, PostgreSQL, Oracle	Snowflake, BigQuery, Redshift, Databricks

Data usually flows from OLTP systems into an OLAP warehouse through ETL or ELT pipelines.

Q33. How do you use CASE in SQL?

FRESHERS · QUERY · Functions

`CASE` is SQL's if-else. It returns a value based on conditions checked in order, and the first true one wins. Use it to create categories, map codes to labels, and for conditional aggregation (see the pivot question later). Always add an `ELSE`, otherwise unmatched rows get NULL.

SQL

```
SELECT name, salary,
       CASE
         WHEN salary >= 150000 THEN 'Senior band'
         WHEN salary >= 90000  THEN 'Mid band'
         ELSE 'Junior band'
       END AS pay_band
FROM employees
ORDER BY salary DESC
LIMIT 6;
```

OUTPUT

name	salary	pay_band
Rajesh Kumar	250000	Senior band
Priya Sharma	150000	Senior band
Karan Mehta	120000	Mid band
Kavya Reddy	120000	Mid band
Pooja Nair	110000	Mid band
Amit Verma	95000	Mid band

(6 rows)

Q34. Find the second highest salary.

FRESHERS · QUERY · Query practice

The most asked SQL query question. Two common approaches: take the maximum salary below the overall maximum, or use `DISTINCT` with `LIMIT 1 OFFSET 1`. Both handle ties correctly, because two people with the top salary still leave the second *distinct* value. The `DENSE_RANK()` approach for any Nth salary appears in the intermediate section.

SQL

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

```
SELECT DISTINCT salary AS second_highest
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET 1;
```

OUTPUT

```
second_highest
-----
150000
(1 row)
```

```
second_highest
-----
150000
(1 row)
```

Q35. Find employees who joined in a particular year.

FRESHERS · QUERY · Query practice

Date functions differ between databases, which interviewers like to check:

- MySQL: `YEAR(hire_date) = 2020`
- PostgreSQL / standard SQL: `EXTRACT(YEAR FROM hire_date) = 2020`
- SQL Server: `YEAR(hire_date)` or `DATEPART(year, hire_date)`
- SQLite: `strftime('%Y', hire_date) = '2020'`

Wrapping the column in a function prevents index use, so on large tables a range is faster:
`hire_date >= '2020-01-01' AND hire_date < '2021-01-01'.`

SQL

```
SELECT name, hire_date
FROM employees
WHERE hire_date >= '2020-01-01' AND hire_date < '2021-01-01';
```

OUTPUT

name	hire_date
Sneha Iyer	2020-03-22
Arjun Patel	2020-08-17

(2 rows)

Q36. Find cities that have more than one employee.

FRESHERS · QUERY · Query practice

Group by city, count, and filter the groups with HAVING. This pattern (GROUP BY plus HAVING COUNT) is the basis for many interview questions, such as departments with more than N employees or customers with more than N orders.

SQL

```
SELECT city, COUNT(*) AS employees
FROM employees
GROUP BY city
HAVING COUNT(*) > 1
ORDER BY employees DESC, city;
```

OUTPUT

city	employees
Mumbai	4
Bengaluru	3
Hyderabad	2
Pune	2

(4 rows)

Q37. How do you find duplicate records?

FRESHERS · QUERY · Query practice

Group by the columns that define a duplicate and keep groups with more than one row. Here the same customer (same name and email) signed up twice. To see the full duplicate rows, join this result back to the table or use a window function.

SQL

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

OUTPUT

name	email	copies
Aarav Shah	aarav@mail.com	2

(1 row)

Q38. Find customers who have never placed an order.

FRESHERS · QUERY · Query practice

Classic "anti-join". Three correct ways: `LEFT JOIN ... WHERE right.id IS NULL`, `NOT EXISTS`, or `NOT IN`. Prefer NOT EXISTS or the LEFT JOIN: `NOT IN` returns no rows at all if the subquery contains a NULL (explained in the intermediate section).

SQL

```
SELECT c.customer_id, c.name
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.customer_id
WHERE o.order_id IS NULL;

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

OUTPUT

```
customer_id | name
-----+-----
108         | Tara Bose
(1 row)

name
-----
Tara Bose
(1 row)
```

Q39. Find the total amount spent by each customer, highest first.

FRESHERS · QUERY · Query practice

Join customers to orders, filter only delivered orders, group by customer and sort by the total. Group by the id as well as the name, because two customers can share a name (Aarav Shah appears twice in our data).

SQL

```
SELECT c.customer_id, c.name,
       COUNT(o.order_id) AS orders,
       SUM(o.amount) AS total_spent
FROM customers c
JOIN orders o ON o.customer_id = c.customer_id
WHERE o.status = 'Delivered'
GROUP BY c.customer_id, c.name
ORDER BY total_spent DESC;
```

OUTPUT

```
customer_id | name          | orders | total_spent
-----+-----+-----+-----
101         | Aarav Shah   | 4      | 11800
102         | Isha Kapoor  | 3      | 7200
104         | Meera Pillai | 1      | 5600
106         | Riya Sen     | 1      | 4300
103         | Yash Malhotra | 1      | 2700
107         | Aarav Shah   | 1      | 600
(6 rows)
```

Q40. Delete duplicate rows but keep one copy.

FRESHERS · QUERY · Query practice

Keep the row with the smallest id in each duplicate group and delete the rest. This version works in SQLite, PostgreSQL and SQL Server. MySQL does not allow selecting from the same table in a DELETE subquery, so there you use a self join: `DELETE c1 FROM customers c1 JOIN customers c2 ON c1.email = c2.email AND c1.customer_id > c2.customer_id;` A `ROW_NUMBER()` version appears in the experienced section.

SQL

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

OUTPUT

customer_id	name	email
101	Aarav Shah	aarav@mail.com
102	Isha Kapoor	isha@mail.com
103	Yash Malhotra	yash@mail.com
104	Meera Pillai	meera@mail.com
105	Dev Choudhary	dev@mail.com
106	Riya Sen	riya@mail.com
108	Tara Bose	NULL

(7 rows)

Intermediate level

All join types, subqueries, CTEs, window functions, pivots, triggers, index design and scenario-based queries.

Q41. How does a RIGHT JOIN differ from a LEFT JOIN?

INTERMEDIATE · QUERY · Joins

A RIGHT JOIN keeps every row from the **right** table and fills the left side with NULL when there is no match. A **RIGHT JOIN B** gives the same result as **B LEFT JOIN A**, so most teams simply write LEFT JOINS for readability. RIGHT JOIN is supported in MySQL, PostgreSQL, SQL Server, Oracle, and SQLite 3.39+.

SQL

```
SELECT e.name, d.dept_name
FROM employees e
RIGHT JOIN departments d ON e.dept_id = d.dept_id
WHERE e.emp_id IS NULL;
```

OUTPUT

```
name | dept_name
-----+-----
NULL | Marketing
(1 row)
```

Q42. What is a FULL OUTER JOIN?

INTERMEDIATE · QUERY · Joins

A FULL OUTER JOIN returns all rows from both tables: matched rows side by side, plus unmatched rows from either side with NULLs. Below, customers with no orders appear with a NULL order, and order 15 (from customer 999, who does not exist) appears with a NULL customer. That makes FULL JOIN handy for reconciliation, such as comparing two systems' records.

MySQL has no FULL OUTER JOIN. Emulate it with **LEFT JOIN ... UNION ... RIGHT JOIN**.

SQL

```
SELECT c.name, o.order_id, o.amount
FROM customers c
FULL OUTER JOIN orders o ON o.customer_id = c.customer_id
WHERE c.customer_id IS NULL OR o.order_id IS NULL;
```

OUTPUT

```
name      | order_id | amount
-----+-----+-----
Tara Bose | NULL     | NULL
NULL      | 15       | 1000
(2 rows)
```

Q43. What is a SELF JOIN? Show each employee with their manager.

INTERMEDIATE · QUERY · Joins

A self join joins a table to itself, using two different aliases. It is the standard way to work with hierarchical data stored in one table, like employees and managers or categories and parent categories. A LEFT JOIN keeps Rajesh, the CEO, who has no manager.

SQL

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

OUTPUT

employee	manager
Rajesh Kumar	NULL
Vikram Joshi	Ananya Rao
Manish Tiwari	Karan Mehta
Neha Gupta	Karan Mehta
Rohit Singh	Karan Mehta
Divya Menon	Neha Gupta
Arjun Patel	Pooja Nair
Amit Verma	Priya Sharma
Kavya Reddy	Priya Sharma
Sneha Iyer	Priya Sharma
Ananya Rao	Rajesh Kumar
Karan Mehta	Rajesh Kumar
Pooja Nair	Rajesh Kumar
Priya Sharma	Rajesh Kumar

(14 rows)

Q44. What is a CROSS JOIN, and when is it useful?

INTERMEDIATE · QUERY · Joins

A CROSS JOIN returns the **Cartesian product**: every row of the first table paired with every row of the second, so $m \times n$ rows. It is rarely what you want by accident (a missing join condition creates one), but it is useful on purpose to generate combinations, such as every department paired with every month for a report that must show zero rows too.

SQL

```
SELECT COUNT(*) AS combinations
FROM departments
CROSS JOIN (SELECT 'Q1' AS quarter UNION ALL SELECT 'Q2' UNION ALL SELECT 'Q3' UNION ALL SELECT 'Q4');
```

OUTPUT

```
combinations
-----
20
(1 row)
```

Q45. What is a correlated subquery? Find employees earning more than their department's average.

INTERMEDIATE · QUERY · Subqueries

A **non-correlated** subquery runs once and does not depend on the outer query. A **correlated** subquery refers to a column of the outer query ([e.dept_id](#) below), so logically it runs once per outer row. It is easy to read but can be slow on large tables; a join to a grouped derived table or a window function usually performs better.

SQL

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

OUTPUT

name	dept_id	salary
Rajesh Kumar	1	250000
Priya Sharma	1	150000
Karan Mehta	2	120000
Ananya Rao	3	85000
Pooja Nair	4	110000

(5 rows)

Q46. What is the difference between IN and EXISTS? What is the NOT IN NULL trap?

INTERMEDIATE · QUERY · Subqueries

IN compares a value against a list returned by the subquery. **EXISTS** just checks whether the subquery returns *any* row, and can stop at the first match. Modern optimisers often run them the same way, but **EXISTS** is usually preferred for correlated checks.

The trap: **NOT IN** returns **no rows at all** if the subquery contains a NULL, because **x NOT IN (1, NULL)** evaluates to UNKNOWN. **NOT EXISTS** does not have this problem. Manish's NULL dept_id breaks the first query below:

SQL

```
SELECT COUNT(*) AS not_in_result
FROM departments
WHERE dept_id NOT IN (SELECT dept_id FROM employees);

SELECT dept_name AS not_exists_result
FROM departments d
WHERE NOT EXISTS (SELECT 1 FROM employees e WHERE e.dept_id = d.dept_id);
```

OUTPUT

```
not_in_result
-----
0
(1 row)

not_exists_result
-----
Marketing
(1 row)
```

Q47. What is a CTE (Common Table Expression)?

INTERMEDIATE · QUERY · CTEs

A CTE is a named temporary result defined with **WITH** at the start of a query and used like a table in the main query. CTEs make long queries readable by breaking them into steps, can be referenced more than once, and can be **recursive** (see the experienced section). They exist only for that one query. Supported by PostgreSQL, SQL Server, Oracle, SQLite and MySQL 8+.

SQL

```
WITH dept_stats AS (
  SELECT dept_id, AVG(salary) AS avg_sal
  FROM employees
  GROUP BY dept_id
)
SELECT d.dept_name, ROUND(s.avg_sal) AS avg_salary
FROM departments d
JOIN dept_stats s ON d.dept_id = s.dept_id
ORDER BY avg_salary DESC;
```

OUTPUT

dept_name	avg_salary
Engineering	142000
Finance	92500
Sales	81750
HR	70000

(4 rows)

Q48. What are window functions, and how are they different from GROUP BY?

INTERMEDIATE · QUERY · Window functions

A window function computes a value across a set of related rows (the "window") **without collapsing them** into one row. GROUP BY returns one row per group; a window function keeps every row and adds the calculated value next to it. The `OVER()` clause defines the window: `PARTITION BY` splits it into groups and `ORDER BY` sets the order inside each group. Common ones: `ROW_NUMBER`, `RANK`, `DENSE_RANK`, `LAG`, `LEAD`, `SUM`/`AVG` `OVER`, `NTILE`.

SQL

```
SELECT name, dept_id, salary,
       ROUND(AVG(salary) OVER (PARTITION BY dept_id)) AS dept_avg,
       salary - ROUND(AVG(salary) OVER (PARTITION BY dept_id)) AS diff
FROM employees
WHERE dept_id IN (1, 2)
ORDER BY dept_id, salary DESC;
```

OUTPUT

name	dept_id	salary	dept_avg	diff
Rajesh Kumar	1	250000	142000	108000
Priya Sharma	1	150000	142000	8000
Kavya Reddy	1	120000	142000	-22000
Amit Verma	1	95000	142000	-47000
Sneha Iyer	1	95000	142000	-47000
Karan Mehta	2	120000	81750	38250
Divya Menon	2	72000	81750	-9750
Neha Gupta	2	70000	81750	-11750
Rohit Singh	2	65000	81750	-16750

(9 rows)

Q49. What is the difference between ROW_NUMBER, RANK and DENSE_RANK?

INTERMEDIATE · QUERY · Window functions

All three number rows in a given order, but treat ties differently:

- `ROW_NUMBER()`: always unique (1, 2, 3, 4), ties broken arbitrarily.
- `RANK()`: ties share a rank, then numbers are **skipped** (1, 2, 2, 4).
- `DENSE_RANK()`: ties share a rank with **no gaps** (1, 2, 2, 3).

Use `DENSE_RANK` for "Nth highest salary" questions and `ROW_NUMBER` when you need exactly one row per group.

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_rnk
FROM employees
ORDER BY salary DESC
LIMIT 8;
```

OUTPUT

name	salary	row_num	rnk	dense_rnk
Rajesh Kumar	250000	1	1	1
Priya Sharma	150000	2	2	2
Karan Mehta	120000	3	3	3
Kavya Reddy	120000	4	3	3
Pooja Nair	110000	5	5	4
Amit Verma	95000	6	6	5
Sneha Iyer	95000	7	6	5
Ananya Rao	85000	8	8	6

(8 rows)

Q50. Find the Nth highest salary (for example, the 3rd highest).

INTERMEDIATE · QUERY · Window functions

Use `DENSE_RANK()` so that ties do not skip a level, then filter on the rank in an outer query (window functions cannot be used directly in `WHERE`). Change 3 to any N. Our data has two people tied at 120000, and both are returned for the 3rd highest salary.

SQL

```
WITH ranked AS (
  SELECT name, salary,
         DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
  FROM employees
)
SELECT name, salary FROM ranked WHERE rnk = 3;
```

OUTPUT

name	salary
Karan Mehta	120000
Kavya Reddy	120000

(2 rows)

Q51. Find the top 2 earners in each department.

INTERMEDIATE · QUERY · Window functions

"Top N per group" is one of the most frequent scenario questions. Partition by department, rank inside each partition, then keep ranks up to N. Use `ROW_NUMBER` for exactly N rows per group, or `DENSE_RANK` to include ties (Engineering has a tie at 95000 for 4th place, which would matter for a top 4).

SQL

```
WITH ranked AS (  
    SELECT d.dept_name, e.name, e.salary,  
           ROW_NUMBER() OVER (PARTITION BY e.dept_id ORDER BY e.salary DESC) AS rn  
    FROM employees e  
    JOIN departments d ON d.dept_id = e.dept_id  
)  
SELECT dept_name, name, salary  
FROM ranked  
WHERE rn <= 2  
ORDER BY dept_name, salary DESC;
```

OUTPUT

dept_name	name	salary
Engineering	Rajesh Kumar	250000
Engineering	Priya Sharma	150000
Finance	Pooja Nair	110000
Finance	Arjun Patel	75000
HR	Ananya Rao	85000
HR	Vikram Joshi	55000
Sales	Karan Mehta	120000
Sales	Divya Menon	72000

(8 rows)

Q52. Calculate a running total of revenue by order date.

INTERMEDIATE · QUERY · Window functions

`SUM(...) OVER (ORDER BY ...)` adds each row's value to all the previous ones. Adding `PARTITION BY customer_id` would restart the total for each customer. When the window has an `ORDER BY`, the default frame is "from the start to the current row".

SQL

```
SELECT order_date, amount,  
       SUM(amount) OVER (ORDER BY order_date) AS running_total  
FROM orders  
WHERE status = 'Delivered'  
ORDER BY order_date  
LIMIT 8;
```

OUTPUT

order_date	amount	running_total
2025-01-15	2500	2500
2025-02-05	1200	3700
2025-02-18	3400	7100
2025-03-20	5600	12700
2025-03-25	2100	14800
2025-04-22	1800	16600
2025-05-11	4300	20900
2025-05-28	2700	23600

(8 rows)

Q53. How do LAG and LEAD work? Show month-over-month revenue change.

INTERMEDIATE · QUERY · Window functions

`LAG(col, n)` reads the value from *n* rows **before** the current row; `LEAD(col, n)` reads from *n* rows **after**. They are ideal for comparing with the previous period, like growth, churn or time between events. The first row has no previous value, so LAG returns NULL there (or a default if you pass a third argument).

SQL

```
WITH monthly AS (  
  SELECT strftime('%Y-%m', order_date) AS month, SUM(amount) AS revenue  
  FROM orders  
  WHERE status = 'Delivered'  
  GROUP BY month  
)  
SELECT month, revenue,  
       LAG(revenue) OVER (ORDER BY month) AS prev_month,  
       revenue - LAG(revenue) OVER (ORDER BY month) AS change  
FROM monthly;
```

OUTPUT

month	revenue	prev_month	change
2025-01	2500	NULL	NULL
2025-02	4600	2500	2100
2025-03	7700	4600	3100
2025-04	1800	7700	-5900
2025-05	7000	1800	5200
2025-06	4500	7000	-2500
2025-07	5100	4500	600

(7 rows)

Q54. What do COALESCE, IFNULL and NULLIF do?

INTERMEDIATE · QUERY · Functions

- `COALESCE(a, b, c)` returns the first non-NULL argument. It is standard SQL and works everywhere.
- `IFNULL(a, b)` in MySQL and SQLite, `ISNULL(a, b)` in SQL Server and `NVL(a, b)` in Oracle are two-argument versions.
- `NULLIF(a, b)` returns NULL if *a* equals *b*, otherwise *a*. Its classic use is avoiding division by zero: `x / NULLIF(y, 0)`.

SQL

```
SELECT name, COALESCE(email, 'not given') AS email  
FROM customers  
WHERE customer_id IN (106, 108);  
  
SELECT name, COALESCE(dept_id, 0) AS dept, 100 / NULLIF(0, 0) AS safe_division  
FROM employees  
WHERE emp_id = 13;
```

OUTPUT

```
name      | email
-----+-----
Riya Sen  | riya@mail.com
Tara Bose | not given
(2 rows)

name      | dept | safe_division
-----+-----+-----
Manish Tiwari | 0    | NULL
(1 row)
```

Q55. Which string functions are commonly asked?

INTERMEDIATE · QUERY · Functions

The common ones: [UPPER/LOWER](#), [LENGTH](#) ([LEN](#) in SQL Server), [SUBSTR/SUBSTRING](#), [TRIM](#), [REPLACE](#), [INSTR/POSITION](#), and concatenation, which differs: `||` in PostgreSQL, Oracle and SQLite, [CONCAT\(\)](#) in MySQL, `+` or [CONCAT\(\)](#) in SQL Server. A favourite question: extract the first name from a full name.

SQL

```
SELECT name,
       UPPER(name) AS upper_name,
       LENGTH(name) AS len,
       SUBSTR(name, 1, INSTR(name, ' ') - 1) AS first_name,
       city || ' (' || SUBSTR(city, 1, 3) || ')' AS city_code
FROM employees
LIMIT 4;
```

OUTPUT

```
name      | upper_name | len | first_name | city_code
-----+-----+-----+-----+-----
Rajesh Kumar | RAJESH KUMAR | 12 | Rajesh    | Bengaluru (Ben)
Priya Sharma | PRIYA SHARMA | 12 | Priya     | Bengaluru (Ben)
Amit Verma   | AMIT VERMA  | 10 | Amit      | Hyderabad (Hyd)
Sneha Iyer   | SNEHA IYER  | 10 | Sneha     | Bengaluru (Ben)
(4 rows)
```

Q56. Show monthly revenue for 2025. How do date functions differ across databases?

INTERMEDIATE · QUERY · Functions

Grouping by month is an everyday reporting task, but the function changes per database:

- MySQL: [DATE_FORMAT\(order_date, '%Y-%m'\)](#)
- PostgreSQL: [TO_CHAR\(order_date, 'YYYY-MM'\)](#) or [DATE_TRUNC\('month', order_date\)](#)
- SQL Server: [FORMAT\(order_date, 'yyyy-MM'\)](#)
- SQLite: [strftime\('%Y-%m', order_date\)](#)

SQL

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

OUTPUT

month	orders	revenue
2025-01	1	2500
2025-02	2	4600
2025-03	2	7700
2025-04	2	2750
2025-05	2	7000
2025-06	2	4500
2025-07	3	7300

(7 rows)

Q57. How do you create a pivot table with conditional aggregation?

INTERMEDIATE · QUERY · Aggregation

Put a CASE expression inside an aggregate to turn row values into columns. This works in every database, unlike the **PIVOT** keyword, which exists only in SQL Server and Oracle. Here each order status becomes its own column.

SQL

```
SELECT strftime('%Y-%m', order_date) AS month,
       SUM(CASE WHEN status = 'Delivered' THEN 1 ELSE 0 END) AS delivered,
       SUM(CASE WHEN status = 'Cancelled' THEN 1 ELSE 0 END) AS cancelled,
       SUM(CASE WHEN status = 'Pending' THEN 1 ELSE 0 END) AS pending
FROM orders
GROUP BY month
ORDER BY month;
```

OUTPUT

month	delivered	cancelled	pending
2025-01	1	0	0
2025-02	2	0	0
2025-03	2	1	0
2025-04	1	0	1
2025-05	2	0	0
2025-06	2	1	0
2025-07	2	0	1

(7 rows)

Q58. Find employees who earn more than their managers.

INTERMEDIATE · QUERY · Query practice

A classic self-join question, also on LeetCode (181). Join the table to itself on `manager_id = emp_id` so each row has the employee and their manager side by side, then compare the two salaries. In our data, Divya was hired as a senior specialist and earns slightly more than her team lead Neha.

SQL

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

OUTPUT

employee	salary	manager	manager_salary
Divya Menon	72000	Neha Gupta	70000

(1 row)

Q59. Find departments that have no employees.

INTERMEDIATE · QUERY · Query practice

An anti-join again: LEFT JOIN and keep rows where the employee side is NULL, or use NOT EXISTS. Remember the NOT IN NULL trap from an earlier question: `NOT IN (SELECT dept_id FROM employees)` returns nothing here because one dept_id is NULL.

SQL

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

OUTPUT

```
dept_id | dept_name
-----+-----
5       | Marketing
(1 row)
```

Q60. Find the highest-paid employee in each department, with their name.

INTERMEDIATE · QUERY · Query practice

A simple `GROUP BY dept_id` with `MAX(salary)` gives the salary but not the name, and adding `name` to the SELECT list is invalid in standard SQL. Two correct ways: join back to the grouped maximum, or rank with a window function. The join version also returns ties.

SQL

```
SELECT d.dept_name, e.name, e.salary
FROM employees e
JOIN (SELECT dept_id, MAX(salary) AS max_sal
      FROM employees GROUP BY dept_id) m
  ON e.dept_id = m.dept_id AND e.salary = m.max_sal
JOIN departments d ON d.dept_id = e.dept_id
ORDER BY e.salary DESC;
```

OUTPUT

```
dept_name | name          | salary
-----+-----+-----
Engineering | Rajesh Kumar | 250000
Sales      | Karan Mehta  | 120000
Finance    | Pooja Nair   | 110000
HR         | Ananya Rao   | 85000
(4 rows)
```

Q61. Find customers with more than 2 delivered orders and total spend above 5000.

INTERMEDIATE · QUERY · Query practice

Combine WHERE (filter rows), GROUP BY (one row per customer) and HAVING with two aggregate conditions. Scenario questions like this appear constantly in data analyst interviews.

SQL

```
SELECT customer_id, COUNT(*) AS orders, SUM(amount) AS total
FROM orders
WHERE status = 'Delivered'
GROUP BY customer_id
HAVING COUNT(*) > 2 AND SUM(amount) > 5000;
```

OUTPUT

customer_id	orders	total
101	4	11800
102	3	7200

(2 rows)

Q62. Show each department's share of the total salary bill as a percentage.

INTERMEDIATE · QUERY · Window functions

Compute the department total with GROUP BY, then divide by the grand total, which a window function gives without a second query: `SUM(SUM(salary)) OVER ()` means "sum the grouped sums across the whole result". Multiply by 100.0 rather than 100 to avoid integer division in databases that truncate integers.

SQL

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

OUTPUT

dept_name	dept_salary	pct_of_total
Engineering	710000	52.1
Sales	327000	24
Finance	185000	13.6
HR	140000	10.3

(4 rows)

Q63. For each customer, find the first and last order date and the days between them.

INTERMEDIATE · QUERY · Query practice

MIN and MAX on a date column give the first and last order. Date arithmetic differs by database: `DATEDIFF(last, first)` in MySQL, `last - first` for DATE values in PostgreSQL, `DATEDIFF(day, first, last)` in SQL Server, and `julianday(last) - julianday(first)` in SQLite.

SQL

```
SELECT customer_id,
       MIN(order_date) AS first_order,
       MAX(order_date) AS last_order,
       CAST(julianday(MAX(order_date)) - julianday(MIN(order_date)) AS INTEGER) AS days_between
FROM orders
WHERE customer_id IN (101, 102, 103)
GROUP BY customer_id;
```

OUTPUT

customer_id	first_order	last_order	days_between
101	2025-01-15	2025-07-30	196
102	2025-02-05	2025-06-19	134
103	2025-03-02	2025-05-28	87

(3 rows)

Q64. What is a stored procedure?

INTERMEDIATE · CONCEPT · Stored code

A stored procedure is a named block of SQL (with variables, conditions and loops) **saved inside the database** and run with **CALL** or **EXEC**. Benefits: logic runs close to the data, fewer network round trips, reusable business rules, and permissions can be granted on the procedure instead of the tables. Downsides: harder to version-control and test, and the syntax is vendor-specific (PL/SQL, T-SQL, PL/pgSQL). SQLite does not support stored procedures; this MySQL example is not runnable here:

SQL (SYNTAX EXAMPLE)

```
DELIMITER //
CREATE PROCEDURE give_raise(IN p_dept INT, IN p_pct DECIMAL(5,2))
BEGIN
    UPDATE employees
    SET salary = salary * (1 + p_pct / 100)
    WHERE dept_id = p_dept;
END //
DELIMITER ;

CALL give_raise(2, 10);
```

Q65. What is the difference between a function and a stored procedure?

INTERMEDIATE · CONCEPT · Stored code

	Function	Stored procedure
Returns	Must return a value (scalar or table)	May return nothing, output parameters or result sets
Use in SELECT	Yes	No, called with CALL or EXEC
Change data (INSERT/UPDATE)	Generally not allowed	Yes
Transactions	Cannot manage them	Can COMMIT and ROLLBACK
Typical use	Calculations, formatting	Business processes, batch jobs

Q66. What is a trigger? Show an example.

INTERMEDIATE · QUERY · Stored code

A trigger is code that runs **automatically** when an INSERT, UPDATE or DELETE happens on a table, either BEFORE or AFTER the change. Typical uses: audit logs, enforcing complex rules, and keeping derived columns in sync. Use them sparingly, because hidden logic makes bugs hard to trace. This trigger records every salary change:

SQL

```
CREATE TABLE salary_audit (emp_id INT, old_salary INT, new_salary INT);

CREATE TRIGGER trg_salary_change
AFTER UPDATE OF salary ON employees
BEGIN
    INSERT INTO salary_audit VALUES (OLD.emp_id, OLD.salary, NEW.salary);
END;

UPDATE employees SET salary = salary + 5000 WHERE dept_id = 3;
SELECT * FROM salary_audit;
```

OUTPUT

emp_id	old_salary	new_salary
8	85000	90000
9	55000	60000

(2 rows)

Q67. What is the difference between a clustered and a non-clustered index?

INTERMEDIATE · CONCEPT · Indexes

A **clustered index** decides the physical order of rows on disk: the table data *is* the index's leaf level. So a table can have only one clustered index, usually on the primary key. Range scans on it are very fast.

A **non-clustered index** is a separate structure holding the indexed column values plus a pointer (row locator or primary key) back to the table row. A table can have many of them, and a lookup may need an extra step to fetch the remaining columns, unless the index *covers* the query.

In MySQL InnoDB the primary key is always the clustered index. In PostgreSQL tables are heaps, and **CLUSTER** only reorders them once.

Q68. What is a composite index, and what is the leftmost prefix rule?

INTERMEDIATE · QUERY · Indexes

A composite index covers several columns, for example `(dept_id, salary)`. It is sorted by the first column, then by the second within it. So it helps queries that filter on `dept_id`, or on `dept_id` and `salary`, but generally **not** queries filtering only on `salary`: that is the leftmost prefix rule. Put the column used most often in equality filters first.

SQL

```
CREATE INDEX idx_dept_salary ON employees(dept_id, salary);
EXPLAIN QUERY PLAN SELECT name FROM employees WHERE dept_id = 1 AND salary > 100000;
EXPLAIN QUERY PLAN SELECT name FROM employees WHERE salary > 100000;
```

OUTPUT

id	parent	notused	detail
3	0	0	SEARCH employees USING INDEX idx_dept_salary (dept_id=? AND salary>?)

(1 row)

id	parent	notused	detail
2	0	0	SCAN employees

(1 row)

Q69. What are super keys, candidate keys, alternate keys and composite keys?

INTERMEDIATE · CONCEPT · Keys & constraints

- **Super key:** any set of columns that uniquely identifies a row, even with extra columns, like `(emp_id, name)`.
- **Candidate key:** a *minimal* super key, with no unnecessary columns. A table may have several, like `emp_id` and `email`.
- **Primary key:** the candidate key you choose as the main identifier.
- **Alternate key:** the candidate keys not chosen as primary, usually enforced with **UNIQUE**.

- **Composite key:** a key made of two or more columns, like ([order_id](#), [product_id](#)) in an `order_items` table.
- **Surrogate key:** an artificial id with no business meaning (auto-increment or UUID), as opposed to a **natural key** like a PAN number.

Q70. What does ON DELETE CASCADE do?

INTERMEDIATE · QUERY · Keys & constraints

It defines what happens to child rows when the parent row is deleted. [CASCADE](#) deletes the children automatically. The other options are [SET NULL](#), [SET DEFAULT](#), and [RESTRICT/NO ACTION](#) (block the delete, the default). [ON UPDATE](#) works the same way for key changes. Cascades are convenient but dangerous: one delete can silently remove thousands of rows.

SQL

```
PRAGMA foreign_keys = ON;
CREATE TABLE teams (id INTEGER PRIMARY KEY, name TEXT);
CREATE TABLE members (id INTEGER PRIMARY KEY, team_id INT REFERENCES teams(id) ON DELETE CASCADE, name TEXT);
INSERT INTO teams VALUES (1, 'Backend'), (2, 'Data');
INSERT INTO members VALUES (1, 1, 'Riya'), (2, 1, 'Kabir'), (3, 2, 'Sneha');
DELETE FROM teams WHERE id = 1;
SELECT * FROM members;
```

OUTPUT

```
id | team_id | name
---+-----+-----
3  | 2       | Sneha
(1 row)
```

Q71. What types of relationships exist between tables?

INTERMEDIATE · CONCEPT · Database design

- **One-to-one:** each row in A matches at most one row in B, such as employee and employee_bank_details. Implement with a foreign key that is also UNIQUE.
- **One-to-many:** one department has many employees. The foreign key goes on the "many" side (`employees.dept_id`).
- **Many-to-many:** students and courses. This needs a **junction (bridge) table**, like `enrollments(student_id, course_id)`, with a composite primary key and two foreign keys.

Q72. What do INTERSECT and EXCEPT (MINUS) do?

INTERMEDIATE · QUERY · Set operations

[INTERSECT](#) returns rows present in both queries. [EXCEPT](#) returns rows in the first query that are not in the second; Oracle calls it [MINUS](#). Both remove duplicates. MySQL added them only in 8.0.31, so older MySQL needs joins or EXISTS instead.

SQL

```
SELECT city FROM employees
INTERSECT
SELECT location FROM departments;

SELECT city FROM employees
EXCEPT
SELECT location FROM departments;
```

OUTPUT

```
city
-----
Bengaluru
Delhi
Mumbai
Pune
(4 rows)

city
-----
Ahmedabad
Hyderabad
Lucknow
(3 rows)
```

Q73. What is the logical order of execution of a SQL query?

INTERMEDIATE · CONCEPT · Basics

SQL is written in one order but logically processed in another, which explains many errors:

1. **FROM** and **JOIN**: build the combined rows
2. **WHERE**: filter rows
3. **GROUP BY**: form groups
4. **HAVING**: filter groups
5. **SELECT**: compute columns and aliases (window functions run here)
6. **DISTINCT**: remove duplicates
7. **ORDER BY**: sort
8. **LIMIT** / **OFFSET** / **TOP**: cut rows

That is why a **SELECT** alias cannot be used in **WHERE** in standard SQL, why aggregates cannot appear in **WHERE**, and why window functions need a subquery or CTE to be filtered. The optimiser may physically execute steps differently, but results always match this logical order.

Q74. What is the difference between JOIN and UNION?

INTERMEDIATE · CONCEPT · Joins

A **JOIN** combines tables **horizontally**: it adds columns from another table to each row, matched on a condition. A **UNION** combines query results **vertically**: it stacks rows from two queries that have the same number of columns with compatible types. Use a **JOIN** to see employees with their department names; use a **UNION** to merge last year's and this year's order tables into one list.

Q75. How do you find the median salary?

INTERMEDIATE · QUERY · Window functions

PostgreSQL and Oracle have `PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary)`, but MySQL and SQLite do not, so interviewers ask for a portable version: number the rows in both directions, then average the middle one or two. With 14 employees, the median is the average of the 7th and 8th salaries.

SQL

```
WITH ordered AS (  
  SELECT salary,  
         ROW_NUMBER() OVER (ORDER BY salary) AS rn,  
         COUNT(*) OVER () AS total  
  FROM employees  
)  
SELECT AVG(salary) AS median_salary  
FROM ordered  
WHERE rn IN ((total + 1) / 2, (total + 2) / 2);
```

OUTPUT

```
median_salary  
-----  
90000  
(1 row)
```

Experienced level

Isolation levels, deadlocks, query optimization, recursive CTEs, gaps and islands, warehouse design and SQL injection.

Q76. What are transaction isolation levels and the read phenomena they prevent?

EXPERIENCED · CONCEPT · Transactions

Isolation levels trade consistency for concurrency. The SQL standard defines four, by which anomalies they allow:

Level	Dirty read	Non-repeatable read	Phantom read
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Prevented	Possible	Possible
REPEATABLE READ	Prevented	Prevented	Possible (prevented in practice in MySQL InnoDB and PostgreSQL)
SERIALIZABLE	Prevented	Prevented	Prevented

- **Dirty read:** reading another transaction's uncommitted change, which may be rolled back.
- **Non-repeatable read:** reading the same row twice and getting different values because someone committed an update in between.
- **Phantom read:** running the same range query twice and getting new rows that someone inserted in between.

Defaults: READ COMMITTED in PostgreSQL, SQL Server and Oracle; REPEATABLE READ in MySQL InnoDB.

Q77. What is a deadlock, and how do you prevent one?

EXPERIENCED · CONCEPT · Transactions

A deadlock happens when two transactions each hold a lock the other needs: T1 locks row A and waits for row B, while T2 locks row B and waits for row A. Neither can continue. The database detects the cycle and kills one transaction (the "victim"), which then gets an error and must retry.

Prevention: always access tables and rows in the **same order** across the application, keep transactions short, avoid user interaction inside a transaction, use proper indexes so fewer rows get locked, use `SELECT ... FOR UPDATE` deliberately, and add retry logic for deadlock errors.

Q78. How do you optimize a slow SQL query?

EXPERIENCED · CONCEPT · Performance

1. **Read the execution plan** (`EXPLAIN` / `EXPLAIN ANALYZE`) to find full table scans, bad join orders and large sorts.
2. **Add or fix indexes** on columns used in WHERE, JOIN and ORDER BY. Consider composite and covering indexes.
3. **Make filters sargable:** avoid functions on indexed columns (`WHERE YEAR(d) = 2025` → `WHERE d >= '2025-01-01' AND d < '2026-01-01'`) and leading wildcards (`LIKE '%x'`).
4. **Select only the columns you need**, not `SELECT *`.
5. **Filter early**, and avoid unnecessary DISTINCT, ORDER BY and subqueries that repeat work.
6. **Rewrite correlated subqueries** as joins or window functions.
7. **Paginate with keyset pagination** (`WHERE id > last_id`) instead of large OFFSETs.

8. **Keep statistics fresh** ([ANALYZE](#)), and for very large tables use partitioning, materialized views or caching.

Q79. How do you read an execution plan? Show the effect of an index.

EXPERIENCED · QUERY · Performance

The execution plan shows how the database will run a query: which tables it scans, which indexes it uses, the join algorithms (nested loop, hash, merge) and estimated costs. The key thing to spot is a full **SCAN** of a large table where you expected an index **SEARCH**. Here the same query goes from a full scan to an index search after the index is created. In PostgreSQL use [EXPLAIN ANALYZE](#) to see actual timings and row counts.

SQL

```
EXPLAIN QUERY PLAN SELECT * FROM orders WHERE customer_id = 101;
CREATE INDEX idx_orders_customer ON orders(customer_id);
EXPLAIN QUERY PLAN SELECT * FROM orders WHERE customer_id = 101;
```

OUTPUT

```
id | parent | notused | detail
---+-----+-----+-----
2  | 0      | 0       | SCAN orders
(1 row)

id | parent | notused | detail
---+-----+-----+-----
3  | 0      | 0       | SEARCH orders USING INDEX idx_orders_customer (customer_id=?)
(1 row)
```

Q80. When can an index hurt performance? What is a covering index?

EXPERIENCED · CONCEPT · Indexes

Indexes are not free:

- Every INSERT, UPDATE and DELETE must also update every index, so write-heavy tables slow down.
- They use disk and memory.
- On **low-selectivity** columns (like gender or a boolean), the optimiser often ignores them, because reading most of the table directly is cheaper.
- Duplicate or unused indexes add cost with no benefit.

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"). For example, [CREATE INDEX idx ON orders\(customer_id, order_date, amount\)](#) covers [SELECT order_date, amount FROM orders WHERE customer_id = 101](#). SQL Server and PostgreSQL also support [INCLUDE](#) columns for this.

Q81. Use a recursive CTE to show the full reporting hierarchy.

EXPERIENCED · QUERY · CTEs

A recursive CTE has an **anchor** part (the starting rows, here the CEO) and a **recursive** part that joins back to the CTE itself to find the next level, repeating until no new rows appear. It is the standard way to walk org charts, category trees, bill-of-materials and graph paths. Use [WITH RECURSIVE](#) in PostgreSQL, MySQL and SQLite; SQL Server and Oracle accept plain [WITH](#).

SQL

```
WITH RECURSIVE org AS (  
  SELECT emp_id, name, manager_id, 1 AS level, name AS path  
  FROM employees WHERE manager_id IS NULL  
  UNION ALL  
  SELECT e.emp_id, e.name, e.manager_id, o.level + 1, o.path || ' > ' || e.name  
  FROM employees e  
  JOIN org o ON e.manager_id = o.emp_id  
)  
SELECT level, name, path FROM org  
WHERE level <= 3 AND emp_id IN (1, 2, 4, 5, 6, 14)  
ORDER BY path;
```

OUTPUT

level	name	path
1	Rajesh Kumar	Rajesh Kumar
2	Karan Mehta	Rajesh Kumar > Karan Mehta
3	Neha Gupta	Rajesh Kumar > Karan Mehta > Neha Gupta
2	Priya Sharma	Rajesh Kumar > Priya Sharma
3	Sneha Iyer	Rajesh Kumar > Priya Sharma > Sneha Iyer

(5 rows)

Q82. Generate a calendar of months and show revenue including months with zero sales.

EXPERIENCED · QUERY · CTEs

Reports often need a row for every period, even ones with no data, and GROUP BY alone skips empty months. Generate the full series with a recursive CTE (PostgreSQL also has [generate_series](#)), then LEFT JOIN the data and COALESCE missing values to 0. Our data has no delivered orders outside January to July 2025, so August and September show 0.

SQL

```
WITH RECURSIVE months(m) AS (  
  SELECT '2025-01-01'  
  UNION ALL  
  SELECT date(m, '+1 month') FROM months WHERE m < '2025-09-01'  
)  
SELECT strftime('%Y-%m', m) AS month,  
       COALESCE(SUM(o.amount), 0) AS revenue  
FROM months  
LEFT JOIN orders o  
  ON strftime('%Y-%m', o.order_date) = strftime('%Y-%m', m)  
  AND o.status = 'Delivered'  
GROUP BY month  
ORDER BY month;
```

OUTPUT

month	revenue
2025-01	2500
2025-02	4600
2025-03	7700
2025-04	1800
2025-05	7000
2025-06	4500
2025-07	5100
2025-08	0
2025-09	0

(9 rows)

Q83. Gaps and islands: find streaks of consecutive login days.

EXPERIENCED · QUERY · Window functions

A famous advanced pattern. For consecutive dates, `date - ROW_NUMBER()` stays constant within a streak, so it works as a group key for each "island". Then group by it to get each streak's start, end and length. The same idea finds consecutive seat numbers, uninterrupted subscriptions or missing invoice numbers (the "gaps").

SQL

```
WITH logins(user_id, day) AS (
  VALUES (1,'2025-08-01'),(1,'2025-08-02'),(1,'2025-08-03'),
         (1,'2025-08-06'),(1,'2025-08-07'),(1,'2025-08-10')
),
grp AS (
  SELECT day,
         julianday(day) - ROW_NUMBER() OVER (ORDER BY day) AS island
  FROM logins
)
SELECT MIN(day) AS streak_start, MAX(day) AS streak_end, COUNT(*) AS days
FROM grp
GROUP BY island
ORDER BY streak_start;
```

OUTPUT

streak_start	streak_end	days
2025-08-01	2025-08-03	3
2025-08-06	2025-08-07	2
2025-08-10	2025-08-10	1

(3 rows)

Q84. Calculate a 3-order moving average using a window frame.

EXPERIENCED · QUERY · Window functions

The frame clause controls exactly which rows the window covers relative to the current row. `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW` averages the current order and the two before it. `ROWS` counts physical rows; `RANGE` groups rows with equal ORDER BY values together, which matters when there are ties. Moving averages smooth noisy metrics like daily sales.

SQL

```
SELECT order_id, order_date, amount,  
       ROUND(AVG(amount) OVER (  
         ORDER BY order_date  
         ROWS BETWEEN 2 PRECEDING AND CURRENT ROW), 0) AS moving_avg_3  
FROM orders  
WHERE status = 'Delivered'  
ORDER BY order_date  
LIMIT 7;
```

OUTPUT

order_id	order_date	amount	moving_avg_3
1	2025-01-15	2500	2500
2	2025-02-05	1200	1850
3	2025-02-18	3400	2367
5	2025-03-20	5600	3400
6	2025-03-25	2100	3700
8	2025-04-22	1800	3167
9	2025-05-11	4300	2733

(7 rows)

Q85. Find each customer's first order.

EXPERIENCED · QUERY · Window functions

"First, latest, or Nth event per entity" is a top-N-per-group problem. Number the orders within each customer by date with `ROW_NUMBER()`, then keep row 1. Change the `ORDER BY` to `DESC` for the latest order. PostgreSQL also offers the shortcut `SELECT DISTINCT ON (customer_id) ... ORDER BY customer_id, order_date`.

SQL

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

OUTPUT

customer_id	order_id	order_date	amount
101	1	2025-01-15	2500
102	2	2025-02-05	1200
103	4	2025-03-02	800
104	5	2025-03-20	5600
105	7	2025-04-10	950
106	9	2025-05-11	4300
107	13	2025-06-25	600
999	15	2025-07-12	1000

(8 rows)

Q86. Find customers who ordered in two consecutive months.

EXPERIENCED · QUERY · Window functions

Get each customer's distinct order months, then use LAG to compare each month with the previous one. A difference of exactly one month means consecutive activity. This is the building block of retention and cohort analysis. Converting months to a number (year × 12 + month) makes the difference easy to compute in any database.

SQL

```
WITH months AS (  
  SELECT DISTINCT customer_id,  
    CAST(strftime('%Y', order_date) AS INT) * 12 + CAST(strftime('%m', order_date) AS INT) AS month_no  
  FROM orders  
,  
diffs AS (  
  SELECT customer_id, month_no,  
    month_no - LAG(month_no) OVER (PARTITION BY customer_id ORDER BY month_no) AS gap  
  FROM months  
)  
SELECT DISTINCT customer_id FROM diffs WHERE gap = 1 ORDER BY customer_id;
```

OUTPUT

```
customer_id  
-----  
101  
102  
(2 rows)
```

Q87. Calculate month-over-month revenue growth as a percentage.

EXPERIENCED · QUERY · Window functions

Aggregate to monthly revenue first, then use LAG to bring the previous month's value onto the same row. Growth = (current – previous) / previous × 100. Use `NULLIF(previous, 0)` to avoid dividing by zero, and multiply by 100.0 to force decimal division.

SQL

```
WITH monthly AS (  
  SELECT strftime('%Y-%m', order_date) AS month, SUM(amount) AS revenue  
  FROM orders WHERE status = 'Delivered'  
  GROUP BY month  
)  
SELECT month, revenue,  
  ROUND(100.0 * (revenue - LAG(revenue) OVER (ORDER BY month))  
    / NULLIF(LAG(revenue) OVER (ORDER BY month), 0), 1) AS growth_pct  
FROM monthly;
```

OUTPUT

month	revenue	growth_pct
2025-01	2500	NULL
2025-02	4600	84
2025-03	7700	67.4
2025-04	1800	-76.6
2025-05	7000	288.9
2025-06	4500	-35.7
2025-07	5100	13.3

(7 rows)

Q88. Show a running total of spend that restarts for each customer.

EXPERIENCED · QUERY · Window functions

Add **PARTITION BY** to the running-total window, and the sum restarts for every customer. This answers questions like "when did each customer cross ₹5,000 lifetime spend?": filter the running total in an outer query.

SQL

```
SELECT customer_id, order_date, amount,
       SUM(amount) OVER (PARTITION BY customer_id ORDER BY order_date) AS lifetime_spend
FROM orders
WHERE status = 'Delivered' AND customer_id IN (101, 102)
ORDER BY customer_id, order_date;
```

OUTPUT

customer_id	order_date	amount	lifetime_spend
101	2025-01-15	2500	2500
101	2025-02-18	3400	5900
101	2025-04-22	1800	7700
101	2025-07-30	4100	11800
102	2025-02-05	1200	1200
102	2025-03-25	2100	3300
102	2025-06-19	3900	7200

(7 rows)

Q89. Delete duplicates using ROW_NUMBER().

EXPERIENCED · QUERY · Query practice

Number the rows within each duplicate group, then delete every row with a number above 1. This version is flexible, because the ORDER BY decides which copy survives (here, the earliest signup). SQL Server can delete straight from the CTE (**DELETE FROM cte WHERE rn > 1**); PostgreSQL, MySQL and SQLite need the ids as below.

SQL

```
WITH ranked AS (
  SELECT customer_id,
         ROW_NUMBER() OVER (PARTITION BY email ORDER BY signup_date) AS rn
  FROM customers
  WHERE email IS NOT NULL
)
DELETE FROM customers
WHERE customer_id IN (SELECT customer_id FROM ranked WHERE rn > 1);

SELECT COUNT(*) AS customers_left FROM customers;
```

OUTPUT

customers_left
7

(1 row)

Q90. Swap two values in a column with a single UPDATE.

EXPERIENCED · QUERY · Query practice

A tricky classic (LeetCode 627 swaps 'm' and 'f'). A single UPDATE with CASE does it in one pass, with no temporary value needed, because each row is evaluated against its original value. Here every Pending order becomes Delivered and every Delivered order becomes Pending.

SQL

```
UPDATE orders
SET status = CASE status
                WHEN 'Pending' THEN 'Delivered'
                WHEN 'Delivered' THEN 'Pending'
                ELSE status
            END;
SELECT status, COUNT(*) AS orders FROM orders GROUP BY status ORDER BY status;
```

OUTPUT

```
status | orders
-----+-----
Cancelled | 2
Delivered | 2
Pending   | 12
(3 rows)
```

Q91. Find the Nth highest salary without LIMIT, TOP or window functions.

EXPERIENCED · QUERY · Query practice

Interviewers sometimes ban LIMIT and window functions to test logic. A correlated subquery counts how many distinct salaries are higher than each salary. The Nth highest has exactly $N - 1$ distinct salaries above it. For $N = 3$:

SQL

```
SELECT DISTINCT salary
FROM employees e1
WHERE 2 = (
    SELECT COUNT(DISTINCT e2.salary)
    FROM employees e2
    WHERE e2.salary > e1.salary
);
```

OUTPUT

```
salary
-----
120000
(1 row)
```

Q92. What is BCNF, and when should you denormalize?

EXPERIENCED · CONCEPT · Database design

Boyce-Codd Normal Form (BCNF) is a stricter 3NF: for every functional dependency $X \rightarrow Y$, X must be a super key. It catches rare cases 3NF allows, such as a table where a non-key column determines part of a composite key (for example, each instructor teaches one subject, but a subject has several instructors).

Denormalize when read performance matters more than write simplicity: reporting tables, data warehouses, search indexes, or pre-computed counters like `order_count` on a customers table. Do it deliberately, and keep the redundant data in sync with triggers, application code or scheduled jobs.

Q93. What are fact and dimension tables? Star vs snowflake schema?

EXPERIENCED · CONCEPT · Database design

In a data warehouse, a **fact table** stores measurable events (sales amount, quantity, clicks) with foreign keys to **dimension tables**, which hold descriptive context (date, customer, product, store).

- **Star schema:** one central fact table joined directly to denormalized dimension tables. Fewer joins, simpler and faster queries, and the most common choice.
- **Snowflake schema:** dimensions are normalized into sub-tables (product → category → department). Less redundancy, but more joins.

Expect this question in data engineer and data analyst interviews, along with slowly changing dimensions (SCD Type 1 overwrites, Type 2 keeps history rows).

Q94. What is the difference between partitioning and sharding?

EXPERIENCED · CONCEPT · Database design

Partitioning splits one large table into smaller pieces **within the same database server**, by range (order_date by month), list (region) or hash. Queries that filter on the partition key read only the relevant partitions ("partition pruning"), and old data can be dropped quickly by partition.

Sharding splits data across **multiple database servers**, for example users with ids 1 to 1M on server A and the rest on server B. It scales writes and storage horizontally, but makes cross-shard joins, transactions and rebalancing much harder, so it is used only when one server cannot cope.

Q95. When would you choose SQL over NoSQL, and vice versa?

EXPERIENCED · CONCEPT · Database design

	SQL (relational)	NoSQL
Data model	Tables with a fixed schema	Documents, key-value, wide-column, graph
Consistency	Strong ACID transactions	Often eventual consistency (varies)
Scaling	Mostly vertical, read replicas	Designed for horizontal scaling
Queries	Powerful joins and aggregations	Fast lookups by key, limited joins
Good for	Payments, orders, inventory, anything relational	Catalogues, sessions, logs, real-time feeds
Examples	PostgreSQL, MySQL, Oracle	MongoDB, Redis, Cassandra, DynamoDB

Many systems use both: PostgreSQL for orders and payments, Redis for caching and sessions.

Q96. What is MVCC? What is optimistic vs pessimistic locking?

EXPERIENCED · CONCEPT · Transactions

MVCC (Multi-Version Concurrency Control), used by PostgreSQL, MySQL InnoDB and Oracle, keeps several versions of a row. Readers see a consistent snapshot without blocking writers, and writers do not block readers. Old versions are cleaned up later (VACUUM in PostgreSQL, the undo log purge in InnoDB).

- **Pessimistic locking** locks the row before changing it (`SELECT ... FOR UPDATE`), assuming conflicts are likely. Safe, but reduces concurrency.
- **Optimistic locking** does not lock. It checks at update time that nobody else changed the row, usually with a version column: `UPDATE ... SET qty = 5, version = version + 1 WHERE id = 7 AND version = 3`. If 0 rows are updated, retry. Best when conflicts are rare.

Q97. What is SQL injection, and how do you prevent it?

EXPERIENCED · CONCEPT · Security

SQL injection happens when user input is pasted directly into a SQL string, letting an attacker change the query. If a login query is built as `"... WHERE user = '" + input + "'"` and the input is `' OR '1'='1'`, the condition is always true and the attacker logs in without a password. It can also be used to read or delete entire tables.

Prevention:

- **Always use parameterised queries** (prepared statements), where input is sent separately from the SQL text, for example `cursor.execute("SELECT * FROM users WHERE email = %s", (email,))`.
- Use an ORM or query builder that parameterises for you.
- Give the application database user the least privileges it needs.
- Validate input types and lengths, and never show raw database errors to users.

SQL (SYNTAX EXAMPLE)

```
-- Vulnerable: input concatenated into the query
SELECT * FROM users WHERE email = '' OR '1'='1';

-- Safe: placeholder, value bound separately by the driver
SELECT * FROM users WHERE email = ?;
```

Q98. What is a materialized view, and how is it different from a view?

EXPERIENCED · CONCEPT · Views

A normal view stores only the query, and runs it every time you read from the view. A **materialized view** stores the query's **result** physically, like a table, so reads are very fast. The data can become stale and must be refreshed, on a schedule or on demand (`REFRESH MATERIALIZED VIEW` in PostgreSQL). Oracle can refresh automatically; SQL Server's equivalent is an indexed view; MySQL has none, so people use summary tables updated by jobs. Use them for expensive aggregations behind dashboards.

Q99. Split customers into spending quartiles with NTILE.

EXPERIENCED · QUERY · Query practice

NTILE(n) divides ordered rows into n roughly equal buckets and returns the bucket number. It is used for segmentation (top 25% customers), percentile-based pricing and A/B group assignment. Here bucket 1 holds the highest spenders.

SQL

```
WITH spend AS (  
  SELECT customer_id, SUM(amount) AS total  
  FROM orders WHERE status = 'Delivered'  
  GROUP BY customer_id  
)  
SELECT customer_id, total,  
       NTILE(4) OVER (ORDER BY total DESC) AS quartile  
FROM spend  
ORDER BY total DESC;
```

OUTPUT

customer_id	total	quartile
101	11800	1
102	7200	1
104	5600	2
106	4300	2
103	2700	3
999	1000	3
107	600	4

(7 rows)

Q100. Find customers whose order amount increased with every order.

EXPERIENCED · QUERY · Query practice

Compare each order with the customer's previous one using LAG, flag any order that did not increase, and keep customers with no such flag and at least two orders. Scenario questions like this test whether you can combine window functions with aggregation in clean steps.

SQL

```
WITH seq AS (  
  SELECT customer_id, amount,  
         LAG(amount) OVER (PARTITION BY customer_id ORDER BY order_date) AS prev  
  FROM orders  
  WHERE status = 'Delivered'  
)  
SELECT customer_id  
FROM seq  
GROUP BY customer_id  
HAVING COUNT(*) >= 2  
      AND SUM(CASE WHEN prev IS NOT NULL AND amount <= prev THEN 1 ELSE 0 END) = 0;
```

OUTPUT

customer_id
102

(1 row)

Keep practising

Revise one level at a time, and explain each answer out loud as if you were in the interview.

Write every query by hand before running it, and practise on the live playground at codeanddebug.in/blog/sql-interview-questions.

Watch the video explanations on YouTube: [@codeanddebug](https://www.youtube.com/@codeanddebug).