SQL INTERVIEW · @CHALKESAHIL

60 SQL queries from easy to hard, with the reasoning behind each

This guide is for students and early-career engineers preparing for SQL rounds in placements and job switches. Every query runs on one small shared schema, so you can copy it, run it and change it. After working through it you should be able to explain each clause, spot the usual traps and write the classic interview queries from scratch.

How to use this PDF

1. Create the four tables from the preface in any SQL tool, SQLite is enough, and insert a few rows of your own.
2. Read the task, then try to write the query yourself before you look at the example.
3. Run the example, then change one thing, such as the condition or the sort, and predict the result before running.
4. Read the answer and the tip. The tip is usually the exact point where interviewers try to trip you.
5. Do the two practice questions at the end of each section out loud, as if explaining to an interviewer.

INSIDE

01 SELECT and filters
8 queries

02 Sorting and LIMIT
8 queries

03 Aggregates, GROUP BY
8 queries

04 Joins
8 queries

05 Subqueries
7 queries

06 Window functions
7 queries

07 Classic problems
7 queries

08 Dates, NULLs, CASE
7 queries

BEFORE YOU START

```
Shared schema used by every query:
employees(id, name, dept_id, manager_id, salary, hire_date)
departments(id, name)
customers(id, name, city)
orders(id, customer_id, amount, order_date)

employees.dept_id points to departments.id, and employees.manager_id points to employees.id.
orders.customer_id points to customers.id. Dates are stored as text in YYYY-MM-DD form.
Queries are written for SQLite, and the answer notes the PostgreSQL or MySQL version wherever
it differs.
```

SECTION 01 / 08

SELECT and filters

Start here. Every harder query is built on these clauses, so get fast and accurate with them. Pay special attention to NULL and to the way AND and OR combine.

01 Select every row

Easy

Show every column and every row of the employees table.

SELECT says which columns you want and FROM says which table to read. The star means all columns, in the order the table was defined. There is no WHERE, so nothing is filtered and every row comes back. This is the quickest way to look at a table you have never seen, and interviewers often start here to check you know the basic shape of a query. In real code, avoid the star because a new column added later changes your output and can break the app that reads it. Name the columns you need instead.

```
SELECT * FROM employees;
```

Interview tip. SELECT * is fine for exploring, but say you would list columns in production code.

02 Columns and aliases

Easy

Show each employee's name and yearly salary, assuming the salary column holds a monthly amount.

The SELECT list can hold expressions, not just column names. Here salary * 12 is calculated for every row. Without a name, the output column would be called salary * 12, which is awkward to use, so AS gives it a clean alias. The alias exists only in the result, the table is not changed. Note that you usually cannot use this alias inside WHERE of the same query, because WHERE runs before SELECT. You can use it in ORDER BY, which runs after. Expressions can use +, -, *, / and functions like ROUND or UPPER.

```
SELECT name, salary * 12 AS annual_salary
FROM employees;
```

Interview tip. An alias cannot be used in WHERE of the same query, only in ORDER BY.

03 Filter with WHERE

Easy

List the names and salaries of employees who earn more than 80000.

WHERE keeps only the rows for which the condition is true and throws the rest away. The database checks the condition on each row of employees, in effect, and passes the survivors to SELECT. Comparison operators are =, <>, <, >, <= and >=. Note that SQL uses a single equals sign for comparison, not two. Text values go in single quotes, numbers do not. A row where the condition is false or unknown (for example when salary is NULL) is dropped. Here only Asha, Ben, Chitra and Jay pass the filter.

```
SELECT name, salary
FROM employees
WHERE salary > 80000;
```

Interview tip. Use single quotes for text and a single = for comparison.

04 AND, OR and brackets

Easy

Find employees in department 1 or 2 who earn more than 70000.

AND is evaluated before OR, just like multiplication before addition. If you write dept_id = 1 OR dept_id = 2 AND salary > 70000, SQL reads it as dept_id = 1 OR (dept_id = 2 AND salary > 70000), so every department 1 employee passes whatever their pay. That is a different question. Brackets force the order you actually mean: first pick the department, then apply the salary rule to both. When a WHERE clause mixes AND with OR, add brackets even where they are not strictly needed, because it makes the intent obvious to the next reader.

```
SELECT name, dept_id, salary
FROM employees
WHERE (dept_id = 1 OR dept_id = 2)
  AND salary > 70000;
```

Interview tip. Mixing AND with OR without brackets is the classic wrong-result bug.

05 IN and BETWEEN

Easy

List employees in departments 1 or 3 whose salary is between 60000 and 95000.

IN checks whether a value matches any item in a list, and it is a tidy replacement for a chain of OR conditions. BETWEEN checks a range and is inclusive on both ends, so a salary of exactly 60000 or exactly 95000 passes. It is shorthand for salary >= 60000 AND salary <= 95000. Both work on numbers, text and dates. NOT IN and NOT BETWEEN flip them. For dates stored with a time part, BETWEEN can miss rows late on the end day, so many people prefer >= start AND < next day.

```
SELECT name, dept_id, salary
FROM employees
WHERE dept_id IN (1, 3)
  AND salary BETWEEN 60000 AND 95000;
```

Interview tip. BETWEEN includes both endpoints, which surprises people with date ranges.

06 Pattern match with LIKE

Easy

Find employees whose name starts with the letter A.

LIKE matches text against a pattern. The percent sign stands for any number of characters, including none, and the underscore stands for exactly one character. So 'A%' means starts with A, '%a' means ends with a, '%an%' means contains an, and '_a%' means second letter is a. In SQLite and MySQL, LIKE ignores upper and lower case by default for plain letters, while PostgreSQL is case sensitive and offers ILIKE for the insensitive version. A pattern that starts with a percent sign cannot use a normal index, so it can be slow on big tables.

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

Interview tip. LIKE case rules differ by database. In PostgreSQL use ILIKE for case-insensitive.

07 Unique values with DISTINCT

Easy

List each city that has at least one customer, without repeats.

DISTINCT removes duplicate rows from the result. The customers table has three people in Mumbai and two in Delhi, but each city appears once in the output. DISTINCT looks at the whole row of selected columns together, so SELECT DISTINCT city, name would only remove rows where both city and name repeat. It is applied after the rows are selected, so it can be slower on large tables because the database must compare or sort the rows. If you only want to know how many different values exist, use COUNT(DISTINCT city) instead.

```
SELECT DISTINCT city
FROM customers;
```

Interview tip. DISTINCT applies to all selected columns together, not just the first one.

08 Find missing values

Easy

Find employees who have no manager.

NULL means unknown or missing, not zero and not an empty string. Any normal comparison with NULL, such as manager_id = NULL, gives unknown instead of true, so the row is dropped and you get nothing back. That is why SQL has the special tests IS NULL and IS NOT NULL. In the sample data only Asha has a NULL manager, which makes her the top of the reporting chain. This is a favourite warm-up question because many beginners write = NULL and wonder why the result is empty.

```
SELECT name
FROM employees
WHERE manager_id IS NULL;
```

Interview tip. Never write = NULL. Always use IS NULL or IS NOT NULL.

Test yourself

1. What is the difference between = NULL and IS NULL, and why does the first return nothing?
2. Rewrite dept_id = 1 OR dept_id = 2 using IN, then add a salary condition without breaking the logic.

SECTION 02 / 08

Sorting and LIMIT

Interviewers use these to check that you understand ordering, ties and pagination. The main trap is that LIMIT without ORDER BY gives arbitrary rows.

09 Sort high to low

Easy

List employees from the highest salary to the lowest.

ORDER BY sorts the final result. The default is ascending, smallest first, and DESC reverses it. Without ORDER BY the database may return rows in any order, and that order can change between runs, so never rely on it. ORDER BY runs near the end of the query, after WHERE and SELECT, which is why it can use column aliases. You can sort by a column that is not in the SELECT list. Ben and Chitra both earn 90000, so their relative order is undefined unless you add a second sort key.

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

Interview tip. Without ORDER BY, row order is not guaranteed, even if it looks stable.

10 Sort by two columns

Easy

Sort employees by department, and inside each department by salary from high to low.

ORDER BY accepts a list of keys. The database sorts by the first key, and only rows that tie on it are sorted by the second key, and so on. Each key has its own direction, so dept_id goes up while salary goes down. This is how you break ties and make output predictable. Adding a unique column like id as the last key gives a fully deterministic order, which matters when you paginate. Note that Hari has a NULL dept_id, and SQLite puts NULL first in ascending order, while PostgreSQL puts it last.

```
SELECT name, dept_id, salary
FROM employees
ORDER BY dept_id ASC, salary DESC;
```

Interview tip. Add a unique key last (like id) so ties never reshuffle between runs.

11 Top 3 earners

Easy

Show the three highest paid employees.

Sort first, then cut. ORDER BY salary DESC puts the biggest salary on top, and LIMIT 3 keeps only the first three rows of that sorted list. LIMIT must come last in the query. Without ORDER BY, LIMIT returns an arbitrary three rows, which is almost never what you want. SQL Server uses SELECT TOP 3 and Oracle 12c onwards uses FETCH FIRST 3 ROWS ONLY, but the idea is the same. Notice the tie: Ben and Chitra both earn 90000, so with LIMIT 3 only one of them is returned and which one is not guaranteed.

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

Interview tip. LIMIT cuts ties arbitrarily. Mention RANK or DENSE_RANK if ties must be kept.

12 Pagination with OFFSET

Easy

Return the second page of employees, 3 per page, ordered by id.

OFFSET skips a number of rows before LIMIT starts counting. With page size 3, page 1 is OFFSET 0, page 2 is OFFSET 3, page 3 is OFFSET 6, so the offset is (page - 1) * size. The ORDER BY on a unique column is essential, otherwise pages can overlap or miss rows. OFFSET gets slow on very large tables because the database still walks past all skipped rows. For big feeds, use keyset pagination: WHERE id > last_seen_id ORDER BY id LIMIT 3, which jumps straight to the right place using an index.

```
SELECT id, name
FROM employees
ORDER BY id
LIMIT 3 OFFSET 3;
```

Interview tip. Large OFFSET values are slow. Say keyset pagination for big tables.

13 Sort by an expression

Medium

Sort employees by the length of their name, shortest first, and alphabetically when lengths tie.

ORDER BY can sort by any expression, not only plain columns. Here LENGTH(name) is computed for every row and the rows are sorted on that value, then by name to break ties. You may also sort by the alias name_len, or by the position of the column in the SELECT list, such as ORDER BY 2, but position numbers break quietly if someone reorders the columns, so avoid them in real code. LENGTH is the name in SQLite, MySQL and PostgreSQL. SQL Server calls it LEN.

```
SELECT name, LENGTH(name) AS name_len
FROM employees
ORDER BY LENGTH(name), name;
```

Interview tip. Avoid ORDER BY 2 style positions. They silently break when columns change.

14 Latest hire

Easy

Find the most recently hired employee.

Sorting the hire dates from newest to oldest puts the latest one on top, and LIMIT 1 keeps only that row. Dates in the ISO format YYYY-MM-DD sort correctly as plain text, which is why SQLite can store them as text and still order them right. Dates in formats like DD-MM-YYYY would sort wrongly as text. If two people were hired on the same day, LIMIT 1 returns just one of them. To get all tied rows, use a subquery: WHERE hire_date = (SELECT MAX(hire_date) FROM employees).

```
SELECT name, hire_date
FROM employees
ORDER BY hire_date DESC
LIMIT 1;
```

Interview tip. Store dates as YYYY-MM-DD so text sorting matches calendar sorting.

15 Where do NULLs sort?

Medium

Sort employees by manager_id ascending but show the ones with no manager last.

Databases disagree on where NULL goes. SQLite and MySQL treat NULL as the smallest value, so it comes first in ascending order. PostgreSQL and Oracle treat it as the largest, so it comes last. To control it everywhere, sort by the test manager_id IS NULL first. That test gives 0 for rows with a value and 1 for NULL rows, so real values come first, then the NULLs, and the second key orders the real values. PostgreSQL and recent SQLite also support NULLS LAST directly.

```
SELECT name, manager_id
FROM employees
ORDER BY manager_id IS NULL, manager_id;
```

> **Interview tip.** NULL ordering is not portable. State which database you assume, or use the IS NULL trick.

16 Top 3 distinct salaries

Medium

List the three highest distinct salary values.

DISTINCT collapses repeated salaries first, so the two 90000 salaries count as one value. Then ORDER BY DESC sorts the unique values, and LIMIT 3 keeps the top three: 150000, 95000 and 90000. This is different from the top 3 earners query, which returns three people. When a question says top 3 salaries, ask whether it means people or distinct amounts. Logically, DISTINCT runs before ORDER BY and LIMIT, which is exactly the order you need here.

```
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
LIMIT 3;
```

> **Interview tip.** Clarify: top 3 people or top 3 distinct values? The queries differ.

Test yourself

1. Why can ORDER BY use a column alias but WHERE cannot?
2. How would you page through a million rows without a slow OFFSET?

SECTION 03 / 08

Aggregates, GROUP BY

Aggregates turn many rows into one number. Learn the rule for what may appear in SELECT, and the difference between WHERE and HAVING, and most questions here become easy.

17 COUNT(*) and COUNT(col)

Easy

Count all employees and count how many have a manager.

COUNT(*) counts rows, no matter what the columns hold. COUNT(column) counts only rows where that column is not NULL. With ten employees and one NULL manager, the first gives 10 and the second gives 9. This difference is one of the most asked basics in interviews. Every aggregate function except COUNT(*) ignores NULL values, which affects SUM, AVG, MIN and MAX too. An aggregate query without GROUP BY collapses the entire table into a single result row.

```
SELECT COUNT(*) AS all_rows,
       COUNT(manager_id) AS with_manager
FROM employees;
```

Interview tip. COUNT(*) counts rows. COUNT(col) skips NULLs. They often differ.

18 SUM, AVG, MIN and MAX

Easy

Show the total, average, lowest and highest salary across the company.

Aggregate functions read many rows and return one value. SUM adds, AVG takes the mean, MIN and MAX find the extremes. All of them skip NULL values, so AVG divides by the count of non-NULL rows, not the count of all rows. Integer division can bite: in some databases, dividing two integers drops the decimals, so use ROUND(AVG(salary), 2) or multiply by 1.0 when you compute ratios yourself. With no GROUP BY the whole table is one group, so you get exactly one output row.

```
SELECT SUM(salary) AS total,
       AVG(salary) AS average,
       MIN(salary) AS lowest,
       MAX(salary) AS highest
FROM employees;
```

Interview tip. AVG ignores NULLs. If NULL should count as zero, wrap it in COALESCE first.

19 Headcount per department

Easy

Count how many employees each department id has.

GROUP BY splits the rows into buckets, one per distinct dept_id, and the aggregate runs separately inside each bucket. The output has one row per group. The golden rule: every column in SELECT must either appear in GROUP BY or be inside an aggregate function. Here dept_id is grouped and COUNT(*) is aggregated, so it is valid. Notice that Hari's NULL dept_id forms its own group, because GROUP BY treats all NULLs as equal. Selecting name here would be an error in strict databases, because a group has many names.

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

Interview tip. Every non-aggregated SELECT column must be in GROUP BY.

20 Average salary per department

Easy

Show the average salary per department id, rounded to two decimals, highest first.

The query groups employees by department, computes the average inside each group, and rounds it. ORDER BY runs last, so it can use the alias avg_salary, and it sorts the grouped output, not the original rows. Mentally run the order: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY, LIMIT. Knowing this order explains most GROUP BY errors. ROUND takes the number of decimals as the second argument and exists in SQLite, MySQL and PostgreSQL, though PostgreSQL needs a numeric type, not a float, for two arguments.

```
SELECT dept_id, ROUND(AVG(salary), 2) AS avg_salary
FROM employees
GROUP BY dept_id
ORDER BY avg_salary DESC;
```

Interview tip. Learn the logical order: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY.

21 Filter groups with HAVING

Medium

Find departments that have more than two employees.

HAVING filters groups after GROUP BY, which is why it can use aggregates like COUNT(*). WHERE cannot do that, because WHERE runs before any groups exist. In the sample data department 1 has four people, department 2 has three, and department 3 has two, so departments 1 and 2 survive. Many databases do not allow an alias like headcount inside HAVING, so repeat the expression. Use HAVING only for conditions on aggregates, and keep ordinary row filters in WHERE, because filtering early means fewer rows to group.

```
SELECT dept_id, COUNT(*) AS headcount
FROM employees
GROUP BY dept_id
HAVING COUNT(*) > 2;
```

Interview tip. WHERE filters rows before grouping. HAVING filters groups after it.

22 WHERE plus HAVING together

Medium

For orders placed after 2024-01-05, find customers whose total is above 1000.

Both filters appear, each at its proper stage. WHERE first removes orders on or before 5 January, so those never reach the grouping step. Then GROUP BY builds one bucket per customer, SUM adds the remaining amounts, and HAVING keeps only buckets whose total exceeds 1000. Moving the date condition into HAVING would not even be valid, since order_date is not grouped or aggregated. Putting row-level conditions in WHERE is also faster, because fewer rows are grouped. Customers 4, 5 and 7 pass here.

```
SELECT customer_id, SUM(amount) AS total
FROM orders
WHERE order_date > '2024-01-05'
GROUP BY customer_id
HAVING SUM(amount) > 1000;
```

Interview tip. Row condition goes in WHERE, aggregate condition goes in HAVING.

23 COUNT DISTINCT

Medium

Count how many different customers have placed at least one order.

COUNT(*) on orders would count every order, and customers with repeat orders would be counted many times. COUNT(DISTINCT customer_id) counts each customer once. NULL values are skipped, so an order without a customer does not add to the count. The sample data has twelve orders from seven different customers, so the answer is 7. You can use DISTINCT inside SUM and AVG too, but COUNT is where it appears most. On very large tables COUNT(DISTINCT) is expensive, because the database has to track every unique value.

```
SELECT COUNT(DISTINCT customer_id) AS buyers
FROM orders;
```

Interview tip. COUNT(DISTINCT x) counts unique non-NULL values. Plain COUNT counts all rows.

24 Group by more than one

Medium

Count orders per customer per month.

Grouping by two expressions makes one bucket for each unique pair, so customer 1 gets a separate row for January and February. strftime with the format %Y-%m turns a date into a year-month label. That function is SQLite specific. In PostgreSQL use TO_CHAR(order_date, 'YYYY-MM') or DATE_TRUNC('month', order_date), and in MySQL use DATE_FORMAT(order_date, '%Y-%m'). The expression in GROUP BY must match the one in SELECT, or you can group by the alias in many databases.

```
SELECT customer_id,
       strftime('%Y-%m', order_date) AS month,
       COUNT(*) AS orders_count
FROM orders
GROUP BY customer_id, strftime('%Y-%m', order_date)
ORDER BY customer_id, month;
```

Interview tip. Date functions differ per database. Name the one you are using.

Test yourself

1. Why does COUNT(*) differ from COUNT(column) on the same table?
2. Write a query for departments whose average salary is above 70000.

SECTION 04 / 08

Joins

Joins are the most tested topic. Decide which table must keep all its rows, pick the join type from that, and always think about what happens to rows with no match.

25 Basic INNER JOIN

Easy

List each employee with the name of their department.

A join combines rows from two tables where the ON condition is true. Here every employee row is matched with the department row that has the same id. INNER JOIN keeps only matches, so Hari, whose dept_id is NULL, disappears from the output because NULL equals nothing. The letters e and d are table aliases, which keep the query short and avoid ambiguity, since both tables have a column called name. Always qualify such columns with the alias. INNER JOIN and plain JOIN mean the same thing.

```
SELECT e.name AS employee, d.name AS department
FROM employees e
INNER JOIN departments d ON d.id = e.dept_id;
```

> **Interview tip.** INNER JOIN drops unmatched rows, so employees with no department vanish.

26 LEFT JOIN keeps everyone

Easy

List every employee with their department name, showing NULL if they have none.

LEFT JOIN keeps every row of the left table, here employees, and fills the right side's columns with NULL when no match exists. Hari now appears, with a NULL department. This is the join to reach for whenever the question says all employees, all customers or even if there are none. RIGHT JOIN is the mirror image and is rarely needed, since you can swap the table order. Remember that the left table is the one written before the word JOIN.

```
SELECT e.name, d.name AS department
FROM employees e
LEFT JOIN departments d ON d.id = e.dept_id;
```

> **Interview tip.** Say all of X, even without Y, and pick LEFT JOIN from X.

27 Departments with no staff

Medium

Find departments that have no employees.

This is the anti-join pattern. LEFT JOIN keeps every department and attaches employees where they exist. For a department with no employees, all the employee columns come back NULL. Filtering WHERE e.id IS NULL keeps exactly those departments, and Legal is the answer. Test a column that can never be NULL in real data, like the primary key, so a genuine NULL value in another column does not trick you. The same result can be written with NOT EXISTS, which many engineers find easier to read and which is safe with NULLs.

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

Interview tip. Anti-join: LEFT JOIN then WHERE right_primary_key IS NULL.

28 Customers who never ordered

Medium

Find customers who have never placed an order.

Same anti-join idea on different tables. Every customer is kept by the LEFT JOIN, and customers with no orders get NULL in all order columns. Checking o.id IS NULL then keeps only the people who never ordered, which is Anita here. Putting a condition on the right table inside WHERE, such as o.amount > 100, would silently turn the LEFT JOIN into an INNER JOIN, because NULL rows fail the test. If you need such a filter, put it in the ON clause instead.

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

Interview tip. A WHERE filter on the right table cancels a LEFT JOIN. Put it in ON.

29 Total spent per customer

Medium

Show each customer's name and total amount spent, with 0 for customers who never ordered.

Join first, then group. The LEFT JOIN keeps customers without orders, whose SUM comes out as NULL because there is nothing to add. COALESCE swaps that NULL for 0. Grouping by c.id as well as c.name matters: the sample data has two customers named Rahul, and grouping by name alone would merge them into one. Group by the primary key plus any columns you display. Note also that SUM skips NULL amounts, so customer 6, whose only order has a NULL amount, shows 0.

```
SELECT c.name, COALESCE(SUM(o.amount), 0) AS total_spent
FROM customers c
LEFT JOIN orders o ON o.customer_id = c.id
GROUP BY c.id, c.name
ORDER BY total_spent DESC;
```

Interview tip. Group by the id, not just the name, or two people with one name merge.

30 Employee and manager names

Medium

List each employee with the name of their manager.

A self join joins a table to itself, so you must give it two different aliases. Think of e as the employee copy and m as the manager copy of the same table. The ON condition links an employee's manager_id to the manager's id. LEFT JOIN keeps Asha, whose manager_id is NULL, with a NULL manager. An INNER JOIN would drop her. Self joins are the standard way to model hierarchies stored in one table, such as employees and managers or categories and parent categories.

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

Interview tip. Use two aliases for the same table. LEFT JOIN keeps the top boss.

31 CROSS JOIN pairs

Medium

Count every possible pairing of an employee with a department.

A CROSS JOIN has no ON condition and returns the Cartesian product, every row of the first table paired with every row of the second. Ten employees times four departments gives forty rows. It is useful for generating combinations, such as every product with every size, or every date with every store before a LEFT JOIN fills in data. It is dangerous by accident: forgetting the join condition between two big tables creates a huge result that can freeze a database. If a join suddenly returns far too many rows, check the ON clause.

```
SELECT COUNT(*) AS pairs
FROM employees
CROSS JOIN departments;
```

Interview tip. A missing ON condition silently creates a CROSS JOIN with huge output.

32 Pairs in the same department

Hard

List each pair of employees who work in the same department, once per pair.

Joining a table to itself on dept_id pairs each person with everyone in their department, including themselves and the same pair in both orders. The extra condition a.id < b.id removes both problems at once: a person can never be paired with themselves, and each pair appears only in the order where the smaller id comes first. Without it, two people would produce four rows. This pattern solves questions about pairs, such as employees with the same salary, or customers from the same city. Hari is absent because NULL does not equal NULL in a join.

```
SELECT a.name AS first_person, b.name AS second_person, a.dept_id
FROM employees a
JOIN employees b
  ON a.dept_id = b.dept_id AND a.id < b.id
ORDER BY a.dept_id, a.id, b.id;
```

Interview tip. Use a.id < b.id to drop self pairs and mirrored duplicates.

Test yourself

1. When does a LEFT JOIN behave like an INNER JOIN by accident?
2. How do you find rows in table A that have no match in table B in three different ways?

SECTION 05 / 08

Subqueries

A subquery is a query inside another query. Practice scalar, list, EXISTS and correlated forms, and remember which ones are safe with NULL values.

33 Above the average

Easy

Find employees who earn more than the company average salary.

The bracketed query is a scalar subquery, meaning it returns exactly one value, here the average salary of 81,500. The database runs it once, then uses that number inside the outer WHERE. You cannot write WHERE salary > AVG(salary) directly, because aggregates are not allowed in WHERE. A scalar subquery must return one row and one column, and if it returns more rows the database raises an error. Here Asha, Ben, Chitra and Jay earn above the average, and Esha, at 75000, sits just below it.

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

Interview tip. Aggregates are not allowed in WHERE. Wrap them in a subquery.

34 Filter with IN subquery

Easy

List employees who work in the Sales department, without using a join.

The inner query produces a list of department ids whose name is Sales. The outer query keeps employees whose dept_id is in that list. This avoids hard-coding the id, so the query still works if the id changes. The same result comes from a join, and good optimizers often treat both the same way. Choose the form that reads clearly. Use IN when you only need columns from the outer table, and a join when you also need columns from the other table, like the department name in the output.

```
SELECT name
FROM employees
WHERE dept_id IN (
  SELECT id FROM departments WHERE name = 'Sales'
);
```

Interview tip. If you need columns from the second table, use a join, not IN.

35 The NOT IN trap

Hard

Find departments that have no employees, using NOT IN safely.

NOT IN has a nasty edge case. If the subquery returns even one NULL, then id NOT IN (1, 2, 3, NULL) evaluates to unknown for every row, and the query returns nothing. In the sample data Hari's dept_id is NULL, so the unprotected version silently returns zero rows. Adding WHERE dept_id IS NOT NULL inside the subquery removes the NULL and fixes it. A cleaner habit is to use NOT EXISTS, which handles NULLs correctly without a special filter. Legal is the correct answer here.

```
SELECT name
FROM departments
WHERE id NOT IN (
  SELECT dept_id FROM employees WHERE dept_id IS NOT NULL
);
```

Interview tip. NOT IN returns nothing if the list contains a NULL. Prefer NOT EXISTS.

36 EXISTS check

Medium

Find customers who have placed at least one order.

EXISTS asks one yes or no question: does the inner query return any row at all? The inner query refers to c.id from the outer query, which makes it correlated, so it is checked once per customer. What the inner query selects does not matter, SELECT 1 is a convention. The database can stop at the first matching order, so EXISTS is often faster than counting. Unlike a join, it never duplicates a customer who has several orders. Rahul with id 1 has four orders but appears once.

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

Interview tip. EXISTS never duplicates rows, unlike a join with many matches.

37 NOT EXISTS check

Medium

Find customers who have never ordered, using NOT EXISTS.

NOT EXISTS keeps the outer row only when the inner query finds nothing. For Anita, no order has her id, so the subquery is empty and she is returned. This version is the safest way to write an anti-join. It ignores NULL values in the inner table completely, unlike NOT IN, and it reads close to the plain English question. Most databases turn it into the same anti-join plan as the LEFT JOIN with IS NULL version. Interviewers like to ask you for all three forms and the trade-offs between them.

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

Interview tip. NOT EXISTS is NULL-safe. NOT IN is not.

38 Above own department average

Hard

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

This is a correlated subquery: the inner query uses e.dept_id from the current outer row, so it is evaluated per row, in effect. For each employee, it computes the average of that employee's own department and compares. The company average would be wrong here, since a Sales salary should be judged against Sales. Hari is not returned because his dept_id is NULL, so the inner query finds no rows and the comparison is unknown. On big tables the same result is faster with a join to a grouped subquery, or with AVG as a window function.

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

Interview tip. Correlated subquery runs per row. A window AVG or a grouped join is faster.

39 Subquery in FROM

Hard

What is the average number of employees per department, counting only employees who have a department?

You cannot nest aggregates like AVG(COUNT(*)) in a plain query, so you do it in two steps. The inner query builds a small table with one row per department and its headcount. The outer query then treats that result as a table, called a derived table, and averages the counts. Every derived table needs an alias, here per_dept, in most databases. A CTE written with WITH is the same idea in a more readable shape, and it is the style many teams prefer for multi-step logic.

```
SELECT AVG(cnt) AS avg_headcount
FROM (
  SELECT dept_id, COUNT(*) AS cnt
  FROM employees
  WHERE dept_id IS NOT NULL
  GROUP BY dept_id
) AS per_dept;
```

Interview tip. Aggregate of an aggregate needs a derived table or a CTE.

Test yourself

1. Why does NOT IN return no rows when the list contains a NULL?
2. Rewrite the above-department-average query using a join.

SECTION 06 / 08

Window functions

Window functions let you rank, compare neighbouring rows and compute running totals without collapsing rows. They appear in almost every mid and senior level SQL round.

40 ROW_NUMBER ranking

Medium

Number employees from the highest salary to the lowest, with unique numbers.

A window function calculates a value across a set of rows but, unlike GROUP BY, keeps every row. The OVER clause defines the window. Here ORDER BY salary DESC sorts the rows, and ROW_NUMBER hands out 1, 2, 3 and so on with no repeats and no gaps. Ben and Chitra tie at 90000, yet they get different numbers, and which one comes first is arbitrary unless you add a tie-breaker like id. ROW_NUMBER is the right tool when you need exactly one row per rank, such as picking one record per group.

```
SELECT name, salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS rn
FROM employees;
```

Interview tip. ROW_NUMBER never ties. Add a tie-breaker column for a stable result.

41 RANK with ties

Medium

Rank employees by salary so that equal salaries share a rank and the next rank skips.

RANK gives tied rows the same number and then skips ahead. Ben and Chitra both get 3, and the next person gets 5, not 4, because two rows already used positions 3 and 4. This matches how a sports leaderboard works. Use RANK when gaps are meaningful, for example the person ranked 5th really has four people above them. Compare with DENSE_RANK, which would not skip, and ROW_NUMBER, which breaks ties arbitrarily. Interviewers often ask you to state the difference between the three using a salary example exactly like this one.

```
SELECT name, salary,
       RANK() OVER (ORDER BY salary DESC) AS rnk
FROM employees;
```

Interview tip. RANK skips numbers after a tie. DENSE_RANK does not.

42 DENSE_RANK without gaps

Medium

Rank employees by salary with ties sharing a rank and no gaps in the numbers.

DENSE_RANK also gives ties the same number, but the next rank continues straight after, with no gap. Ben and Chitra share 3 and the next salary gets 4. That makes it perfect for questions about the nth highest value, because rank n then means the nth distinct salary. The same data under RANK would jump from 3 to 5. A quick way to remember: ROW_NUMBER is 1,2,3,4, RANK is 1,2,2,4, and DENSE_RANK is 1,2,2,3. All three need ORDER BY inside OVER to define what the ranking is based on.

```
SELECT name, salary,
       DENSE_RANK() OVER (ORDER BY salary DESC) AS drnk
FROM employees;
```

Interview tip. Nth highest distinct value means DENSE_RANK, not RANK.

43 Previous order with LAG

Hard

For each order, show the previous order amount of the same customer and the change.

LAG looks backwards and returns a value from an earlier row in the window. PARTITION BY customer_id restarts the window for every customer, so one customer's orders never leak into another's. ORDER BY decides what earlier means. For the first order of each customer there is nothing before it, so LAG returns NULL, and you can pass a default as the third argument, such as LAG(amount, 1, 0). LAG(amount, 2) looks two rows back. Subtracting gives the change between consecutive orders.

```
SELECT customer_id, order_date, amount,
       LAG(amount) OVER (PARTITION BY customer_id ORDER BY order_date, id) AS prev_amount,
       amount - LAG(amount) OVER (PARTITION BY customer_id ORDER BY order_date, id) AS change
FROM orders
ORDER BY customer_id, order_date, id;
```

Interview tip. The first row in each partition gets NULL from LAG. Set a default if needed.

44 Running total

Hard

Show a running total of order amounts in date order.

Using SUM as a window function with ORDER BY inside OVER creates a cumulative sum. For each row, the window runs from the first row up to the current one, so every line shows the total so far. Adding id to the order makes the sequence unique. Without that tie-breaker, two orders on the same day are, by default, treated as peers, and both get the same running total that includes each other. Add PARTITION BY customer_id to restart the total for each customer. NULL amounts are skipped by SUM, so they do not break the total.

```
SELECT id, order_date, amount,
       SUM(amount) OVER (ORDER BY order_date, id) AS running_total
FROM orders
ORDER BY order_date, id;
```

Interview tip. Ties in ORDER BY share a running total unless you add a unique tie-breaker.

45 Share of department total

Hard

Show each employee's salary as a percentage of their department's total salary.

SUM with OVER (PARTITION BY dept_id) and no ORDER BY adds up the whole department for every row, while still keeping every employee row. Dividing each salary by that total gives the share. The 100.0 matters: it forces decimal division, because in some databases an integer divided by an integer drops the fraction. Without a window function you would need a join to a grouped subquery to get the same effect. Partitioning is just GROUP BY for windows, with the difference that rows are not collapsed.

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

Interview tip. PARTITION BY is GROUP BY for windows. It keeps all rows.

46 Next order with LEAD

Hard

For each order, show how many days until the same customer's next order.

LEAD is the mirror of LAG and reads from a later row in the partition. The last order of each customer has no next row, so the value is NULL, and so is the gap. julianday converts a date into a day count, which makes subtraction give days. That function is SQLite specific. In PostgreSQL you can subtract two dates directly, and in MySQL you use DATEDIFF(next_date, order_date). Gap analysis like this appears in churn, retention and session-length questions.

```
SELECT customer_id, order_date,
       LEAD(order_date) OVER (PARTITION BY customer_id ORDER BY order_date, id) AS next_date,
       CAST(julianday(LEAD(order_date) OVER (PARTITION BY customer_id ORDER BY order_date,
id)) - julianday(order_date) AS INTEGER) AS days_gap
FROM orders
ORDER BY customer_id, order_date, id;
```

Interview tip. Date subtraction syntax differs per database. Say which one you assume.

Test yourself

1. What are the three outputs of ROW_NUMBER, RANK and DENSE_RANK for the values 10, 10, 8?
2. How do you restart a running total for each customer?

Classic problems

These are the questions that come up again and again. Learn the pattern behind each one, because interviewers change the table names but keep the idea.

47 Second highest salary

Medium

Find the second highest distinct salary.

The inner query finds the top salary, 150000. The outer query then takes the maximum of everything strictly below it, which is the second highest distinct value, 95000. Using strictly less than handles ties at the top correctly. If the table has fewer than two distinct salaries, MAX over no rows returns NULL instead of an error, which is a sensible answer. Other ways exist: ORDER BY salary DESC LIMIT 1 OFFSET 1 on distinct values, or DENSE_RANK equal to 2. Knowing two or three approaches impresses interviewers.

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

Interview tip. Use distinct values. With ties at the top, OFFSET 1 alone gives the wrong answer.

48 Nth highest salary

Hard

Find the third highest distinct salary using a window function.

DENSE_RANK numbers distinct salaries 1, 2, 3 without gaps, so rank 3 is exactly the third highest distinct salary, here 90000. It sits in a derived table because a window function result cannot be filtered in the same query level's WHERE. DISTINCT collapses tied rows, since two people share rank 3. Change 3 to any n for the general case. If fewer than n distinct salaries exist, the query returns no rows, so say how you want that handled, for example by wrapping in a scalar subquery to return NULL.

```
SELECT DISTINCT salary
FROM (
  SELECT salary,
         DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
  FROM employees
) t
WHERE rnk = 3;
```

Interview tip. Window results cannot be filtered in the same WHERE. Wrap in a subquery.

49 Find duplicate rows

Medium

Find customer names that appear more than once.

Group the rows by the column or columns that define a duplicate, then keep only groups whose count is above one. Here Rahul appears twice, once in Mumbai and once in Pune. To find duplicates across several columns, list them all in GROUP BY, for example name and city. This query tells you which values repeat but not which rows. To see the full duplicate rows, join the result back to the table on name. In an interview, also ask what counts as a duplicate, since different cities may make them separate customers.

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

Interview tip. Define duplicate first. Same name is not always the same person.

50 Delete duplicates, keep one

Hard

Delete duplicate customers by name, keeping the row with the lowest id.

The subquery groups customers by name and picks the smallest id in each group, so it returns one survivor id per name. The DELETE then removes every row whose id is not on that list, so only the extra copies go. Always run the equivalent SELECT first to preview what will be deleted, and wrap the delete in a transaction so you can roll back. MySQL does not allow deleting from a table you also select from in the same statement, so there you wrap the subquery in another derived table. Here Rahul with id 3 is removed.

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

Interview tip. Preview with SELECT first, and run deletes inside a transaction.

51 Top 2 per department

Hard

Find the two highest paid employees in each department.

This is the top N per group pattern. PARTITION BY dept_id restarts the numbering in every department, and ORDER BY salary DESC puts the best paid first, so rn 1 is the top earner and rn 2 the runner up. The outer query keeps rn up to 2. A plain LIMIT cannot do this, since it cuts the whole result, not each group. Choose the ranking function carefully: ROW_NUMBER returns exactly two rows per group, while RANK or DENSE_RANK can return more when salaries tie.

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

Interview tip. Ties decide the function: ROW_NUMBER for exactly N, DENSE_RANK to keep ties.

52 Earns more than manager

Medium

Find employees who earn more than their own manager.

This is a self join. The table appears twice, once as e for the employee and once as m for their manager, and the ON clause links manager_id to id. A plain JOIN is right because an employee with no manager cannot earn more than one, so Asha is naturally excluded. The WHERE clause then compares the two salaries on the same joined row. In the sample data Esha earns more than Dev, Gita more than Farid, and Jay more than Ben. This is one of the most common SQL interview questions, so know it well.

```
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;
```

Interview tip. Self join with two aliases, then compare the two copies of salary.

53 Three days in a row

Hard

Find customers who placed orders on at least three consecutive days.

This is the gaps and islands trick. Consecutive dates go up by one day, and ROW_NUMBER also goes up by one, so date minus row number stays constant inside a streak and changes when there is a break. Grouping by that constant value gives one group per streak. First remove duplicate dates, because two orders on one day would break the counting. Then HAVING keeps streaks of three or more days. Customer 1 ordered on 1, 2 and 3 January, so the streak runs three days.

```
WITH d AS (
  SELECT DISTINCT customer_id, order_date FROM orders
), g AS (
  SELECT customer_id, order_date,
         julianday(order_date) - ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY
order_date) AS grp
  FROM d
)
SELECT customer_id, MIN(order_date) AS start_date, MAX(order_date) AS end_date, COUNT(*) AS
days
FROM g
GROUP BY customer_id, grp
HAVING COUNT(*) >= 3;
```

Interview tip. Remove duplicate days first, or same-day orders break the streak count.

Test yourself

1. How would you get the 2nd highest salary per department?
2. How would you find customers who ordered on 3 consecutive days, ignoring duplicate days?

SECTION 08 / 08

Dates, NULLs, CASE

Real data is messy. Dates, missing values and conditional logic decide whether a report is correct, and dialect differences are a common follow-up question.

54 Group by year

Medium

Count how many employees were hired in each year.

strftime with the format %Y extracts the year from a date in SQLite, and GROUP BY then makes one bucket per year. SQLite lets you group by the alias, though not every database does. In PostgreSQL write EXTRACT(YEAR FROM hire_date), in MySQL YEAR(hire_date). Because the extracted year is calculated per row, a normal index on hire_date cannot help this grouping. For big tables, filtering by a date range is faster than filtering by a function of the date. Always say which database dialect you assume when you answer date questions.

```
SELECT strftime('%Y', hire_date) AS hire_year, COUNT(*) AS hired
FROM employees
GROUP BY hire_year
ORDER BY hire_year;
```

Interview tip. Date functions are the least portable part of SQL. Name your dialect.

55 Filter a date range

Medium

Find all orders placed in January 2024.

The safest range filter is a closed start and an open end, so you test greater than or equal to the first day and strictly less than the first day of the next month. It works whether the column holds plain dates or dates with a time of day, which BETWEEN '2024-01-01' AND '2024-01-31' would miss for anything after midnight on the 31st. It also lets the database use an index on the column, which a function like strftime on the column would prevent. Six of the twelve sample orders fall outside January, so the filter cuts them away.

```
SELECT id, customer_id, order_date
FROM orders
WHERE order_date >= '2024-01-01'
  AND order_date < '2024-02-01';
```

Interview tip. Use start inclusive and end exclusive. It is safe with times and indexes.

56 Days between two dates

Medium

Show each employee's tenure in days up to 2024-12-31.

julianday turns a date into a continuous day number, so subtracting two of them gives the difference in days. CAST to INTEGER drops the fractional part. We use a fixed end date so the result is repeatable. In real use, replace it with date('now') in SQLite, CURRENT_DATE in PostgreSQL, or CURDATE() in MySQL. The syntax for date arithmetic varies a lot: PostgreSQL subtracts dates directly, MySQL uses DATEDIFF, and SQL Server uses DATEDIFF with a unit argument. Interviewers mainly check that you know such a function exists and are careful about the dialect.

```
SELECT name, hire_date,
       CAST(julianday('2024-12-31') - julianday(hire_date) AS INTEGER) AS tenure_days
FROM employees
ORDER BY tenure_days DESC;
```

Interview tip. Use a fixed date while testing so results do not change daily.

57 Replace NULL with COALESCE

Easy

Show each order's amount, using 0 when the amount is missing.

COALESCE takes a list of values and returns the first one that is not NULL. With two arguments it works as a default value. Order 11 has a NULL amount, so it shows 0. It is available in SQLite, MySQL, PostgreSQL and SQL Server, while IFNULL, NVL and ISNULL are vendor-specific versions with only two arguments. Be careful about treating missing as zero: it is right for a total, but wrong for an average, where a missing amount should usually be excluded, not counted as a zero.

```
SELECT id, customer_id, COALESCE(amount, 0) AS amount_clean
FROM orders;
```

Interview tip. COALESCE is the portable one. IFNULL and NVL are vendor specific.

58 NULLs in aggregates

Medium

Show how the average order amount changes when NULL amounts are treated as zero.

Aggregates skip NULL. AVG(amount) divides the sum by the 11 rows that have an amount, while AVG(COALESCE(amount, 0)) divides by all 12 rows, so the second number is lower. COUNT(*) is 12 but COUNT(amount) is 11. Neither average is wrong, they answer different questions: the average of known orders, or the average counting unknown orders as zero revenue. In an interview, show that you notice this and ask which one the business wants. Silent NULLs are a leading cause of reports that do not add up.

```
SELECT COUNT(*) AS all_orders,
       COUNT(amount) AS with_amount,
       AVG(amount) AS avg_ignoring_null,
       AVG(COALESCE(amount, 0)) AS avg_null_as_zero
FROM orders;
```

Interview tip. Ask whether missing should be excluded or counted as zero.

59 Bands with CASE

Medium

Label each employee as High, Mid or Low based on salary.

CASE is SQL's if, else if and else. Conditions are tested from top to bottom and the first true one wins, so the order matters: 90000 matches the first branch and never reaches the second. ELSE catches everything that is left, and if you leave ELSE out, unmatched rows get NULL. All branches should return the same type. CASE can be used in SELECT, ORDER BY, GROUP BY and inside aggregates. Keep conditions from most specific to least specific, and use BETWEEN carefully so ranges do not overlap or leave gaps.

```
SELECT name, salary,
       CASE
         WHEN salary >= 90000 THEN 'High'
         WHEN salary >= 65000 THEN 'Mid'
         ELSE 'Low'
       END AS band
FROM employees
ORDER BY salary DESC;
```

Interview tip. Branches run top to bottom. Missing ELSE gives NULL, not an error.

60 Pivot with CASE and SUM

Hard

For each department, count employees earning at least 80000 and those earning less, in two columns.

This is conditional aggregation, the standard way to turn rows into columns. The CASE returns 1 for rows that fit the condition and 0 for the others, and SUM adds those ones, which counts matching rows within each group. One query produces several counts side by side, with no need for a pivot feature that not every database has. You can do the same with SUM(CASE WHEN ... THEN amount END) for money totals, or COUNT(CASE WHEN ... THEN 1 END), where the missing ELSE gives NULL and COUNT skips it.

```
SELECT dept_id,
       SUM(CASE WHEN salary >= 80000 THEN 1 ELSE 0 END) AS high_earners,
       SUM(CASE WHEN salary < 80000 THEN 1 ELSE 0 END) AS other_earners
FROM employees
GROUP BY dept_id
ORDER BY dept_id;
```

Interview tip. COUNT(CASE WHEN ... THEN 1 END) works too, since COUNT skips NULL.

Test yourself

1. Why is a function on a date column in WHERE bad for index use?
2. How do you count rows by category in separate columns using one query?