20 SQL Interview Questions for Data Analysts

Pivotalstats · youtube.com/@pivotalstats · six patterns, solved in SQL Server on the free InterviewLab database

Q1 · Pattern 1 · Filter, aggregate, HAVING

Revenue + orders by country

"Revenue and number of orders by country, highest revenue first."

The pattern: GROUP BY with COUNT(DISTINCT)

```
SELECT c.country,
    SUM(ol.revenue) AS revenue,
    COUNT(DISTINCT o.order_id) AS orders
FROM orders AS o
JOIN customers AS c ON c.customer_id = o.customer_id
JOIN order_lines AS ol ON ol.order_id = o.order_id
GROUP BY c.country
ORDER BY revenue DESC;
```

Result (7 rows):

Ireland	58386.91	614
United Kingdom	40060.79	486
France	35419.09	372
United States	31974.41	336
Germany	29219.56	342
Spain	24550.53	261
Canada	15408.76	192

The trap: COUNT(*) here counts order lines, not orders. Ireland: orders: trap gives 995, right answer 614.

Next they'll ask: Why not join customers last? Order of joins doesn't change an inner-join result.

Q2 · Pattern 1 · Filter, aggregate, HAVING

Average order value

"What's the average order value in each country?"

The pattern: order-level totals, then the average

```
SELECT c.country,
    CAST(SUM(ol.revenue) / COUNT(DISTINCT o.order_id) AS DECIMAL(10,2))
        AS avg_order_value
FROM orders AS o
JOIN customers AS c ON c.customer_id = o.customer_id
JOIN order_lines AS ol ON ol.order_id = o.order_id
GROUP BY c.country
ORDER BY avg_order_value DESC;
```

Result (7 rows):

France	95.21
United States	95.16
Ireland	95.09
Spain	94.06
Germany	85.44
United Kingdom	82.43
Canada	80.25

The trap: AVG(revenue) averages order lines, not orders. France: average order: trap gives $59.13, right answer $95.21.

Next they'll ask: What's the median instead? PERCENTILE_CONT over order totals.

Q3 · Pattern 1 · Filter, aggregate, HAVING

Bought Bags AND Mugs

"Which customers bought both Bags and Mugs?"

The pattern: HAVING COUNT(DISTINCT)

```
SELECT c.customer_name
FROM customers AS c
JOIN orders AS o ON o.customer_id = c.customer_id
JOIN order_lines AS ol ON ol.order_id = o.order_id
JOIN products AS p ON p.product_id = ol.product_id
WHERE p.category IN ('Bags', 'Mugs')
GROUP BY c.customer_id, c.customer_name
HAVING COUNT(DISTINCT p.category) = 2
ORDER BY c.customer_name;
```

Result (250 rows):

Ana A. Byrne
Ana A. Chen
Ana A. Doyle
Ana A. Garcia
Ana A. Kelly
Ana A. Novak
Ana A. Patel
Ana B. Byrne

... 242 more rows

The trap: IN without HAVING returns anyone who bought either. customers returned: trap gives 297 rows, right answer 250 rows.

Next they'll ask: Bags but never Mugs? Add a NOT EXISTS for Mugs.

Q4 · Pattern 2 · Joins and anti-joins

Signed up, never ordered

"Which customers signed up but never placed an order?"

The pattern: anti-join with NOT EXISTS

```
SELECT c.customer_id, c.customer_name, c.signup_date
FROM customers AS c
WHERE NOT EXISTS (
    SELECT 1 FROM orders AS o
    WHERE o.customer_id = c.customer_id
)
ORDER BY c.signup_date, c.customer_id;
```

Result (20 rows):

C312	Theo Nolan	2025-01-04
C316	Rory Dunne	2025-01-25
C314	Jude Farrell	2025-02-04
C313	Lily Boyle	2025-02-22
C303	Isla Moran	2025-02-23
C305	Freya Cullen	2025-03-16
C311	Erin Duffy	2025-03-26
C310	Adam Keane	2025-04-12

... 12 more rows

The trap: An INNER JOIN can never find them: 0 rows. customers found: trap gives 0 rows, right answer 20 rows.

Next they'll ask: LEFT JOIN ... WHERE o.order_id IS NULL gives the same 20 rows.

Q5 · Pattern 2 · Joins and anti-joins

Revenue + returns together

"Revenue and returned units per category, in one result."

The pattern: aggregate before you join

```
WITH sales AS (
    SELECT p.category, SUM(ol.revenue) AS revenue
    FROM order_lines AS ol
    JOIN products AS p ON p.product_id = ol.product_id
    GROUP BY p.category
), rets AS (
    SELECT p.category, SUM(r.quantity) AS returned_units
    FROM returns AS r
    JOIN products AS p ON p.product_id = r.product_id
    GROUP BY p.category
)
SELECT s.category, s.revenue,
    COALESCE(r.returned_units, 0) AS returned_units
FROM sales AS s
LEFT JOIN rets AS r ON r.category = s.category
ORDER BY s.revenue DESC;
```

Result (5 rows):

Bags	126061.62	54
Prints	47450.47	66
Desk Accessories	35525.95	48
Mugs	16015.83	62
Notebooks	9966.18	44

The trap: Join returns on order_id alone and revenue inflates. total revenue: trap gives $235,546.60, right answer $235,020.05.

Next they'll ask: Return rate? Returned units divided by units sold, both pre-aggregated.

Q6 · Pattern 2 · Joins and anti-joins

Employee + manager

“List every employee with their manager's name.”

The pattern: self-join, LEFT vs INNER

```
SELECT e.full_name AS employee, e.job_title, m.full_name AS manager
FROM employees AS e
LEFT JOIN employees AS m ON m.employee_id = e.manager_id
ORDER BY e.employee_id;
```

Result (18 rows):

Aoife Byrne	Co-founder & CEO	NULL
Daniel Okafor	Co-founder & COO	NULL
Priya Nair	Head of Marketing	Aoife Byrne
Tom Walsh	Head of Operations	Daniel Okafor
Grace Kim	Head of Customer Care	Daniel Okafor
Hannah Lee	Senior Data Analyst	Daniel Okafor
Marco Silva	Senior Data Analyst	Daniel Okafor
Liam Murphy	Marketing Manager	Priya Nair

... 10 more rows

The trap: INNER JOIN silently drops the two co-founders. employees listed: trap gives 16 rows, right answer 18 rows.

Next they'll ask: Show 'none' instead of NULL? COALESCE(m.full_name, 'none').

Q7 · Pattern 2 · Joins and anti-joins

Who isn't a manager?

“Which employees don't manage anyone?”

The pattern: NOT IN vs NOT EXISTS with NULLs

```
SELECT e.full_name, e.job_title
FROM employees AS e
WHERE NOT EXISTS (
    SELECT 1 FROM employees AS r
    WHERE r.manager_id = e.employee_id
)
ORDER BY e.full_name;
```

Result (10 rows):

Ben Carter	Customer Care Advisor
Chloe Martin	Customer Care Advisor
Jack Doyle	Picker Packer
Leah Fischer	Data Analyst
Marco Silva	Senior Data Analyst
Nina Kowalski	Picker Packer
Owen Hughes	Junior Data Analyst
Ravi Patel	Content Writer

... 2 more rows

The trap: One NULL in the NOT IN list returns 0 rows. employees found: trap gives 0 rows, right answer 10 rows.

Next they'll ask: Keep NOT IN? Filter the subquery: WHERE manager_id IS NOT NULL.

Q8 · Pattern 3 · Top N per group

Top 3 per category

"Top 3 products by revenue in each category."

The pattern: ROW_NUMBER per group

```
WITH product_revenue AS (
    SELECT p.category, p.product_name, SUM(ol.revenue) AS revenue,
        ROW_NUMBER() OVER (PARTITION BY p.category
                           ORDER BY SUM(ol.revenue) DESC) AS rn
    FROM order_lines AS ol
    JOIN products AS p ON p.product_id = ol.product_id
    GROUP BY p.category, p.product_name
)
SELECT category, product_name, revenue, rn
FROM product_revenue
WHERE rn <= 3
ORDER BY category, rn;
```

Result (15 rows):

Bags	Classic Tote	25109.85	1
Bags	Grande Tote	20734.96	2
Bags	Retro Tote	18256.69	3
Desk Accessories	Mini Desk Tidy	6310.0	1
Desk Accessories	Nordic Desk Tidy	6169.48	2
Desk Accessories	Grande Desk Tidy	5425.92	3
Mugs	Retro Mug	2843.55	1
Mugs	Grande Mug	2459.16	2

... 7 more rows

The trap: TOP 3 gives 3 rows in total, not 3 per category. rows returned: trap gives 3 rows, right answer 15 rows.

Next they'll ask: Ties at third place? Use RANK or DENSE_RANK and say which you chose.

Q9 · Pattern 3 · Top N per group

Second-highest salary

"What's the second-highest salary?"

The pattern: DENSE_RANK

```
WITH ranked AS (
    SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS salary_rank
    FROM employees
)
SELECT DISTINCT salary
FROM ranked
WHERE salary_rank = 2;
```

Result (1 row):

85000

The trap: Two people share the top salary, so OFFSET 1 returns it again. second-highest salary: trap gives 120,000, right answer 85,000.

Next they'll ask: Nth highest? Change the 2. No second salary? The query returns no rows.

Q10 · Pattern 3 · Top N per group

Top earner per department

"Highest-paid person in each department. Ties count."

The pattern: RANK vs ROW_NUMBER

```
WITH ranked AS (
    SELECT department, full_name, salary,
        RANK() OVER (PARTITION BY department
                     ORDER BY salary DESC) AS salary_rank
    FROM employees
)
SELECT department, full_name, salary
FROM ranked
WHERE salary_rank = 1
ORDER BY department, full_name;
```

Result (7 rows):

Customer Care	Grace Kim	68000
Data	Hannah Lee	72000
Data	Marco Silva	72000
Leadership	Aoife Byrne	120000
Leadership	Daniel Okafor	120000
Marketing	Priya Nair	85000
Operations	Tom Walsh	82000

The trap: ROW_NUMBER drops one of the tied people. people returned: trap gives 5 rows, right answer 7 rows.

Next they'll ask: RANK vs DENSE_RANK? They differ only after a tie: 1,1,3 vs 1,1,2.

Q11 · Pattern 3 · Top N per group

Each customer's first order

"For each customer, their first order's date and value."

The pattern: ROW_NUMBER, pick row 1

```
WITH order_totals AS (
    SELECT o.order_id, o.customer_id, o.order_date,
        SUM(ol.revenue) AS order_value
    FROM orders AS o
    JOIN order_lines AS ol ON ol.order_id = o.order_id
    GROUP BY o.order_id, o.customer_id, o.order_date
), ranked AS (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY customer_id
                                 ORDER BY order_date, order_id) AS rn
    FROM order_totals
)
SELECT customer_id, order_date AS first_order_date,
    order_value AS first_order_value
FROM ranked
WHERE rn = 1
ORDER BY customer_id;
```

Result (300 rows):

C001	2024-03-24	12.42
C002	2024-03-12	12.06
C003	2024-03-16	148.86
C004	2024-02-10	12.42
C005	2024-02-15	9.65
C006	2024-09-19	75.33
C007	2024-01-28	112.21
C008	2024-01-03	209.01

... 292 more rows

The trap: MIN(date) and MIN(value) can come from different orders. Liam Byrne: first order value: trap gives $26.36, right answer $148.86.

Next they'll ask: Latest order instead? ORDER BY order_date DESC in the window.

Q12 · Pattern 4 · LAG and LEAD

Month-over-month change

"Month-over-month revenue change, in money and percent."

The pattern: LAG over a monthly CTE

```
WITH monthly AS (
    SELECT DATEFROMPARTS(YEAR(o.order_date), MONTH(o.order_date), 1)
            AS month_start,
        SUM(ol.revenue) AS revenue
    FROM orders AS o
    JOIN order_lines AS ol ON ol.order_id = o.order_id
    GROUP BY DATEFROMPARTS(YEAR(o.order_date), MONTH(o.order_date), 1)
)
SELECT month_start, revenue,
    revenue - LAG(revenue) OVER (ORDER BY month_start) AS change,
    CAST(100.0 * (revenue - LAG(revenue) OVER (ORDER BY month_start))
        / NULLIF(LAG(revenue) OVER (ORDER BY month_start), 0)
        AS DECIMAL(6,1)) AS change_pct
FROM monthly
ORDER BY month_start;
```

Result (24 rows):

2024-01-01	7116.39	NULL	NULL
2024-02-01	7234.07	117.68	1.7
2024-03-01	9492.76	2258.69	31.2
2024-04-01	7031.89	-2460.87	-25.9
2024-05-01	9395.11	2363.22	33.6
2024-06-01	9669.28	274.17	2.9
2024-07-01	10251.59	582.31	6.0
2024-08-01	8367.17	-1884.42	-18.4

... 16 more rows

The trap: ISNULL(LAG(...), 0) turns the first month into fake growth. Jan 2024 change: trap gives $7,116.39, right answer NULL (blank).

Next they'll ask: Same month last year? LAG(revenue, 12).

Q13 · Pattern 4 · LAG and LEAD

Days between orders

"On average, how many days pass between a customer's orders?"

The pattern: LAG per customer + DATEDIFF

```
WITH gaps AS (
    SELECT customer_id,
        DATEDIFF(day,
            LAG(order_date) OVER (PARTITION BY customer_id
                                  ORDER BY order_date, order_id),
            order_date) AS days_since_prev
    FROM orders
)
SELECT CAST(AVG(CAST(days_since_prev AS DECIMAL(10,2))) AS DECIMAL(10,1))
    AS avg_days_between_orders
FROM gaps;
```

Result (1 row):

73.6

The trap: AVG of an integer returns an integer in SQL Server. average days between orders: trap gives 73 days, right answer 73.6 days.

Next they'll ask: Customers with one order? Their gap is NULL, and AVG skips it.

Q14 · Pattern 4 · LAG and LEAD

Back after 180+ days

"Which customers came back after 180 or more days away?"

The pattern: LAG, then filter in an outer query

```
WITH gaps AS (
    SELECT customer_id, order_date,
        LAG(order_date) OVER (PARTITION BY customer_id
                              ORDER BY order_date, order_id) AS prev_order_date
    FROM orders
)
SELECT c.customer_name, g.prev_order_date, g.order_date AS returned_on,
    DATEDIFF(day, g.prev_order_date, g.order_date) AS days_away
FROM gaps AS g
JOIN customers AS c ON c.customer_id = g.customer_id
WHERE DATEDIFF(day, g.prev_order_date, g.order_date) >= 180
ORDER BY days_away DESC, c.customer_name, returned_on;
```

Result (203 rows):

Ana B. Walsh	2024-03-19	2025-09-07	537
Ana A. Patel	2024-06-15	2025-10-17	489
Emma Doyle	2024-09-14	2025-12-17	459
Ana Silva	2024-06-30	2025-09-07	434
Noah Novak	2024-07-13	2025-09-18	432
Tom Garcia	2024-10-27	2025-12-27	426
Emma A. Chen	2024-10-21	2025-12-11	416
Sofia Byrne	2024-06-26	2025-08-01	401

... 195 more rows

The trap: A window function in WHERE is an error. comebacks of 180+ days: trap gives Error: window function in WHERE, right answer 203 rows.

Next they'll ask: Why the CTE? WHERE runs before window functions are calculated.

Q15 · Pattern 5 · Running totals and moving averages

Running total by year

"A daily running revenue total that restarts every year."

The pattern: SUM() OVER with an explicit frame

```
WITH daily AS (
    SELECT o.order_date, SUM(ol.revenue) AS revenue
    FROM orders AS o
    JOIN order_lines AS ol ON ol.order_id = o.order_id
    GROUP BY o.order_date
)
SELECT order_date, revenue,
    SUM(revenue) OVER (PARTITION BY YEAR(order_date)
                       ORDER BY order_date
                       ROWS UNBOUNDED PRECEDING) AS running_revenue
FROM daily
ORDER BY order_date;
```

Result (701 rows):

2024-01-02	341.08	341.08
2024-01-03	652.47	993.55
2024-01-04	114.96	1108.51
2024-01-05	435.43	1543.94
2024-01-06	304.83	1848.77
2024-01-07	542.15	2390.92
2024-01-08	140.09	2531.01
2024-01-09	274.78	2805.79

... 693 more rows

The trap: The default RANGE frame gives every line on a date the same total. 02 Jan 2024: running total: trap gives all 2 lines show $341.08, right answer one row per day, growing.

Next they'll ask: Running total per customer? Add customer_id to PARTITION BY.

Q16 · Pattern 5 · Running totals and moving averages

3-month moving average

"A 3-month moving average, blank until 3 months exist."

The pattern: ROWS BETWEEN 2 PRECEDING

```
WITH monthly AS (
    SELECT DATEFROMPARTS(YEAR(o.order_date), MONTH(o.order_date), 1)
            AS month_start,
        SUM(ol.revenue) AS revenue
    FROM orders AS o
    JOIN order_lines AS ol ON ol.order_id = o.order_id
    GROUP BY DATEFROMPARTS(YEAR(o.order_date), MONTH(o.order_date), 1)
)
SELECT month_start, revenue,
    CASE WHEN COUNT(*) OVER (ORDER BY month_start
                             ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) = 3
        THEN CAST(AVG(revenue) OVER (ORDER BY month_start
                  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS DECIMAL(10,2))
    END AS moving_avg_3m
FROM monthly
ORDER BY month_start;
```

Result (24 rows):

2024-01-01	7116.39	NULL
2024-02-01	7234.07	NULL
2024-03-01	9492.76	7947.74
2024-04-01	7031.89	7919.57
2024-05-01	9395.11	8639.92
2024-06-01	9669.28	8698.76
2024-07-01	10251.59	9771.99
2024-08-01	8367.17	9429.35

... 16 more rows

The trap: Without a guard, the first two months average a partial window. Feb 2024 moving average: trap gives $7,175.23, right answer NULL (blank).

Next they'll ask: Centred average? ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING.

Q17 · Pattern 5 · Running totals and moving averages

Share of total

"Each category's share of total revenue."

The pattern: SUM() OVER ()

```
SELECT p.category, SUM(ol.revenue) AS revenue,
    CAST(100.0 * SUM(ol.revenue) / SUM(SUM(ol.revenue)) OVER ()
        AS DECIMAL(5,1)) AS pct_of_total
FROM order_lines AS ol
JOIN products AS p ON p.product_id = ol.product_id
GROUP BY p.category
ORDER BY revenue DESC;
```

Result (5 rows):

Bags	126061.62	53.6
Prints	47450.47	20.2
Desk Accessories	35525.95	15.1
Mugs	16015.83	6.8
Notebooks	9966.18	4.2

The trap: SUM / SUM inside the same GROUP BY is 100% on every row. Bags: share: trap gives 100.0%, right answer 53.6%.

Next they'll ask: Share within each country? SUM(SUM(...)) OVER (PARTITION BY country).

Q18 · Pattern 6 · CTEs for multi-step logic

Remove duplicates

"This import has duplicate rows. Remove them, keep one of each."

The pattern: ROW_NUMBER + DELETE in a CTE

```
SELECT COUNT(*) AS rows_before FROM orders_raw_import;
WITH ranked AS (
    SELECT *, ROW_NUMBER() OVER (
        PARTITION BY order_id, order_date, customer_id, amount, source_file
        ORDER BY (SELECT NULL)) AS rn
    FROM orders_raw_import
)
DELETE FROM ranked WHERE rn > 1;
SELECT COUNT(*) AS rows_after FROM orders_raw_import;
```

Result (1 row):

52

The trap: DISTINCT hides duplicates in the result; the table keeps them. rows in the table: trap gives 60 rows, right answer 52 rows.

Next they'll ask: Keep the newest copy? ORDER BY a load timestamp DESC in the window.

Q19 · Pattern 6 · CTEs for multi-step logic

Cohort retention

"Of customers who first ordered in each 2024 month, what % came back within 3 months?"

The pattern: multi-step CTEs

```
WITH firsts AS (
    SELECT customer_id, MIN(order_date) AS first_date
    FROM orders
    GROUP BY customer_id
), came_back AS (
    SELECT DISTINCT f.customer_id
    FROM firsts AS f
    JOIN orders AS o ON o.customer_id = f.customer_id
        AND o.order_date > f.first_date
        AND o.order_date <= DATEADD(month, 3, f.first_date)
)
SELECT DATEFROMPARTS(YEAR(f.first_date), MONTH(f.first_date), 1)
        AS cohort_month,
    COUNT(*) AS customers,
    COUNT(cb.customer_id) AS came_back,
    CAST(100.0 * COUNT(cb.customer_id) / COUNT(*) AS DECIMAL(5,1))
        AS retention_pct
FROM firsts AS f
LEFT JOIN came_back AS cb ON cb.customer_id = f.customer_id
WHERE f.first_date < '2025-01-01'
GROUP BY DATEFROMPARTS(YEAR(f.first_date), MONTH(f.first_date), 1)
ORDER BY cohort_month;
```

Result (11 rows):

2024-01-01	64	38	59.4
2024-02-01	65	48	73.8
2024-03-01	48	31	64.6
2024-04-01	25	15	60.0
2024-05-01	29	20	69.0
2024-06-01	19	14	73.7
2024-07-01	17	12	70.6
2024-08-01	7	5	71.4

... 3 more rows

The trap: Counting the first order itself makes retention 100%. Jan 2024 cohort retention: trap gives 100.0%, right answer 59.4%.

Next they'll ask: Within 6 months? Change the DATEADD. Same-day repeat orders? They don't count here.

Q20 · Pattern 6 · CTEs for multi-step logic

Org chart levels

"Show the org chart: each employee's level below the top."

The pattern: recursive CTE

```
WITH org AS (
    SELECT employee_id, full_name, job_title, 0 AS lvl
    FROM employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.employee_id, e.full_name, e.job_title, org.lvl + 1
    FROM employees AS e
    JOIN org ON e.manager_id = org.employee_id
)
SELECT lvl AS level, full_name, job_title
FROM org
ORDER BY lvl, full_name;
```

Result (18 rows):

0	Aoife Byrne	Co-founder & CEO
0	Daniel Okafor	Co-founder & COO
1	Grace Kim	Head of Customer Care
1	Hannah Lee	Senior Data Analyst
1	Marco Silva	Senior Data Analyst
1	Priya Nair	Head of Marketing
1	Tom Walsh	Head of Operations
2	Ben Carter	Customer Care Advisor

... 10 more rows

The trap: Drop WHERE manager_id IS NULL and 18 people become many more rows. rows for 18 people: trap gives 49 rows, right answer 18 rows.

Next they'll ask: Show the reporting path? Build a string column: org.path + ' > ' + e.full_name.

Running these in PostgreSQL or MySQL

SQL Server (this video)	PostgreSQL	MySQL 8
TOP / OFFSET ... FETCH	LIMIT n / LIMIT n OFFSET m	LIMIT n / LIMIT m, n
DATEDIFF(day, a, b)	b - a (dates) or DATE_PART('day', b - a)	DATEDIFF(b, a)
DATEADD(month, 3, d)	d + INTERVAL '3 months'	DATE_ADD(d, INTERVAL 3 MONTH)
DATEFROMPARTS(y, m, 1)	make_date(y, m, 1) or date_trunc('month', d)	DATE_FORMAT(d, '%Y-%m-01')
ISNULL(x, 0)	COALESCE(x, 0)	IFNULL(x, 0) or COALESCE
WITH org AS (... UNION ALL ...)	WITH RECURSIVE org AS (...)	WITH RECURSIVE org AS (...)
DELETE FROM a CTE (Q18)	DELETE ... USING / WHERE ctid IN (...)	DELETE t FROM t JOIN (...)
AVG(int) truncates (Q13)	AVG(int) returns numeric (no truncation)	AVG(int) returns decimal

Window functions, CTEs, NOT EXISTS, RANK, DENSE_RANK, ROW_NUMBER, LAG and LEAD work the same in all three.