Interview and Real-World SQL Queries

Interview Question Types

1. Find the Nth Highest/Lowest

Problem: "Find the 3rd highest salary."

Trap: Using OFFSET without handling ties.

Bad:

```sql
SELECT salary FROM employees ORDER BY salary DESC LIMIT 1 OFFSET 2;
```

Problem: If two people have the 2nd highest salary, this returns the wrong person.

Good (window function):

```sql
SELECT DISTINCT salary
FROM (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rank
  FROM employees
) ranked
WHERE rank = 3;
```

Why DENSE_RANK: Handles ties correctly. If two people earn $100k (2nd highest), the next salary is rank 3.

Alternative (subquery):

```sql
SELECT MAX(salary)
FROM employees
WHERE salary < (
  SELECT MAX(salary) FROM employees
  WHERE salary < (SELECT MAX(salary) FROM employees)
);
```

Ugly, nested, hard to extend to Nth.

2. Find Duplicates

Problem: "Find employees with duplicate emails."

Solution:

```sql
SELECT email, COUNT(*)
FROM employees
GROUP BY email
HAVING COUNT(*) > 1;
```

Include the names:

```sql
SELECT e.*
FROM employees e
JOIN (
  SELECT email FROM employees GROUP BY email HAVING COUNT(*) > 1
) dupes ON e.email = dupes.email;
```

Or with window function:

```sql
SELECT * FROM (
  SELECT *, COUNT(*) OVER (PARTITION BY email) AS cnt
  FROM employees
) sub
WHERE cnt > 1;
```

Comparison:

- **GROUP BY + JOIN:** Classic, readable
- **Window function:** More concise, single pass

3. Running Totals

Problem: "Show each order and the running total of sales."

Solution (window function):

```sql
SELECT
  order_id,
  order_date,
  total,
  SUM(total) OVER (ORDER BY order_date) AS running_total
FROM orders;
```

Without window functions (ugly):

```sql
SELECT
  o1.order_id,
  o1.order_date,
  o1.total,
  (SELECT SUM(total) FROM orders o2 WHERE o2.order_date <= o1.order_date) AS
running_total
FROM orders o1;
```

Problem: Correlated subquery, runs once per row ($O(n^2)$).

Lesson: Window functions are essential.

4. Top N per Group

Problem: "Top 3 products by sales in each category."

Solution:

```sql
SELECT * FROM (
  SELECT
    category_id,
    product_id,
    sales,
    ROW_NUMBER() OVER (PARTITION BY category_id ORDER BY sales DESC) AS rank
  FROM product_sales
) ranked
WHERE rank <= 3;
```

Without window functions:

```sql
SELECT ps1.*
FROM product_sales ps1
WHERE (
  SELECT COUNT(*)
  FROM product_sales ps2
  WHERE ps2.category_id = ps1.category_id AND ps2.sales >= ps1.sales
) <= 3;
```

Ugly, slow, hard to read.

Lesson: `ROW_NUMBER()` or `RANK()` with `PARTITION BY`.

5. Gaps and Islands

Problem: "Find consecutive date ranges."

Example: User logged in on 2024-01-01, 01-02, 01-03, then skipped 01-04, then logged in 01-05, 01-06.

Goal: Group into:

- Island 1: 2024-01-01 to 2024-01-03
- Island 2: 2024-01-05 to 2024-01-06

Solution:

```sql
WITH numbered AS (
  SELECT
    user_id,
    login_date,
    login_date - (ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY login_date))::int
AS grp
  FROM logins
)
SELECT
  user_id,
  MIN(login_date) AS start_date,
  MAX(login_date) AS end_date,
  COUNT(*) AS consecutive_days
FROM numbered
GROUP BY user_id, grp;
```

How it works:

- `ROW_NUMBER()` generates 1, 2, 3, ...
- Subtract from the date: consecutive dates produce the same value
- Group by that value

Example:

```
login_date   ROW_NUMBER   login_date - ROW_NUMBER
2024-01-01   1            2023-12-31  ← same group
2024-01-02   2            2023-12-31  ← same group
2024-01-03   3            2023-12-31  ← same group
2024-01-05   4            2024-01-01  ← different group
2024-01-06   5            2024-01-01  ← same group
```

Lesson: Gaps and islands require clever math with window functions.

6. Self-Joins (Hierarchies)

Problem: "Find all employees and their managers."

Solution:

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

Find employees who earn more than their manager:

```
SELECT e.name
FROM employees e
JOIN employees m ON e.manager_id = m.id
WHERE e.salary > m.salary;
```

Find all subordinates (recursive CTE):

```
WITH RECURSIVE subordinates AS (
  SELECT id, name, manager_id FROM employees WHERE id = 123
  UNION ALL
  SELECT e.id, e.name, e.manager_id
  FROM employees e
  JOIN subordinates s ON e.manager_id = s.id
)
SELECT * FROM subordinates;
```

Lesson: Self-joins for direct relationships, recursive CTEs for transitive closure.

7. Date Calculations

Problem: "Users who signed up in the last 30 days and haven't logged in."

Solution:

```
SELECT u.*
FROM users u
LEFT JOIN logins l ON u.id = l.user_id AND l.login_date > NOW() - INTERVAL '30 days'
WHERE u.created_at > NOW() - INTERVAL '30 days'
  AND l.id IS NULL;
```

Breakdown:

- `u.created_at > NOW() - INTERVAL '30 days'` : Signed up recently
- `LEFT JOIN ... AND l.login_date > NOW() - INTERVAL '30 days'` : Check recent logins
- `l.id IS NULL` : No recent logins

Common mistake:

```
SELECT u.*
FROM users u
WHERE u.created_at > NOW() - INTERVAL '30 days'
  AND NOT EXISTS (SELECT 1 FROM logins WHERE user_id = u.id);
```

This finds users with NO logins EVER, not "no logins in last 30 days."

8. Pivot Tables

Problem: "Count users by sign-up month and country."

Solution (FILTER):

```
SELECT
  DATE_TRUNC('month', created_at) AS month,
  COUNT(*) FILTER (WHERE country = 'US') AS us_count,
  COUNT(*) FILTER (WHERE country = 'UK') AS uk_count,
  COUNT(*) FILTER (WHERE country = 'CA') AS ca_count
FROM users
GROUP BY month;
```

Alternative (CASE):

```
SELECT
  DATE_TRUNC('month', created_at) AS month,
  COUNT(CASE WHEN country = 'US' THEN 1 END) AS us_count,
  COUNT(CASE WHEN country = 'UK' THEN 1 END) AS uk_count,
  COUNT(CASE WHEN country = 'CA' THEN 1 END) AS ca_count
FROM users
GROUP BY month;
```

Dynamic pivot (crosstab extension):

```
SELECT * FROM crosstab(
  'SELECT DATE_TRUNC(''month'', created_at), country, COUNT(*) FROM users GROUP BY
```

```
1, 2 ORDER BY 1, 2'
) AS ct(month date, US bigint, UK bigint, CA bigint);
```

Lesson: `FILTER` or `CASE` for static pivots, `crosstab()` for dynamic.

Real-World Scenarios

Scenario 1: Debugging Incorrect Counts

Problem: "Why is this returning the wrong count?"

```
SELECT u.name, COUNT(o.id)
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE o.status = 'completed';
```

Expected: All users, with order counts.

Actual: Only users with completed orders.

Why: `WHERE` filters *after* the JOIN but *before* aggregation. NULL rows (users without orders) fail the `o.status = 'completed'` check.

Fix:

```
SELECT u.name, COUNT(o.id)
FROM users u
LEFT JOIN orders o ON u.id = o.user_id AND o.status = 'completed'
GROUP BY u.name;
```

Lesson: Conditions on the "right" table of a LEFT JOIN go in the `ON` clause, not `WHERE`.

Scenario 2: Slow Query

Problem: "This query is slow."

```
SELECT u.*, COUNT(o.id) AS order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id;
```

Check EXPLAIN ANALYZE:

```
EXPLAIN ANALYZE
SELECT u.*, COUNT(o.id) AS order_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
GROUP BY u.id;
```

Possible issues:

1. **Sequential scan on `users`:**
   - Add `WHERE` clause or index if filtering.
2. **Sequential scan on `orders`:**
   - Add index on `user_id`: `CREATE INDEX idx_orders_user_id ON orders(user_id);`
3. **Hash Join vs Nested Loop:**
   - If `users` is small, Nested Loop with index lookup on `orders(user_id)` is faster.
   - If `users` is large, Hash Join is better.
4. **Excessive rows in GROUP BY:**
   - Postgres must accumulate all orders per user, then count.
   - Pre-aggregate if possible:

   ```
   WITH order_counts AS (
     SELECT user_id, COUNT(*) AS cnt FROM orders GROUP BY user_id
   )
   SELECT u.*, COALESCE(oc.cnt, 0) AS order_count
   FROM users u
   LEFT JOIN order_counts oc ON u.id = oc.user_id;
   ```

Lesson: Use EXPLAIN ANALYZE to diagnose, don't guess.

Scenario 3: Report Query

Problem: "Monthly sales by product category, with year-over-year growth."

Solution:

```
WITH monthly_sales AS (
  SELECT
    DATE_TRUNC('month', o.order_date) AS month,
    c.name AS category,
    SUM(oi.quantity * oi.price) AS sales
  FROM orders o
  JOIN order_items oi ON o.id = oi.order_id
  JOIN products p ON oi.product_id = p.id
  JOIN categories c ON p.category_id = c.id
  GROUP BY month, c.id, c.name
)
SELECT
  ms.month,
  ms.category,
  ms.sales,
  LAG(ms.sales, 12) OVER (PARTITION BY ms.category ORDER BY ms.month) AS
sales_year_ago,
  (ms.sales - LAG(ms.sales, 12) OVER (PARTITION BY ms.category ORDER BY ms.month))
    / NULLIF(LAG(ms.sales, 12) OVER (PARTITION BY ms.category ORDER BY ms.month), 0)
```

```sql
* 100 AS yoy_growth_pct
FROM monthly_sales ms;
```

Breakdown:

- **CTE:** Aggregate monthly sales by category.
- **LAG(sales, 12):** Sales 12 months ago.
- **NULLIF:** Prevent division by zero.

Lesson: CTEs + window functions for complex reports.

Scenario 4: Efficiently Paginating

Problem: "Paginate orders with OFFSET."

Naive:

```sql
SELECT * FROM orders ORDER BY created_at DESC LIMIT 20 OFFSET 1000;
```

Problem: Database must scan and discard the first 1000 rows every time.

Better (keyset pagination):

```sql
-- Page 1
SELECT * FROM orders ORDER BY created_at DESC, id DESC LIMIT 20;

-- Page 2 (where last_created_at and last_id are from the previous page)
SELECT * FROM orders
WHERE (created_at, id) < (last_created_at, last_id)
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

Advantages:

- Only scans relevant rows.
- Consistent results (no duplicates/missing rows if data changes).

Disadvantages:

- Can't jump to arbitrary page numbers.

Lesson: Use keyset pagination for large datasets.

Scenario 5: Detecting Anomalies

Problem: "Find orders where total != sum(item prices)."

Solution:

```sql
SELECT o.id, o.total, SUM(oi.quantity * oi.price) AS computed_total
FROM orders o
JOIN order_items oi ON o.id = oi.order_id
GROUP BY o.id, o.total
HAVING o.total != SUM(oi.quantity * oi.price);
```

Lesson: `HAVING` for filtering on aggregates.

Scenario 6: Finding Active Users

Problem: "Users who logged in at least 5 days in the last 30 days."

Solution:

```
SELECT user_id
FROM logins
WHERE login_date > NOW() - INTERVAL '30 days'
GROUP BY user_id
HAVING COUNT(DISTINCT DATE(login_date)) >= 5;
```

Why `COUNT(DISTINCT DATE(...))`: If a user logs in multiple times per day, count each day once.

Scenario 7: Soft Delete Trap

Problem: "Why am I seeing deleted records?"

Schema:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT,
  deleted_at TIMESTAMPTZ
);
```

Query:

```
SELECT * FROM users WHERE email = 'alice@example.com';
```

Returns deleted users!

Fix:

```
SELECT * FROM users WHERE email = 'alice@example.com' AND deleted_at IS NULL;
```

Better: Use views.

```
CREATE VIEW active_users AS
SELECT * FROM users WHERE deleted_at IS NULL;

-- Query the view
SELECT * FROM active_users WHERE email = 'alice@example.com';
```

Lesson: Soft deletes require discipline. Use views to enforce filters.

Scenario 8: Time Zone Confusion

Problem: "Orders created today."

Wrong:

```sql
SELECT * FROM orders WHERE DATE(created_at) = CURRENT_DATE;
```

Problem: `created_at` is UTC, `CURRENT_DATE` is server time zone. Mismatch!

Right:

```sql
SELECT * FROM orders
WHERE created_at >= CURRENT_DATE AT TIME ZONE 'UTC'
  AND created_at < (CURRENT_DATE + 1) AT TIME ZONE 'UTC';
```

Or store time zone:

```sql
SELECT * FROM orders
WHERE created_at AT TIME ZONE 'America/New_York' >= CURRENT_DATE
  AND created_at AT TIME ZONE 'America/New_York' < CURRENT_DATE + 1;
```

Lesson: Always specify time zones explicitly.

Debugging Strategies

1. Simplify the Query

Complex query returning wrong results?

Break it down:

```sql
-- Start with the FROM clause
SELECT * FROM orders;

-- Add JOIN
SELECT * FROM orders o JOIN order_items oi ON o.id = oi.order_id;

-- Add WHERE
SELECT * FROM orders o JOIN order_items oi ON o.id = oi.order_id WHERE o.status =
'completed';

-- Add GROUP BY
SELECT o.id, COUNT(*) FROM orders o JOIN order_items oi ON o.id = oi.order_id WHERE
o.status = 'completed' GROUP BY o.id;
```

Lesson: Build incrementally, verify each step.

2. Check for NULLs

Problem: "WHERE status != 'canceled'" doesn't return all non-canceled orders.

Why: `NULL != 'canceled'` is UNKNOWN, not TRUE.

Fix:

```sql
WHERE (status != 'canceled' OR status IS NULL);
-- Or
WHERE status IS DISTINCT FROM 'canceled';
```

3. Count Rows at Each Stage

Problem: "I expected 100 rows, got 10."

Debug:

```sql
SELECT COUNT(*) FROM orders;  -- 100
SELECT COUNT(*) FROM order_items;  -- 500
SELECT COUNT(*) FROM orders o JOIN order_items oi ON o.id = oi.order_id;  -- 500
(join explodes!)
SELECT COUNT(DISTINCT o.id) FROM orders o JOIN order_items oi ON o.id = oi.order_id;
-- 100
```

Lesson: JOINs can multiply rows. Use `DISTINCT` or aggregate carefully.

4. Test with Small Data

Problem: "Query works on test data, fails in production."

Why: Small datasets hide cardinality issues (join explosions, N+1, etc.).

Fix: Test with realistic data volumes.

Performance Trade-Offs

Trade-Off 1: DISTINCT vs GROUP BY

Goal: Get unique user IDs.

Option 1 (DISTINCT):

```sql
SELECT DISTINCT user_id FROM orders;
```

Option 2 (GROUP BY):

```sql
SELECT user_id FROM orders GROUP BY user_id;
```

Performance: Usually equivalent. Postgres optimizes both similarly.

When GROUP BY is better: When you also need aggregates.

```sql
SELECT user_id, COUNT(*) FROM orders GROUP BY user_id;
```

Trade-Off 2: EXISTS vs JOIN

Goal: "Users with at least one order."

Option 1 (EXISTS):

```
SELECT * FROM users u WHERE EXISTS (SELECT 1 FROM orders WHERE user_id = u.id);
```

Option 2 (JOIN + DISTINCT):

```
SELECT DISTINCT u.* FROM users u JOIN orders o ON u.id = o.user_id;
```

Performance: `EXISTS` is often faster (stops at first match). `JOIN` must scan all orders.

Lesson: Use `EXISTS` for existence checks.

Trade-Off 3: Subquery vs CTE

Goal: Multi-stage query.

Option 1 (subquery):

```
SELECT * FROM (
  SELECT user_id, COUNT(*) AS cnt FROM orders GROUP BY user_id
) sub WHERE cnt > 10;
```

Option 2 (CTE):

```
WITH order_counts AS (
  SELECT user_id, COUNT(*) AS cnt FROM orders GROUP BY user_id
)
SELECT * FROM order_counts WHERE cnt > 10;
```

Performance: Usually identical. CTE is more readable.

Exception: CTEs with `MATERIALIZED` hint can force materialization (useful for repeated references).

Trade-Off 4: Index Scan vs Sequential Scan

Query:

```
SELECT * FROM orders WHERE status = 'completed';
```

If 90% of orders are completed:

- **Sequential scan is faster** (fewer random I/Os).

If 1% of orders are completed:

- **Index scan is faster** (jump directly to matching rows).

Postgres chooses automatically based on statistics.

Lesson: Don't force index usage unless profiling shows benefit.

Final Practical Questions

Question 1: Explain This Query

```sql
SELECT
  u.name,
  COUNT(o.id) AS order_count,
  COALESCE(SUM(o.total), 0) AS total_spent
FROM users u
LEFT JOIN orders o ON u.id = o.user_id AND o.status = 'completed'
GROUP BY u.id, u.name;
```

What does it do?

- All users
- Count of completed orders per user
- Total spent on completed orders

Why `COALESCE`?

- `SUM(NULL)` returns NULL. `COALESCE(NULL, 0)` converts to 0.

Why `AND o.status = 'completed'` in the ON clause?

- Left join means "include all users." Filtering in `ON` only excludes orders, not users.

Question 2: Fix This Query

```sql
SELECT u.name, o.total
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE o.total > 100;
```

Problem: Turns the LEFT JOIN into an INNER JOIN (excludes users without orders).

Fix:

```sql
SELECT u.name, o.total
FROM users u
LEFT JOIN orders o ON u.id = o.user_id AND o.total > 100;
```

Now: Includes all users, but only shows orders > 100.

Question 3: Optimize This Query

```sql
SELECT
  p.name,
  (SELECT COUNT(*) FROM reviews r WHERE r.product_id = p.id) AS review_count
FROM products p;
```

Problem: Correlated subquery (runs once per product).

Fix:

```sql
SELECT
  p.name,
  COUNT(r.id) AS review_count
FROM products p
LEFT JOIN reviews r ON p.id = r.product_id
GROUP BY p.id, p.name;
```

Or with window function:

```sql
SELECT DISTINCT
  p.id,
  p.name,
  COUNT(r.id) OVER (PARTITION BY p.id) AS review_count
FROM products p
LEFT JOIN reviews r ON p.id = r.product_id;
```

Key Takeaways

1. **Window functions** (ROW_NUMBER, RANK, LAG, SUM OVER) solve 80% of complex interview problems.
2. **Gaps and islands:** Subtract `ROW_NUMBER()` from a sequence to find consecutive ranges.
3. **Self-joins** for hierarchies, recursive CTEs for transitive closure.
4. **LEFT JOIN conditions** go in `ON`, not `WHERE`.
5. **COUNT(DISTINCT DATE(...))** for counting unique days.
6. **Keyset pagination** for large datasets.
7. **EXISTS** is faster than `JOIN + DISTINCT` for existence checks.
8. **EXPLAIN ANALYZE** before optimizing.
9. **Test edge cases:** NULLs, empty tables, join explosions.
10. **Break complex queries into CTEs** for debuggability.

You've reached the end of the SQL deep dive!

Next steps:

- Practice with real datasets (Kaggle, LeetCode SQL)
- Profile queries with EXPLAIN ANALYZE
- Read Postgres documentation (it's excellent)
- Write raw SQL, even when your ORM could do it—build intuition

Remember: SQL is declarative. You describe *what* you want, the optimizer figures out *how*. But you still need to understand *how* to debug *what* went wrong.