SQL Habits For Busy Developers

7 real-world SQL tips every developer should know

beekeeperstudio.io

Inside you'll find 7 practical SQL habits, like:

- Be specific: avoid SELECT *
- Use CTEs to break up messy logic
- Filter before you join (and keep it index-friendly)

…and more patterns to make your SQL safer, faster, and easier to debug.

Table Of Contents

Write Queries You'll Still Understand Tomorrow

Be specific: avoid SELECT *

Rule: Fetch only the columns you need. SELECT * is convenient but risky.

```sql
-- Pulls every column, can break on schema changes
SELECT * FROM user_profiles WHERE user_id = ?;

-- Explicit columns show intent and avoid surprises
SELECT user_id, display_name, avatar_url, last_login_ip
FROM user_profiles WHERE user_id = ?;
```

Why this matters: Explicit columns reduce surprises. From schema changes, to extra data and duplicate field names.

Use CTEs to break up complex logic

Rule: Use WITH Common Table Expressions (CTEs) to split complex queries into named steps you can SELECT from and test in isolation.

```sql
-- Hard to follow: nested subquery
SELECT customer_id, COUNT(*)
FROM (
  SELECT customer_id, created_at
  FROM orders
  WHERE created_at > now() - interval '7 days'
) o
GROUP BY customer_id;


-- Clearer: same logic with a CTE defined via WITH
WITH recent_orders AS (
  SELECT customer_id, created_at
  FROM orders
  WHERE created_at > now() - interval '7 days'
)
SELECT customer_id, COUNT(*)
FROM recent_orders
GROUP BY customer_id;
```

Why this matters: CTEs clarify intent and simplify debugging and reuse. However, some databases materialize them, so inline subqueries may be faster in hot paths.

Comment your intent, not just your logic

Rule: Use comments to explain why the query exists, not just what it does.

```SQL
-- Find users who signed up but never activated
-- (used in the weekly retention cohort report)
SELECT
  u.id,
  u.email
FROM users u
LEFT JOIN events e
  ON u.id = e.user_id AND e.name = 'account_activated'
WHERE e.id IS NULL
  AND u.created_at > now() - interval '7 days';
```

Why this matters: SQL shows *what* a query does but not *why*. A short intent comment preserves context for future readers and reviewers.

Make Your Queries Faster

Make your filters index-friendly

Rule: Write filters that match how data is stored so the database can use indexes.

```sql
-- Forces a full scan: index on email can’t be used
WHERE LOWER(email) = 'test@example.com';

-- Uses the index if values are stored lowercase
WHERE email = 'test@example.com';

-- Blocks index on created_at
WHERE DATE(created_at) = '2025-08-04';

-- Range filter keeps index usable
WHERE created_at >= '2025-08-04'
  AND created_at < '2025-08-05';
```

Why this matters: Queries run faster and more consistently when filters use indexes. If you need a transformation, create a functional index instead.

Filter before you join

Rule: Filter each table before joining so you combine only the rows that matter.

```sql
-- Joins every user, even inactive ones
SELECT e.*
FROM users u
JOIN events e ON e.user_id = u.id;
```

```sql
-- Join only active, recently seen users
WITH active_users AS (
  SELECT id
  FROM users
  WHERE active = true
    AND last_seen > now() - interval '90 days'
)
SELECT e.*
FROM active_users u
JOIN events e ON e.user_id = u.id;
```

Why this matters: Filtering first avoids unnecessary data bloat, helps the optimizer choose a better plan, and makes the query's purpose clearer.

Get Safer, More Predictable Results

Use EXISTS instead of IN

Rule: Use EXISTS rather than IN to test for related rows. It's more reliable and efficient.

```sql
-- Inefficient: checks every row in the subquery
-- Unreliable: NULLs can make comparisons fail
SELECT *
FROM users
WHERE id IN (SELECT user_id FROM orders);


-- Efficient: stops at the first match
-- Reliable: works even if NULLs appear
SELECT *
FROM users u
WHERE EXISTS (
  SELECT 1 FROM orders o
  WHERE o.user_id = u.id
);
```

Why this matters: EXISTS is safer and faster. It stops once it finds a match and doesn't get tripped up by NULL values, which can cause IN to skip rows you expect to match.

Always specify ORDER BY

Rule: Row order is undefined unless you add ORDER BY. Make it explicit.

```sql
-- Unstable: row order is undefined and may change
SELECT id, name
FROM users
WHERE active = true
LIMIT 10;

-- Predictable: rows are sorted by a stable column
SELECT id, name
FROM users
WHERE active = true
ORDER BY created_at DESC
LIMIT 10;
```

Why this matters: Without ORDER BY, results can change between runs or environments. Sorting explicitly gives you consistent, reliable output, which is especially important for pagination or cached queries.