Query Execution Order: Why Your Intuition Is Wrong

The Problem

You write SQL top-to-bottom:

```
SELECT ...
FROM ...
WHERE ...
GROUP BY ...
```

But the database executes it in a completely different order. And until you internalize this, you'll keep hitting mysterious errors.

The Logical Execution Order

Here's the order SQL **logically** processes a query:

```
1. FROM / JOIN       → Identify and combine source tables
2. WHERE             → Filter rows
3. GROUP BY          → Aggregate rows into groups
4. HAVING            → Filter groups
5. SELECT            → Project columns and apply expressions
6. DISTINCT          → Remove duplicate rows
7. ORDER BY          → Sort result set
8. LIMIT / OFFSET    → Restrict result set
```

Notice: **SELECT comes FIFTH.**

This is why you can't reference a SELECT alias in WHERE. The alias doesn't exist yet when WHERE is evaluated.

Let's Break It Down (With Real Examples)

Step 1: FROM / JOIN — Build the Working Dataset

The database starts here. It identifies all tables and joins them.

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

What happens:

1. Database loads `users` (or prepares to scan it)
2. Database loads `orders`
3. Database performs the join (nested loop, hash, or merge)
4. **Result:** A combined rowset with columns from both tables

At this point, nothing is filtered. The database has a (potentially huge) intermediate result set.

Step 2: WHERE — Filter Rows

Now the database applies row-level filters.

```
WHERE o.total > 100
```

Key insight: WHERE operates on **rows**, not groups. It filters **before** aggregation.

Common Mistake #1: Trying to Filter on Aggregates in WHERE

```
SELECT user_id, COUNT(*) as order_count
FROM orders
WHERE order_count > 5  -- ERROR!
GROUP BY user_id;
```

Why it fails: At the WHERE stage, rows aren't grouped yet. `order_count` doesn't exist.

Fix: Use HAVING (comes after GROUP BY):

```
SELECT user_id, COUNT(*) as order_count
FROM orders
GROUP BY user_id
HAVING COUNT(*) > 5;
```

Common Mistake #2: Using SELECT Aliases in WHERE

```
SELECT price * 1.1 AS price_with_tax
FROM products
WHERE price_with_tax > 100;  -- ERROR!
```

Why it fails: SELECT hasn't executed yet when WHERE runs.

Fix: Repeat the expression:

```
SELECT price * 1.1 AS price_with_tax
FROM products
WHERE price * 1.1 > 100;
```

Or use a subquery/CTE:

```
WITH prices AS (
  SELECT price * 1.1 AS price_with_tax FROM products
)
SELECT * FROM prices WHERE price_with_tax > 100;
```

Step 3: GROUP BY — Collapse Rows into Groups

This is where aggregation happens. Rows with the same GROUP BY values get collapsed into a single row.

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

What happens:

- Database scans through filtered rows
- Groups rows by `user_id`
- For each group, it can compute aggregates (COUNT, SUM, AVG, etc.)

Critical rule: After GROUP BY, you can only SELECT:

1. Columns in the GROUP BY clause
2. Aggregate functions

Example: Valid vs Invalid

Valid:

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

Invalid (in Postgres/MySQL strict mode):

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

Why invalid? What's `order_id` here? Each user has multiple orders. Which one should it show? **Undefined.**

MySQL used to allow this and pick an arbitrary value. Postgres always rejected it. Modern MySQL (with `ONLY_FULL_GROUP_BY`) rejects it too.

PostgreSQL Functional Dependency Exception

Postgres is smart about primary keys:

```
SELECT user_id, email, COUNT(*)
FROM orders
JOIN users ON orders.user_id = users.id
GROUP BY users.id;  -- Notice: only grouping by users.id
```

This works because Postgres knows `users.id` is a primary key. If you group by `users.id`, all columns in `users` are functionally determined. So it lets you SELECT `users.email` even though it's not in GROUP BY.

MySQL doesn't do this. You'd need:

```
GROUP BY users.id, users.email
```

Step 4: HAVING — Filter Groups

HAVING is like WHERE, but for **groups** instead of rows.

```
SELECT user_id, COUNT(*) as order_count
FROM orders
GROUP BY user_id
HAVING COUNT(*) > 5;
```

What happens:

- Database computes COUNT(*) for each group
- Filters out groups where COUNT(*) ≤ 5

When to Use WHERE vs HAVING

Use WHERE when filtering rows before grouping:

```
-- "Count orders over $100 per user, for users with >5 such orders"
SELECT user_id, COUNT(*) as expensive_orders
FROM orders
WHERE total > 100
GROUP BY user_id
HAVING COUNT(*) > 5;
```

Use HAVING when filtering based on aggregate results:

```
-- "Users with total order value > $10,000"
SELECT user_id, SUM(total) as total_spent
FROM orders
GROUP BY user_id
HAVING SUM(total) > 10000;
```

Performance tip: WHERE filters early (fewer rows to group). HAVING filters late (after aggregation). Push filters into WHERE when possible.

Common Mistake: Filtering in HAVING When WHERE Would Work

Slow:

```
SELECT user_id, COUNT(*)
FROM orders
GROUP BY user_id
HAVING user_id > 1000;  -- Bad! This isn't an aggregate condition
```

Fast:

```
SELECT user_id, COUNT(*)
FROM orders
WHERE user_id > 1000  -- Filter before grouping
GROUP BY user_id;
```

The second version filters before aggregation, reducing work.

Step 5: SELECT — Project Columns and Apply Expressions

Finally, SELECT runs. This is where:

- Aliases are created
- Functions are applied
- Columns are projected

```
SELECT
  user_id,
  COUNT(*) as order_count,
  ROUND(AVG(total), 2) as avg_order_total
FROM orders
GROUP BY user_id;
```

What happens:

- Database takes grouped rows
- Computes aggregate expressions
- Creates aliases (`order_count` , `avg_order_total`)

Key insight: Aliases are now available for ORDER BY and LIMIT, but *not* for WHERE/GROUP BY/HAVING (they already executed).

Common Mistake: Referencing SELECT Aliases in SELECT

This doesn't work:

```
SELECT
  price * 1.1 AS price_with_tax,
  price_with_tax * 0.9 AS discounted  -- ERROR!
FROM products;
```

Why? SELECT evaluates expressions simultaneously, not sequentially.

Fix:

```
SELECT
  price * 1.1 AS price_with_tax,
  price * 1.1 * 0.9 AS discounted
FROM products;
```

Or use a subquery:

```
SELECT
  price_with_tax,
  price_with_tax * 0.9 AS discounted
FROM (
```

```
  SELECT price * 1.1 AS price_with_tax FROM products
) sub;
```

Step 6: DISTINCT — Remove Duplicates

If you use DISTINCT, it happens after SELECT:

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

What happens:

- Database projects `user_id` column
- Removes duplicate values

Performance note: DISTINCT requires sorting or hashing. If you can avoid it with proper joins or GROUP BY, do so.

DISTINCT vs GROUP BY

These are equivalent:

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

GROUP BY is often faster because it's optimized for aggregation. But for simple de-duplication, DISTINCT is clearer.

Step 7: ORDER BY — Sort Results

ORDER BY is **the last meaningful operation** (before LIMIT).

```
SELECT user_id, SUM(total) as total_spent
FROM orders
GROUP BY user_id
ORDER BY total_spent DESC;
```

What happens:

- Database takes the result set from SELECT
- Sorts it by `total_spent`

Key insight: ORDER BY **can** reference SELECT aliases because SELECT already executed.

Common Mistake: Ordering by Columns Not in SELECT

```
SELECT user_id FROM orders ORDER BY created_at;
```

Is this valid?

- **In Postgres:** Yes, if `created_at` is in the source table
- **With DISTINCT:** No! Because after DISTINCT, `created_at` is gone
- **Best practice:** Make it explicit in SELECT if you're ordering by it

Ordering by Aggregate Expressions

You can order by expressions:

```
SELECT user_id, COUNT(*) as order_count
FROM orders
GROUP BY user_id
ORDER BY COUNT(*) DESC;  -- Or ORDER BY order_count DESC
```

Both work because ORDER BY happens after SELECT.

Step 8: LIMIT / OFFSET — Restrict Rows

Finally, LIMIT restricts how many rows to return.

```
SELECT user_id, SUM(total) as total_spent
FROM orders
GROUP BY user_id
ORDER BY total_spent DESC
LIMIT 10;
```

What happens:

- Database sorts by `total_spent`
- Takes the top 10 rows
- Returns them

Performance note: The database still has to compute ALL results before applying LIMIT. If you have an index supporting the ORDER BY, it can stop early. Otherwise, it computes everything, sorts, then truncates.

LIMIT + OFFSET for Pagination

```
SELECT * FROM users ORDER BY created_at DESC LIMIT 10 OFFSET 20;
```

What this means: "Skip the first 20 rows, then return the next 10."

Performance trap: OFFSET is expensive for large offsets. If `OFFSET 100000`, the database still has to materialize 100,000 rows and throw them away.

Better pagination: Keyset pagination (we'll cover this in the performance section).

Putting It All Together: A Complex Example

Let's trace execution:

```
SELECT
  u.country,
  COUNT(DISTINCT u.id) as user_count,
  ROUND(AVG(o.total), 2) as avg_order_total
FROM users u
JOIN orders o ON u.id = o.user_id
WHERE o.created_at >= '2024-01-01'
```

```
  AND u.active = true
GROUP BY u.country
HAVING COUNT(DISTINCT u.id) > 100
ORDER BY avg_order_total DESC
LIMIT 5;
```

Execution Trace

1. **FROM/JOIN:**
   - Load `users` and `orders`
   - Perform JOIN on `u.id = o.user_id`
   - Result: Combined rowset with all user-order pairs
2. **WHERE:**
   - Filter rows where `o.created_at >= '2024-01-01'`
   - Filter rows where `u.active = true`
   - Result: Subset of rows matching both conditions
3. **GROUP BY:**
   - Group rows by `u.country`
   - For each country group, prepare to compute aggregates
4. **HAVING:**
   - Compute `COUNT(DISTINCT u.id)` for each group
   - Filter out groups where count ≤ 100
   - Result: Only countries with >100 users
5. **SELECT:**
   - Project `u.country`
   - Compute `COUNT(DISTINCT u.id)` as `user_count`
   - Compute `ROUND(AVG(o.total), 2)` as `avg_order_total`
   - Result: Three columns per group
6. **ORDER BY:**
   - Sort result set by `avg_order_total DESC`
7. **LIMIT:**
   - Take top 5 rows

Why This Matters

If you tried to write:

```
WHERE user_count > 100  -- ERROR! user_count doesn't exist yet
```

It would fail. Because WHERE executes before SELECT.

Similarly:

```
WHERE AVG(o.total) > 50  -- ERROR! Can't aggregate in WHERE
```

Aggregates don't exist until GROUP BY. Use HAVING instead.

Real-World Debugging Scenarios

Bug #1: "Column must appear in GROUP BY"

Code:

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

Error: `column "orders.email" must appear in the GROUP BY clause or be used in an aggregate function`

Why? You're grouping by `user_id`, but selecting `email`. Which email should it show if a user has multiple orders? Undefined.

Fix: Either add `email` to GROUP BY, or use an aggregate:

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

Bug #2: "Aggregate functions are not allowed in WHERE"

Code:

```
SELECT user_id, SUM(total)
FROM orders
WHERE SUM(total) > 1000
GROUP BY user_id;
```

Error: `aggregate functions are not allowed in WHERE`

Why? WHERE filters rows before grouping. SUM doesn't exist yet.

Fix: Use HAVING:

```
SELECT user_id, SUM(total)
FROM orders
GROUP BY user_id
HAVING SUM(total) > 1000;
```

Bug #3: "Column does not exist" (Alias in WHERE)

Code:

```sql
SELECT price * 1.1 AS price_with_tax
FROM products
WHERE price_with_tax > 100;
```

Error: `column "price_with_tax" does not exist`

Why? WHERE executes before SELECT. The alias isn't created yet.

Fix: Repeat the expression:

```sql
SELECT price * 1.1 AS price_with_tax
FROM products
WHERE price * 1.1 > 100;
```

Bug #4: ORDER BY Ambiguity

Code:

```sql
SELECT user_id, created_at FROM orders
UNION
SELECT user_id, updated_at FROM users
ORDER BY created_at;
```

Error: `column "created_at" does not exist` (or ambiguous)

Why? After UNION, the result set combines both queries. One has `created_at`, the other has `updated_at` in that position. The column names might not align.

Fix: Use positional ordering or alias:

```sql
ORDER BY 2  -- Order by second column
```

Or unify aliases:

```sql
SELECT user_id, created_at as date FROM orders
UNION
SELECT user_id, updated_at as date FROM users
ORDER BY date;
```

Logical vs Physical Execution

Important nuance: What we've described is the **logical** execution order. The physical execution can be different.

Example: Predicate Pushdown

You write:

```sql
SELECT * FROM (
  SELECT * FROM orders WHERE total > 100
```

```
) sub
WHERE user_id = 5;
```

Logically, this filters `total > 100` in the subquery, then filters `user_id = 5` in the outer query.

Physically, the optimizer might push both predicates down and execute:

```
SELECT * FROM orders WHERE total > 100 AND user_id = 5;
```

Why? Because it's more efficient. The optimizer rewrites your query.

Example: Join Reordering

You write:

```
SELECT *
FROM users u
JOIN orders o ON u.id = o.user_id
JOIN products p ON o.product_id = p.id;
```

Logically, this joins `users` → `orders` → `products`.

Physically, the optimizer might join `orders` → `products` first if it's cheaper, then join `users`.

You don't control this. The optimizer does.

Practical Implications

1. Write Filters as Early as Possible

Put conditions in WHERE, not HAVING, when possible:

Slow:

```
SELECT user_id, COUNT(*)
FROM orders
GROUP BY user_id
HAVING user_id < 1000;
```

Fast:

```
SELECT user_id, COUNT(*)
FROM orders
WHERE user_id < 1000
GROUP BY user_id;
```

The second filters early, reducing rows to aggregate.

2. Be Explicit About What You Need

Don't SELECT unnecessary columns. The database might materialize them even if you don't use them:

Wasteful:

```
SELECT * FROM orders WHERE id = 123;
```

Efficient:

```
SELECT id, total, created_at FROM orders WHERE id = 123;
```

Especially true for wide tables with TEXT or JSONB columns.

3. Avoid Functions in WHERE on Indexed Columns

Non-sargable (can't use index):

```
WHERE LOWER(email) = 'user@example.com'
```

The function prevents index usage.

Sargable:

```
WHERE email = 'user@example.com'  -- Assumes email is stored lower-case
```

Or create a functional index:

```
CREATE INDEX idx_users_email_lower ON users(LOWER(email));
```

4. Understand What DISTINCT Costs

DISTINCT requires de-duplication (sorting or hashing). If you're doing:

```
SELECT DISTINCT user_id FROM orders WHERE user_id IN (1, 2, 3);
```

And you know `user_id` is already unique, don't use DISTINCT. It's wasted work.

Interactive Exercise: Predict the Error

Try to predict what's wrong before looking at the answer.

Query 1

```
SELECT order_id, SUM(quantity * price) AS total
FROM order_items
WHERE total > 100
GROUP BY order_id;
```

▶ Answer

Query 2

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

▶ Answer

Query 3

```sql
SELECT product_id, SUM(quantity) AS total_sold
FROM sales
WHERE SUM(quantity) > 100
GROUP BY product_id;
```

▶ Answer

Key Takeaways

1. **Execution order ≠ writing order.** Internalize FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT.
2. **WHERE filters rows, HAVING filters groups.** Use each appropriately.
3. **Aliases don't exist until SELECT runs.** You can't reference them in WHERE/GROUP BY/HAVING.
4. **Aggregates don't exist until GROUP BY runs.** You can't use COUNT/SUM in WHERE.
5. **The optimizer can reorder operations physically**, but you should understand the logical order to write correct queries.

Next up: We'll dive into JOINs, where execution order becomes even more critical.