GROUP BY and Aggregations: Where Developers Freeze

The Fundamental Problem

GROUP BY collapses rows. This is where set-based thinking becomes critical.

When you write:

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

You're saying: "Take the set of orders, partition it by `user_id`, collapse each partition into a single row, and count the rows in each partition."

The collapse is irreversible. Once you GROUP BY, you can only SELECT:

1. Columns in the GROUP BY clause
2. Aggregate functions (COUNT, SUM, AVG, etc.)

Anything else is **meaningless**—which order row should it pick?

GROUP BY Semantics

What GROUP BY Actually Does

```
SELECT category, AVG(price)
FROM products
GROUP BY category;
```

Conceptual execution:

1. Take all products
2. Partition them into groups by category
3. For each group, compute AVG(price)
4. Return one row per group

Visually:

```
Before GROUP BY:
product_id | category   | price
-----------|------------|------
1          | Books      | 10
2          | Books      | 20
3          | Electronics| 100
4          | Electronics| 200

After GROUP BY category:
category    | AVG(price)
------------|------------
```

```
Books       | 15
Electronics | 150
```

The Critical Rule: SELECT Only What Makes Sense

Valid:

```
SELECT category, COUNT(*), AVG(price)
FROM products
GROUP BY category;
```

Invalid:

```
SELECT category, product_id, COUNT(*)
FROM products
GROUP BY category;
```

Why invalid? Each category has multiple product IDs. Which one should SQL show? Undefined.

PostgreSQL's Functional Dependency Rule

Postgres is smart about primary keys:

```
-- This works in Postgres:
SELECT user_id, email, COUNT(*)
FROM users
JOIN orders ON users.id = orders.user_id
GROUP BY users.id;
```

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

MySQL (strict mode): Requires all non-aggregated columns in GROUP BY:

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

Aggregate Functions

COUNT: The Tricky One

COUNT(*) vs COUNT(column)

```
SELECT COUNT(*) FROM users;       -- Counts rows
SELECT COUNT(email) FROM users;   -- Counts non-NULL emails
```

If `email` is nullable, these return different numbers.

Example:

```
id | email
---|------
1  | a@example.com
2  | NULL
3  | c@example.com
```

- `COUNT(*)` = 3
- `COUNT(email)` = 2

COUNT(DISTINCT column)

```sql
SELECT COUNT(DISTINCT user_id) FROM orders;
```

What it does: Counts unique user IDs (de-duplicates first).

Performance note: DISTINCT requires sorting or hashing. Can be expensive on large datasets.

SUM and AVG: NULLs Matter

```sql
SELECT AVG(rating) FROM reviews;
```

Behavior: NULLs are **ignored**.

Example: ``` rating

5 4 NULL 3

```
- `SUM(rating)` = 12
- `AVG(rating)` = 4 (12 / 3, not 12 / 4)

**Trap:** If all values are NULL, `AVG` returns NULL, not 0.

### MIN and MAX: Simple, But Watch for NULLs

```sql
SELECT MIN(price), MAX(price) FROM products;
```

Behavior: Ignores NULLs.

Edge case: If all prices are NULL, returns NULL.

STRING_AGG (Postgres) / GROUP_CONCAT (MySQL): Concatenate Strings

```sql
-- Postgres:
SELECT user_id, STRING_AGG(product_name, ', ' ORDER BY product_name)
FROM orders
GROUP BY user_id;

-- MySQL:
```
```

```
SELECT user_id, GROUP_CONCAT(product_name ORDER BY product_name SEPARATOR ', ')
FROM orders
GROUP BY user_id;
```

Result:

```
user_id | products
--------|------------------------
1       | Book, Laptop, Mouse
2       | Phone, Tablet
```

Use case: Flattening related data into a comma-separated list.

ARRAY_AGG (Postgres): Collect into Array

```
SELECT user_id, ARRAY_AGG(product_name ORDER BY product_name)
FROM orders
GROUP BY user_id;
```

Result:

```
user_id | products
--------|------------------------
1       | ["Book", "Laptop", "Mouse"]
2       | ["Phone", "Tablet"]
```

Use case: When you need an actual array (e.g., for JSONB processing).

JSONB_AGG (Postgres): Aggregate into JSON

```
SELECT user_id, JSONB_AGG(JSONB_BUILD_OBJECT('product', product_name, 'qty',
quantity))
FROM order_items
GROUP BY user_id;
```

Result:

```
user_id | items
--------|-----------------------------------------------
1       | [{"product": "Book", "qty": 2}, {"product": "Laptop", "qty": 1}]
```

Use case: Returning nested JSON from SQL (API responses).

HAVING: Filtering Aggregated Results

Remember: **WHERE filters rows before grouping, HAVING filters groups after aggregation.**

Valid Use of HAVING

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

What happens:

1. Group orders by user
2. Compute COUNT for each group
3. Filter out groups where COUNT ≤ 5

Result: Only users with >5 orders.

Invalid Use: Filtering in HAVING When WHERE Would Work

Slow:

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

Fast:

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

Why it matters: WHERE filters early (fewer rows to group). HAVING filters late (after aggregation). Push non-aggregate conditions to WHERE.

HAVING with Multiple Conditions

```sql
SELECT category, AVG(price) AS avg_price
FROM products
GROUP BY category
HAVING AVG(price) > 50 AND COUNT(*) > 10;
```

What it does: Only categories with average price >$50 AND more than 10 products.

GROUP BY with Multiple Columns

Grouping by Multiple Columns

```sql
SELECT country, city, COUNT(*) AS user_count
FROM users
GROUP BY country, city;
```

What happens: Groups by unique (country, city) pairs.

Example:

```
country | city      | user_count
--------|-----------|------------
USA     | New York  | 100
USA     | Boston    | 50
Canada  | Toronto   | 75
```

Key insight: Each unique combination of GROUP BY columns becomes one row.

Order Matters in GROUP BY? No.

```
GROUP BY country, city
```

is the same as:

```
GROUP BY city, country
```

But: Order matters for indexes (we'll cover this later).

GROUP BY with JOINs: Where Things Explode

The Join Explosion Problem

Schema:

```
users (1) ----< (N) orders (N) >---- (1) products
```

Naive query:

```
SELECT
  u.name,
  COUNT(o.id) AS order_count,
  COUNT(p.id) AS product_count
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
LEFT JOIN products p ON o.product_id = p.id
GROUP BY u.id, u.name;
```

Problem: If a user has 3 orders, each with 2 products, the join creates 6 rows before grouping. COUNT gets messed up.

Fix #1: Count distinct

```
SELECT
  u.name,
  COUNT(DISTINCT o.id) AS order_count,
  COUNT(DISTINCT p.id) AS product_count
```

```
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
LEFT JOIN products p ON o.product_id = p.id
GROUP BY u.id, u.name;
```

Fix #2: Aggregate before joining

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

Cleaner, more predictable.

DISTINCT vs GROUP BY

Semantically Similar

```
-- These are equivalent:
SELECT DISTINCT category FROM products;
SELECT category FROM products GROUP BY category;
```

Both return unique categories.

When to Use Which

Use DISTINCT when:

- Simple de-duplication
- No aggregation needed

Use GROUP BY when:

- You need aggregates
- More complex grouping logic

Performance: Often similar, but GROUP BY can be faster because it's optimized for aggregation.

GROUP BY Without Aggregates: The Implicit DISTINCT

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

What it does: Returns unique user IDs (like DISTINCT).

Why you might do this: To prepare for adding aggregates later, or for semantic clarity.

ROLLUP, CUBE, GROUPING SETS: Advanced Grouping

These are for generating **subtotals** and **grand totals** in a single query.

ROLLUP: Hierarchical Subtotals

```
SELECT country, city, COUNT(*) AS user_count
FROM users
GROUP BY ROLLUP(country, city);
```

Result:

```
country | city      | user_count
--------|-----------|------------
USA     | New York  | 100
USA     | Boston    | 50
USA     | NULL      | 150        ← USA subtotal
Canada  | Toronto   | 75
Canada  | NULL      | 75         ← Canada subtotal
NULL    | NULL      | 225        ← Grand total
```

What it does: Groups by (country, city), then (country), then ().

Use case: Reporting with subtotals.

CUBE: All Combinations

```
SELECT country, city, COUNT(*) AS user_count
FROM users
GROUP BY CUBE(country, city);
```

Result includes:

- (country, city)
- (country)
- (city)
- () — grand total

Use case: OLAP-style reporting (all possible aggregations).

GROUPING SETS: Custom Combinations

```
SELECT country, city, COUNT(*) AS user_count
FROM users
GROUP BY GROUPING SETS ((country, city), (country), ());
```

What it does: Only the specified groupings (not all combinations like CUBE).

Common Logical Errors with GROUP BY

Error 1: Forgetting to Group By Non-Aggregated Columns

Code:

```sql
SELECT country, city, COUNT(*)
FROM users
GROUP BY country;
```

Error: `column "users.city" must appear in the GROUP BY clause`

Why: You're selecting `city`, but not grouping by it. Which city should SQL show? Undefined.

Fix: Add `city` to GROUP BY, or use an aggregate like `MIN(city)`.

Error 2: Using Aggregate in WHERE

Code:

```sql
SELECT user_id, COUNT(*)
FROM orders
WHERE COUNT(*) > 5
GROUP BY user_id;
```

Error: `aggregate functions are not allowed in WHERE`

Why: WHERE executes before GROUP BY. Aggregates don't exist yet.

Fix: Use HAVING:

```sql
HAVING COUNT(*) > 5
```

Error 3: Referencing SELECT Alias in HAVING

Code:

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

Does this work?

- **Postgres:** Yes (Postgres extends SQL to allow this)
- **MySQL:** Yes
- **Standard SQL:** No (must use `HAVING COUNT(*) > 5`)

Best practice: Use the aggregate expression in HAVING for portability.

Error 4: Mixing Grouped and Non-Grouped Columns

Code:

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

Error: `column "orders.email" must appear in the GROUP BY clause`

Why: `email` isn't in GROUP BY or an aggregate. Which email should SQL pick? Undefined.

Fix: Group by both, or use an aggregate:

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

Aggregating Over NULLs: The Surprises

COUNT Ignores NULLs

```
SELECT COUNT(email) FROM users;
```

If `email` is nullable, only counts non-NULL emails.

Trap: If you want to count rows with NULL emails:

```
SELECT COUNT(*) FROM users WHERE email IS NULL;
```

SUM of NULLs is NULL, Not 0

```
VALUES
  id | amount
  ---|-------
  1  | 10
  2  | NULL
  3  | 20
```

```
SELECT SUM(amount) FROM transactions;  -- Returns 30 (ignores NULL)
```

But:

```
SELECT user_id, SUM(amount) FROM transactions GROUP BY user_id;
```

If a user has only NULL amounts, their SUM is NULL, not 0.

Fix: Use COALESCE:

```sql
SELECT user_id, COALESCE(SUM(amount), 0) AS total_amount
FROM transactions
GROUP BY user_id;
```

AVG and NULLs

```sql
SELECT AVG(rating) FROM reviews;
```

NULLs are excluded from both numerator and denominator.

Example:

```
rating: 5, NULL, 3
```

- `AVG` = 4 (not 8/3 = 2.67)

To include NULLs as 0:

```sql
SELECT AVG(COALESCE(rating, 0)) FROM reviews;
```

GROUP BY and Performance

Index Usage

Query:

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

If there's an index on `user_id` : The database can scan the index and group efficiently.

Without an index: Full table scan + sort/hash.

GROUP BY Multiple Columns and Index Order

Index on (country, city) helps:

```sql
GROUP BY country, city
```

But NOT:

```sql
GROUP BY city, country  -- Index can't be used efficiently
```

Why? Indexes are ordered left-to-right. The query must match.

work_mem and Aggregation

If the grouped data doesn't fit in `work_mem` , Postgres spills to disk (slow).

Check with EXPLAIN ANALYZE:

```
->  HashAggregate  (cost=... rows=...)
     Disk Usage: 123456 kB  ← Bad! Spilled to disk
```

Fix: Increase `work_mem` (for this session):

```
SET work_mem = '256MB';
```

Or optimize the query (filter earlier with WHERE).

GROUP BY in Subqueries: Composing Aggregates

Aggregating Aggregates

Goal: "Average number of orders per user."

Wrong:

```
SELECT AVG(COUNT(*)) FROM orders GROUP BY user_id;  -- Syntax error!
```

Right:

```
SELECT AVG(order_count)
FROM (
  SELECT user_id, COUNT(*) AS order_count
  FROM orders
  GROUP BY user_id
) user_orders;
```

How it works:

1. Inner query groups orders by user, computes count
2. Outer query averages those counts

Real-World GROUP BY Scenarios

Scenario 1: Top N per Group

Goal: Top 3 products per category by sales.

Without window functions:

```
SELECT *
FROM products p1
WHERE (
  SELECT COUNT(*)
  FROM products p2
  WHERE p2.category_id = p1.category_id
    AND p2.sales >= p1.sales
```

```
) <= 3
ORDER BY category_id, sales DESC;
```

Slow (correlated subquery).

With window functions:

```
SELECT category_id, product_name, sales
FROM (
  SELECT
    category_id,
    product_name,
    sales,
    ROW_NUMBER() OVER (PARTITION BY category_id ORDER BY sales DESC) AS rank
  FROM products
) ranked
WHERE rank <= 3;
```

Much faster.

Scenario 2: Conditional Aggregation

Goal: Count orders by status.

Naive approach (multiple queries):

```
SELECT COUNT(*) FROM orders WHERE status = 'pending';
SELECT COUNT(*) FROM orders WHERE status = 'completed';
```

Better (single query with CASE):

```
SELECT
  COUNT(CASE WHEN status = 'pending' THEN 1 END) AS pending_count,
  COUNT(CASE WHEN status = 'completed' THEN 1 END) AS completed_count
FROM orders;
```

Or with FILTER (Postgres):

```
SELECT
  COUNT(*) FILTER (WHERE status = 'pending') AS pending_count,
  COUNT(*) FILTER (WHERE status = 'completed') AS completed_count
FROM orders;
```

Cleaner and more performant.

Scenario 3: Running Totals (Without Window Functions)

Goal: Cumulative sales by date.

Old-school self-join:

```sql
SELECT
  s1.date,
  s1.amount,
  SUM(s2.amount) AS cumulative
FROM sales s1
JOIN sales s2 ON s2.date <= s1.date
GROUP BY s1.date, s1.amount
ORDER BY s1.date;
```

Cost: O(N^2) — don't do this.

With window functions:

```sql
SELECT
  date,
  amount,
  SUM(amount) OVER (ORDER BY date) AS cumulative
FROM sales;
```

Cost: O(N) — much better.

Scenario 4: Pivot Table (Row to Column)

Goal: Transform rows into columns.

Data:

```
product | quarter | sales
--------|---------|------
Widget  | Q1      | 100
Widget  | Q2      | 150
Gadget  | Q1      | 200
```

Goal:

```
product | Q1  | Q2
--------|-----|----
Widget  | 100 | 150
Gadget  | 200 | NULL
```

SQL:

```sql
SELECT
  product,
  SUM(CASE WHEN quarter = 'Q1' THEN sales END) AS Q1,
  SUM(CASE WHEN quarter = 'Q2' THEN sales END) AS Q2
FROM sales
GROUP BY product;
```

Postgres CROSSTAB: There's a `crosstab()` function, but the CASE approach is more portable.

Key Takeaways

1. **GROUP BY collapses rows.** You can only SELECT grouped columns or aggregates.
2. **WHERE filters before grouping; HAVING filters after.** Use each appropriately.
3. **COUNT(*) counts rows; COUNT(column) counts non-NULL values.** Different results if column is nullable.
4. **Aggregates ignore NULLs** (except COUNT(*)).
5. **Join explosions mess up aggregates.** Use DISTINCT or aggregate before joining.
6. **Postgres allows functional dependency.** MySQL (strict mode) requires all non-aggregated columns in GROUP BY.
7. **ROLLUP/CUBE generate subtotals.** Useful for reporting.
8. **Use window functions instead of self-joins** for running totals and ranking.

Next up: Indexes and performance—how to make your queries actually fast.