CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 7: Aggregation & Grouping

Amit Kumar Dhar

IIT Bhilai

Where we are

- We can filter, project, join, and combine sets.
- But we can't yet **summarize**: totals, averages, counts per group.
- Today: **aggregate functions**, **`GROUP BY`**, and **`HAVING`** — how SQL turns many rows into summary rows.
- We switch datasets: the **`store`** database (orders, products) makes aggregation vivid.

Agenda

- Aggregate functions & how they treat NULLs
- Aggregation over the whole table
- `GROUP BY`: one summary row per group
- The ”every non-aggregated column must be grouped” rule
- `HAVING` vs `WHERE`
- Grouping on expressions, multiple columns
- `GROUP BY` with joins
- `ROLLUP` / `GROUPING SETS` (preview)
- String aggregation

The store schema (Lab 3 dataset)

```
categories(cat_id, cat_name)
customers(cust_id, name, city, country, join_date)
products(prod_id, prod_name, cat_id, price, stock)
orders(order_id, cust_id, order_date, status)
order_items(order_id, prod_id, quantity, unit_price)
```

- Revenue of a line item = `quantity * unit_price`.

Aggregate functions

Function	Returns
`COUNT(*)`	number of rows
`COUNT(col)`	number of **non-NULL** values
`COUNT(DISTINCT col)`	number of distinct non-NULL values
`SUM(col)`	total
`AVG(col)`	mean of non-NULL values
`MIN(col)`, `MAX(col)`	extremes

An aggregate **collapses many rows into one value**.

Quick Check: Aggregate functions

Question: You have a table `employees(id, name, salary, bonus)`. Some employees have `NULL` as their bonus. Which of the following queries calculates the average bonus of employees who actually receive a bonus?

1. `SELECT AVG(bonus) FROM employees;`
2. `SELECT SUM(bonus) / COUNT(*) FROM employees;`

Quick Check: Aggregate functions

Question: You have a table `employees(id, name, salary, bonus)`. Some employees have `NULL` as their bonus. Which of the following queries calculates the average bonus of employees who actually receive a bonus?

1. `SELECT AVG(bonus) FROM employees;`
2. `SELECT SUM(bonus) / COUNT(*) FROM employees;`

Answer:

- Query 1 correctly calculates the average only over employees with a non-NULL bonus because `AVG()` ignores `NULL`s.
- Query 2 divides the total bonus by the total number of employees (including those with `NULL`), which might not be what you want.

Aggregation over the whole table

```sql
SELECT COUNT(*)          AS n_products,
       AVG(price)        AS avg_price,
       MIN(price)        AS cheapest,
       MAX(price)        AS priciest
FROM   products;
```

- No GROUP BY → the **entire table is one group** → exactly **one** result row.

COUNT: three flavors

```sql
SELECT COUNT(*)                 AS all_rows,       -- every row
       COUNT(status)            AS non_null_status,
       COUNT(DISTINCT status)   AS distinct_status
FROM   orders;
```

- `COUNT(*)` counts rows regardless of NULLs.
- `COUNT(col)` skips NULLs in that column.
- `COUNT(DISTINCT col)` counts unique non-NULL values.

NULLs and aggregates — the golden rule

- All aggregates except **`COUNT(*)`** ignore NULLs.
- `AVG(price)` = `SUM(price) / COUNT(price)` — divides by non-NULL count.
- **`SUM`** over zero non-NULL values is **`NULL`**, not 0.
- Guard with **`COALESCE`**:

```
SELECT COALESCE(SUM(quantity), 0) FROM order_items WHERE order_id = -1;
```

GROUP BY — summary per group

”Average price per category”:

```
SELECT cat_id, AVG(price) AS avg_price, COUNT(*) AS n
FROM   products
GROUP  BY cat_id;
```

- Rows are partitioned into groups by `cat_id`.
- Each aggregate is computed **within** each group.
- Output: **one row per group**.

Quick Check: GROUP BY

Question: Write a query to find the maximum `stock` level for each `cat_id` in the `products` table.

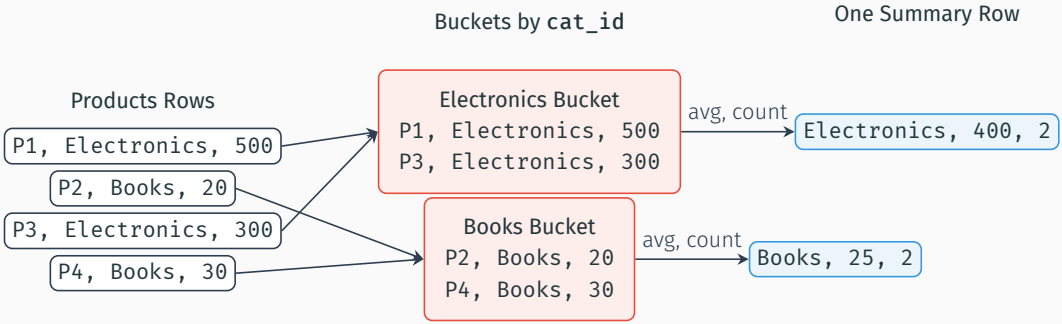
Quick Check: GROUP BY

Question: Write a query to find the maximum `stock` level for each `cat_id` in the `products` table.

Answer:

```sql
SELECT cat_id, MAX(stock) AS max_stock
FROM products
GROUP BY cat_id;
```

What GROUP BY does (mental model)



- GROUP BY = ”partition, then aggregate each partition.”

The cardinal rule of GROUP BY

> *Every column in the* `SELECT` *list must be* ***either*** *in the* `GROUP BY` ***or*** *inside an aggregate function.*

```sql
-- WRONG (in standard SQL / PostgreSQL):
-- prod_name is neither grouped nor aggregated
SELECT cat_id, prod_name, AVG(price) FROM products GROUP BY cat_id;
```

- PostgreSQL **rejects** this. SQLite/MySQL may accept it and return an **arbitrary** `prod_name` — a silent bug. Don't rely on it.

Grouping by multiple columns

”Revenue per customer per year”:

```sql
SELECT o.cust_id,
       strftime('%Y', o.order_date) AS yr,     -- SQLite
       SUM(oi.quantity * oi.unit_price) AS revenue
FROM   orders o
JOIN   order_items oi ON oi.order_id = o.order_id
GROUP  BY o.cust_id, strftime('%Y', o.order_date);
```

- A group is defined by the **combination** of grouping columns.

Grouping on an expression

- You can group by a computed value:

```
SELECT CASE WHEN price < 1000 THEN 'budget' ELSE 'premium' END AS tier,
       COUNT(*) AS n
FROM   products
GROUP  BY CASE WHEN price < 1000 THEN 'budget' ELSE 'premium' END;
```

- Repeat the expression (Postgres also allows `GROUP BY 1` by position, and the alias in some dialects). SQLite allows grouping by output alias.

HAVING — filter groups

- `WHERE` filters **rows**; `HAVING` filters **groups** (after aggregation).

”Categories with average price above 1500”:

```
SELECT cat_id, AVG(price) AS avg_price
FROM   products
GROUP  BY cat_id
HAVING AVG(price) > 1500;
```

- `HAVING` can reference aggregates; `WHERE` cannot.

Quick Check: HAVING

Question: Which query correctly finds categories that have more than 5 products?

1. `SELECT cat_id FROM products WHERE COUNT(*) > 5 GROUP BY cat_id;`
2. `SELECT cat_id FROM products GROUP BY cat_id HAVING COUNT(*) > 5;`

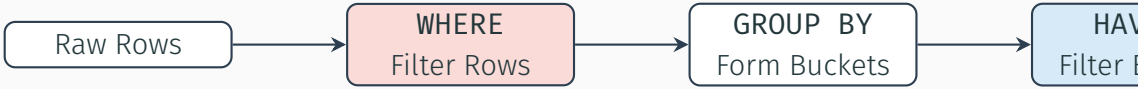
Quick Check: HAVING

Question: Which query correctly finds categories that have more than 5 products?

1. `SELECT cat_id FROM products WHERE COUNT(*) > 5 GROUP BY cat_id;`
2. `SELECT cat_id FROM products GROUP BY cat_id HAVING COUNT(*) > 5;`

Answer: Query 2. `WHERE` evaluates row-by-row before grouping, so it cannot contain aggregate functions like `COUNT(*)`. `HAVING` evaluates after grouping.

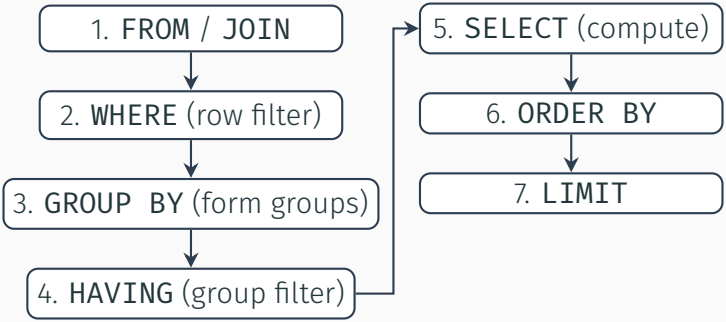
WHERE vs HAVING — the difference




```sql
SELECT cat_id, COUNT(*) AS n_expensive
FROM   products
WHERE  price > 1000          -- 1) drop cheap products BEFORE grouping
GROUP  BY cat_id
HAVING COUNT(*) >= 2;        -- 2) keep groups with >= 2 such products
```

- WHERE runs **before** GROUP BY (per-row).
- HAVING runs **after** GROUP BY (per-group).
- Use WHERE for row conditions (it's cheaper — fewer rows to group).

Full clause evaluation order



- This order explains many ”why can’t I use that alias here?” errors.

Why SELECT aliases fail in WHERE/GROUP BY

```
SELECT quantity * unit_price AS line_total
FROM   order_items
WHERE  line_total > 5000;        -- ERROR in PostgreSQL: line_total not known y
```

- `SELECT` is computed **after** `WHERE`/`GROUP BY`, so the alias doesn't exist yet. Repeat the expression, or use a subquery/CTE (Lecture 8).
- `ORDER BY` *can* use `SELECT` aliases (it runs last).

Aggregates with joins — revenue per category

```sql
SELECT c.cat_name,
       SUM(oi.quantity * oi.unit_price) AS revenue
FROM   categories c
JOIN   products p     ON p.cat_id = c.cat_id
JOIN   order_items oi ON oi.prod_id = p.prod_id
GROUP  BY c.cat_name
ORDER  BY revenue DESC;
```

- Join first to bring columns together, then group & aggregate.

Careful: joins can distort aggregates

- Joining to a one-to-many table **multiplies** rows before aggregation.
- Summing a column from the ”one” side after such a join **double-counts**.
- Example: `SUM(o.some_order_total)` after joining `order_items` counts the order total once per item.
- Fix: aggregate the many-side in a **subquery** first (Lecture 8), or sum the correct per-row quantity (`quantity * unit_price`).

COUNT with LEFT JOIN — including zeros

```sql
SELECT c.name, COUNT(o.order_id) AS n_orders
FROM   customers c
LEFT JOIN orders o ON o.cust_id = c.cust_id
GROUP  BY c.cust_id, c.name
ORDER  BY n_orders;
```

- `COUNT(o.order_id)` yields `0` for customers with no orders (NULLs ignored).
- `COUNT(*)` would wrongly return 1 for them — pick the column deliberately.

Top-N per aggregate

”Top 3 products by units sold”:

```
SELECT p.prod_name, SUM(oi.quantity) AS units
FROM   products p
JOIN   order_items oi ON oi.prod_id = p.prod_id
GROUP  BY p.prod_id, p.prod_name
ORDER  BY units DESC
LIMIT  3;
```

- Group → aggregate → order → limit. (Top-N **per group** needs window functions — Lecture 9.)

DISTINCT inside aggregates

```sql
-- how many distinct products has each customer bought?
SELECT o.cust_id, COUNT(DISTINCT oi.prod_id) AS distinct_products
FROM   orders o
JOIN   order_items oi ON oi.order_id = o.order_id
GROUP  BY o.cust_id;
```

- `COUNT(DISTINCT ...)` de-duplicates within the group before counting.

Filtering with aggregates: HAVING examples

```sql
-- customers who have spent more than 20,000 in total
SELECT o.cust_id, SUM(oi.quantity * oi.unit_price) AS spent
FROM   orders o JOIN order_items oi ON oi.order_id = o.order_id
GROUP  BY o.cust_id
HAVING SUM(oi.quantity * oi.unit_price) > 20000;
```

- The `HAVING` predicate uses the same aggregate as (or a different one from) the `SELECT`.

ROLLUP & GROUPING SETS (preview)

- Want subtotals **and** a grand total in one query?

```
-- PostgreSQL
SELECT cat_id, SUM(stock)
FROM   products
GROUP  BY ROLLUP (cat_id);      -- per-category rows + a grand-total row (cat_i
```

- `ROLLUP`, `CUBE`, `GROUPING SETS` build multi-level summaries.
- Postgres supports them; SQLite does not (emulate with `UNION ALL`).

Quick Check: ROLLUP

Question: What extra row does `ROLLUP(cat_id)` add to the result set that a regular `GROUP BY cat_id` does not?

Quick Check: ROLLUP

Question: What extra row does `ROLLUP(cat_id)` add to the result set that a regular `GROUP BY cat_id` does not?

Answer: It adds a ”grand total” row where `cat_id` is `NULL`, and the aggregate function (like `SUM(stock)`) contains the sum across all categories combined.

String aggregation

- Concatenate values within a group into one string:

```
-- PostgreSQL
SELECT cat_id, STRING_AGG(prod_name, ', ' ORDER BY prod_name) FROM products GRO
-- SQLite
SELECT cat_id, GROUP_CONCAT(prod_name, ', ') FROM products GROUP BY cat_id;
```

- Useful for ”list all products in each category” on a single row.

Summary — key takeaways

- Aggregates collapse rows; **all but `COUNT(*)` ignore `NULL`s**.
- `GROUP BY` → one row per group; every selected column is grouped or aggregated.
- `WHERE` filters rows **before** grouping; `HAVING` filters groups **after**.
- Watch join-induced **row multiplication** distorting sums/counts.
- `ROLLUP`/`GROUPING SETS` add subtotals (Postgres).

Exercises (on `store.sql`)

1. Number of products and average price **per category** (show category name).
2. Total revenue per **country** of the customer.
3. Customers who placed **more than 3** orders.
4. For each product, total units sold; include products never sold (0).
5. Months (YYYY-MM) whose total revenue exceeded 15,000.

Before next lecture

- Load `store.sql` into SQLite and try all five exercises.
- Optional reading: Silberschatz §3.7 (aggregation).

Next time — Lecture 8: subqueries & CTEs — nesting queries, `EXISTS`/`IN`, correlated subqueries, and `WITH`.

Any Questions?