GROUP BY & HAVING — theory, correct logic, and code points (deep but readable)

Great — here's a focused, no-fluff deep-dive into **GROUP BY** and **HAVING**: exact logic, execution order, syntax templates, common pitfalls, best example(s) (using your employees table), and practical tips you'll actually use in interviews and real work.

---

1 — Big idea (one-line summary)

- **GROUP BY**: groups rows into buckets so you can compute aggregates per bucket.
- **HAVING**: filters the groups created by GROUP BY (it's the "WHERE for groups").

---

2 — SQL logical execution order (critical to understand)

This order explains *when* filtering happens and why WHERE vs HAVING behave differently:

1. **FROM** (tables joined, rows produced)
2. **WHERE** (row-level filtering)
3. **GROUP BY** (rows grouped into groups/buckets)
4. **HAVING** (group-level filtering)
5. **SELECT** (expressions/aggregates evaluated; note: SELECT can reference group aggregates)
6. **ORDER BY** (final ordering)
7. **LIMIT / TOP** (final row limiting)

Because of this order:

- WHERE cannot use aggregate functions (it runs before grouping).
- HAVING can use aggregate functions (it runs after grouping).

---

3 — Basic syntax templates

Grouping with aggregates

```
SELECT <group_cols>, AGG_FUNC(<col>) AS agg_alias
FROM table
WHERE <row_conditions>      -- optional
GROUP BY <group_cols>
HAVING <group_conditions>   -- optional, can use aggregates
ORDER BY <expr>             -- optional (can order by agg_alias in many DBs)
LIMIT n;                    -- or TOP n depending on DB
```

Examples of aggregate functions

COUNT(*), COUNT(col), SUM(col), AVG(col), MIN(col), MAX(col)

---

4 — Precise rules & code points (what's allowed/required)

- **Every non-aggregated column in SELECT must appear in GROUP BY** (ANSI SQL). Example wrong:
- SELECT city, salary FROM employees GROUP BY city; -- error: salary neither grouped nor aggregated
  Correct:
  SELECT city, SUM(salary) AS total_pay FROM employees GROUP BY city;
- **You can GROUP BY expressions**, not only base columns:
- SELECT YEAR(joining_date) AS yr, COUNT(*) FROM employees GROUP BY YEAR(joining_date);
- **HAVING can reference aggregates**:
- HAVING COUNT(*) > 10
- HAVING AVG(salary) > 50000
- **Prefer WHERE for row-level filtering** — it's faster and reduces work:
- WHERE joining_date >= '2020-01-01' -- filters rows before grouping
- **HAVING for group-level constraints**:
- HAVING COUNT(*) >= 5 -- filter out small groups
- **Aliases in HAVING and ORDER BY**: many DBs allow using agg_alias in ORDER BY; use of alias in HAVING is less portable — safer to repeat aggregate expression or use a subquery/CTE:
- -- Safer: use the aggregate expression
- HAVING SUM(salary) > 100000
- -- Or use CTE:
- WITH dept_sums AS (
- SELECT department, SUM(salary) AS total_sal FROM employees GROUP BY department
-)
- SELECT * FROM dept_sums WHERE total_sal > 100000;
- **NULLs in GROUP BY**: NULL values are grouped together (all NULLs in a column form one group).

---

5 — Common patterns & must-know idioms (with code)

5.1 Count per group

```
SELECT city, COUNT(*) AS cnt
FROM employees
GROUP BY city;
```

5.2 Filter groups with HAVING

```
SELECT city, COUNT(*) AS cnt
FROM employees
GROUP BY city
HAVING COUNT(*) > 10;     -- only cities with > 10 employees
```

5.3 Aggregate multiple metrics

```
SELECT department,
    COUNT(*) AS emp_count,
```

```
    SUM(salary) AS total_salary,
    AVG(salary) AS avg_salary,
    MAX(salary) AS max_salary
FROM employees
GROUP BY department;
```

5.4 WHERE + GROUP BY + HAVING together (best practice)

Filter rows first with WHERE, group the remainder, then filter groups:

```
SELECT department, COUNT(*) AS emp_count, AVG(age) AS avg_age
FROM employees
WHERE joining_date >= '2020-01-01'   -- row-level pre-filter
GROUP BY department
HAVING COUNT(*) >= 5                 -- group-level filter
ORDER BY avg_age DESC;
```

5.5 GROUP BY multiple columns (compound groups)

```
SELECT city, department, COUNT(*) AS cnt
FROM employees
GROUP BY city, department;
```

Each unique (city, department) pair is one group.

5.6 Grouping on expressions

```
SELECT CASE
       WHEN salary >= 100000 THEN 'High'
       WHEN salary >= 50000  THEN 'Medium'
       ELSE 'Low'
     END AS salary_band,
     COUNT(*) AS cnt
FROM employees
GROUP BY CASE
        WHEN salary >= 100000 THEN 'High'
        WHEN salary >= 50000  THEN 'Medium'
        ELSE 'Low'
      END;
```

---

6 — Best example (full walk-through using **employees**)

Problem: For each department, compute number of employees, total salary, and average age — but only show departments that (a) have at least 3 employees, and (b) average age > 30. Order by total salary desc.

Code

```
SELECT department,
    COUNT(*)        AS emp_count,
    SUM(salary)     AS total_salary,
    ROUND(AVG(age),1) AS avg_age
FROM employees
WHERE department IS NOT NULL        -- row-level filter
GROUP BY department
HAVING COUNT(*) >= 3 AND AVG(age) > 30
ORDER BY total_salary DESC;
```

Step-by-step execution

1. FROM: read employees table.
2. WHERE: drop rows where department is NULL.
3. GROUP BY: create one group per department.
4. SELECT (aggregates evaluated per group): COUNT, SUM, AVG computed for each department.
5. HAVING: keep only those groups where COUNT >= 3 AND AVG(age) > 30.
6. ORDER BY: sort remaining groups by total_salary descending.
7. Return result set.

Why WHERE first? Because removing null departments early reduces groups and speeds aggregation.

---

7 — Common mistakes (and how to avoid them)

1. **Using WHERE with aggregates**
   ❌ WHERE COUNT(*) > 1 — wrong. Use HAVING.
   ✅ HAVING COUNT(*) > 1
2. **Selecting non-grouped, non-aggregated columns**
   ❌ SELECT department, full_name, SUM(salary) FROM employees GROUP BY department;
   ✅ include full_name only inside an aggregate or group by it.
3. **Relying on alias in HAVING (portability issue)**
   Some DBs accept HAVING total_salary > 10000 if total_salary is alias; others don't. Safer to use the aggregate or a CTE.
4. **Forgetting NULLs**
   Remember groups can be NULL — use WHERE column IS NOT NULL when you intentionally want to exclude NULL groups.
5. **Using functions on grouping columns (hurts index use)**
   GROUP BY UPPER(city) prevents index usage on city. If possible, store normalized values or do the transformation in a computed column.

---

8 — Performance tips (practical)

- **Filter early**: always push row-level filters into WHERE.
- **Index grouping columns** (helps sort/aggregate — DB-specific): e.g., index on department if you GROUP BY department a lot.
- **Avoid SELECT *** with GROUP BY; select only needed columns.
- **Use approximate aggregation** for very large datasets if exact accuracy is not needed (DB-dependent features).
- **Use CTEs/derived tables** to pre-aggregate complex subqueries for clarity and sometimes performance.

---

10 — Quick cheatsheet (copy-paste)

Must-use pattern

```
-- filter rows first, then group, then filter groups
SELECT group_col, AGG(col) AS alias
FROM table
WHERE <row-level conditions>
GROUP BY group_col
HAVING <group-level conditions using aggregates>
ORDER BY alias DESC;
```

Quick do/don't

- Do: `WHERE` for rows, `HAVING` for groups.
- Don't: `WHERE COUNT(*) > 1` — use HAVING.
- Do: `GROUP BY` non-aggregated select columns.
- Don't: apply unnecessary functions on grouping columns (index loss).

---

11 — Final quick interview-ready talking points

- "GROUP BY groups rows; HAVING filters groups."
- "Execution order: WHERE → GROUP BY → HAVING → SELECT → ORDER BY."
- "Always filter with WHERE when you can — it reduces data before aggregation."
- "HAVING is necessary when your filter references aggregates (COUNT, SUM, AVG…)."
- Give the best example above (department aggregates with HAVING) when asked to demonstrate.

📌 3. Deep understanding (45 min) Focus on concept clarity: • Why SELECT cannot come before aggregate • Why WHERE cannot use aggregate (COUNT, SUM cannot be used inside WHERE) • Why aggregate returns only one row • What happens when table has NULL values

1. Why SELECT cannot come before aggregate?

👉 Because SQL needs the data first, then it can calculate totals.

Think like cooking:

- First bring all vegetables (raw data)
- Then cut and cook (aggregate)
- Not possible to **cook first** before getting the vegetables.

In SQL:

Before it can **SUM(), COUNT(), AVG()**, SQL must fetch the rows.

So SQL internally works like this:

1. FROM → pick the rows
2. WHERE → filter rows
3. GROUP BY → make groups
4. AGGREGATE → calculate SUM/COUNT
5. SELECT → finally show result

In-line explanation (super simple):

1. **FROM ShoppingList** →
   SQL first **picks all items** from the table (Apple, Banana, Carrot, Potato).
2. **WHERE Price > 20** →
   SQL **removes items that don't satisfy the condition**
   (Banana is removed because its price is 20).
   Remaining: Apple, Carrot, Potato.
3. **GROUP BY Category** →
   SQL **creates groups**:
   - Fruit → Apple
   - Vegetable → Carrot, Potato
4. **SUM(Price)** (Aggregate) →
   SQL **adds prices inside each group**:
   - Fruit = 50
   - Vegetable = 30 + 40 = 70
5. **SELECT** →
   SQL **shows the final result**:
   (Fruit, 50) and (Vegetable, 70)

🔻 **That's why SELECT cannot come before aggregates.**
SQL must **finish the calculation first**, then SELECT displays it.

---

2. Why WHERE cannot use aggregates like COUNT(), SUM()?

Very simple:

WHERE works before aggregation happens.
Aggregates happen later.
Analogy:

- You cannot check the total marks **before** adding them.
- WHERE is checking each row **one-by-one**, not totals.

❓ Wrong moment: WHERE tries to use SUM()
✔ But SUM() is not ready yet — SUM() happens after grouping.

Correct logic:

- WHERE filters individual rows.
- HAVING filters aggregated results.

Example:
You want to show only those departments where total salary > 1,00,000
❓ This won't work:

```
WHERE SUM(salary) > 100000
```

✔ But HAVING works, because HAVING runs **after SUM()**:

```
HAVING SUM(salary) > 100000
```

---

3. Why aggregates return only one row (without GROUP BY)?

Very simple:
When you say:
SELECT COUNT(*) FROM employees;
You are asking:
☞ "Give me **one value**: total number of employees."
SQL combines all rows into **one result**.
Just like:

- Total marks → one number
- Total salary → one number
- Average age → one number

So…

- No GROUP BY → SQL treats the **whole table as one group**
- That's why you get **only 1 row**.

---

4. What happens when the table has NULL values?

Very simple:
Aggregates ignore NULL values.
Examples:
Salary
1000
2000
NULL

COUNT()

COUNT(salary)
Ignores NULL → result = 2

SUM()

SUM(salary)
Ignores NULL → sum = 3000

AVG()

AVG(salary)
Ignores NULL → (1000 + 2000) / 2 = 1500
(division by 2, not 3)

MIN() and MAX()

Also ignore NULL.

🕵 Exception:

COUNT(*)

Counts ALL rows including NULL → result = 3