Aggregation and GROUP BY

COUNT/SUM/AVG/MIN/MAX, GROUP BY, and HAVING -- entirely new material with no equivalent in Spring Data JPA's Pageable/Sort/projection vocabulary -- with this project's own real "topics per category" LEFT JOIN + GROUP BY query. The 10th and FINAL lesson in the PostgreSQL Foundations category.

Intermediate 20 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/aggregation-and-group-by

---

Every query so far has returned one row per row of underlying data — a `WHERE` might narrow which rows show up, but the row count in equals the row count out. This lesson is the first place that changes: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, and `GROUP BY` collapse many rows into one summary row, or one summary row per group — nothing in Spring Data JPA's own `Pageable` / `Sort` /projection vocabulary, already covered in the Spring Data JPA course, does anything like this; it's genuinely new material.

Aggregate Functions: COUNT, SUM, AVG, MIN, MAX

An aggregate function takes many rows and returns one value computed across all of them:

```
SELECT COUNT(*) FROM topic;
SELECT AVG(estimated_minutes) FROM topic;
SELECT MIN(estimated_minutes), MAX(estimated_minutes) FROM topic;
```

`COUNT(*)` counts rows, regardless of any column's value; `COUNT(column)` counts only rows where that column is not `NULL` — a real distinction on this project's own nullable `estimated_minutes` column, where `COUNT(*)` and `COUNT(estimated_minutes)` could genuinely differ if any row had it unset. `AVG` / `SUM` / `MIN` / `MAX` all operate on a single numeric (or, for `MIN` / `MAX`, orderable) column, ignoring `NULL`s automatically rather than letting one `NULL` poison the whole calculation.

Aggregating Without GROUP BY: One Row for the Whole Table

Every example above returns exactly one row, no matter how many rows the table has — an aggregate function with no `GROUP BY` treats the entire result of the `FROM` / `WHERE` clauses as a single group. Adding a `WHERE` narrows which rows feed the aggregate, same as any other query:

```
SELECT COUNT(*) FROM topic t
JOIN category cat ON t.category_id = cat.id
WHERE cat.slug = 'postgresql-foundations';
```

still one row back — the count of `postgresql-foundations`'s own topics specifically, not the whole `topic` table (using a join on `slug` rather than a hardcoded numeric id, the same reasoning "Inserting, Updating, and Deleting Data"'s `INSERT ... SELECT` pattern already established for not hardcoding ids).

GROUP BY: One Row Per Group

`GROUP BY` changes an aggregate from "one row total" to "one row per distinct value" of whatever it groups on:

```
SELECT difficulty, COUNT(*)
FROM topic
GROUP BY difficulty;
```

Run against this project's own real `topic` table, this returns one row per `Difficulty` value actually present — `BEGINNER`, `INTERMEDIATE`, `ADVANCED` — each paired with how many topics currently have that difficulty. Every column in the `SELECT` list must be either an aggregate function or one of the columns named in `GROUP BY` — `SELECT slug, difficulty, COUNT(*) ... GROUP BY difficulty` would be rejected, because for a given `difficulty` group PostgreSQL has no single `slug` to report; there could be many.

A Real Example: How Many Topics Per Category

```
SELECT cat.name AS category_name, COUNT(t.id) AS topic_count
FROM category cat
LEFT JOIN topic t ON t.category_id = cat.id
GROUP BY cat.name;
```

This combines "JOINs" (a `LEFT JOIN`, chosen deliberately so a category with zero topics still shows up with a count of `0`, rather than being silently dropped the way an `INNER JOIN` would drop it) with `GROUP BY` in one query — grouping by category, counting how many

`topic` rows joined to each one. Run against this project's own real data, `PostgreSQL Foundations` currently shows a count of `10`, this very lesson included. `COUNT(t.id)` rather than `COUNT(*)` matters here specifically: with the `LEFT JOIN`, a category with no matching topics still produces one output row where every `t.*` column is `NULL` — `COUNT(*)` would count that row as `1`, while `COUNT(t.id)` correctly counts it as `0`, since `t.id` itself is `NULL` for it.

Multiple Aggregates in One Query

Any number of aggregate functions can appear together, each computed per group:

```
SELECT difficulty, COUNT(*) AS topic_count, AVG(estimated_minutes) AS avg_minutes
FROM topic
GROUP BY difficulty;
```

One row per `difficulty`, with both a count and an average computed independently within that group — run against this project's real `topic` table as a whole (spanning every course, not just this one), it's a genuine way to check whether `ADVANCED` topics (like "The Persistence Context and Locking," back in the Spring Data JPA course) tend to carry a higher `avg_minutes` than `BEGINNER` ones — a question no single-row query could answer.

HAVING: Filtering Groups, Not Rows

`WHERE` filters individual rows *before* grouping happens; `HAVING` filters entire groups *after* aggregation, based on the aggregate's own result:

```
SELECT category_id, COUNT(*) AS topic_count
FROM topic
GROUP BY category_id
HAVING COUNT(*) >= 9;
```

This keeps only categories with `9` or more topics — a condition that's meaningless to express in `WHERE`, since `COUNT(*)` doesn't exist yet for any single row being filtered; it only exists once the grouping has already happened. `HAVING` can reference an aggregate directly (as above) or an alias defined in the `SELECT` list, depending on the exact query — but never a raw, non-aggregated, non-grouped column, for the same reason `SELECT` can't.

WHERE vs. HAVING: When Each Runs

The two clauses run at genuinely different stages of the same query, and mixing them up which does which is one of the most common `GROUP BY` mistakes:

```
SELECT category_id, COUNT(*) AS topic_count
FROM topic
WHERE difficulty = 'ADVANCED'
GROUP BY category_id
HAVING COUNT(*) >= 2;
```

`WHERE difficulty = 'ADVANCED'` runs first, discarding non-`ADVANCED` rows entirely before any grouping happens; `GROUP BY category_id` then groups whatever rows survived; `HAVING COUNT(*) >= 2` runs last, keeping only the resulting groups whose count (of already-filtered rows) meets the threshold. `WHERE` narrows the rows an aggregate sees; `HAVING` narrows which computed groups make it into the final result — the same distinction "SELECT and Filtering"'s `WHERE` coverage never needed to draw, because that lesson had no grouping stage at all.

Common Misconceptions

"`COUNT(*)` and `COUNT(column)` always return the same number." Only when that column has no `NULL`s — this project's own nullable `estimated_minutes` is a real column where they could genuinely diverge. **"`WHERE` can filter on an aggregate like `COUNT(*)`."** It can't — `WHERE` runs before aggregation exists, which is exactly why `HAVING` is a separate clause rather than an extra condition tacked onto `WHERE`. **"Every column selected alongside `GROUP BY` needs to be listed in the `GROUP BY` clause, or it's a bug in the query."** It's not a bug — it's a rule PostgreSQL actively enforces: a non-aggregated, non-grouped column simply has no single well-defined value per group, so PostgreSQL rejects the query outright rather than picking one arbitrarily.

Best Practices

- Choose `COUNT(t.id)` over `COUNT(*)` specifically after a `LEFT JOIN` when zero-count groups matter — this project's own "topics per category" query depends on exactly this distinction to correctly report `0` instead of `1` for an empty category.
- Reach for `HAVING` only when the condition genuinely depends on an aggregate's result (a count, a sum) — a condition on a plain column belongs in `WHERE`, which runs earlier and lets PostgreSQL discard non-matching rows before doing any grouping work at all.
- List every non-aggregated `SELECT` column in `GROUP BY` deliberately, not just to satisfy PostgreSQL's rule — each one changes what "a group" actually means, potentially producing far more groups than intended.
- Compute multiple aggregates together in one query (as in the `COUNT` + `AVG` example) rather than issuing one query per aggregate — each is computed from the same grouped rows in a single pass.

Common Mistakes

- Writing an aggregate condition into `WHERE` instead of `HAVING` (`WHERE COUNT(*) >= 5`) and getting a genuine SQL error, not a wrong answer — a useful signal, once recognized, of exactly which clause was needed.
- Selecting a non-aggregated, non-grouped column and being confused by PostgreSQL's rejection, rather than recognizing it as the query genuinely being ambiguous about which row's value to report for that column within each group.
- Using `COUNT(*)` after a `LEFT JOIN` when the real intent was "how many matching rows," and getting `1` instead of `0` for groups with no match, because `COUNT(*)` counts the single all-`NULL` row a `LEFT JOIN` produces for an unmatched left-hand row.
- Assuming `GROUP BY` and `DISTINCT` do the same job because both can collapse duplicate-looking rows — `DISTINCT` only removes exact row duplicates from a result set; it can't compute a count, sum, or average per group the way `GROUP BY` genuinely can.

Summary, Cheat Sheet, and Glossary

Summary

- Aggregate functions (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) compute one value across many rows; without `GROUP BY`, the whole filtered result is treated as a single group.
- `GROUP BY <column>` produces one row per distinct value of that column; every non-aggregated `SELECT` column must appear in `GROUP BY`.
- This project's own real "topics per category" query combines a `LEFT JOIN` with `GROUP BY` and `COUNT(t.id)` (not `COUNT(*)`) so empty categories correctly show `0`.
- `HAVING` filters groups after aggregation, based on the aggregate's own result; `WHERE` filters individual rows before grouping happens — the two run at genuinely different stages of the same query.
- `GROUP BY` has no equivalent anywhere in Spring Data JPA's `Pageable`/`Sort`/projection vocabulary already covered in this curriculum — this lesson is entirely new material, not a raw-SQL restatement of something already taught.

Cheat Sheet

```
SELECT COUNT(*) FROM t;
SELECT COUNT(col) FROM t;            -- ignores NULLs
SELECT AVG(col), MIN(col), MAX(col) FROM t;

SELECT g, COUNT(*) FROM t GROUP BY g;

SELECT g, COUNT(*) FROM t
GROUP BY g
HAVING COUNT(*) >= 5;

SELECT g, COUNT(*) FROM t
WHERE col = 'x'
GROUP BY g
HAVING COUNT(*) >= 5;
```

Glossary

- **Aggregate function**: a function (`COUNT` , `SUM` , `AVG` , `MIN` , `MAX`) that computes one value from many rows.
- **GROUP BY**: a clause that produces one result row per distinct value (or combination of values) of the columns it names.
- **HAVING**: a clause that filters groups after aggregation, based on an aggregate's result — the `WHERE` of a grouped query.
- **Group**: the set of rows sharing the same value(s) in the `GROUP BY` columns, collapsed into one output row.