JOINs

INNER/LEFT/RIGHT/FULL JOIN, with this project's own real topic→category→course chain and its real TopicRepository.findBySlugWithCategoryAndCourse JPQL join fetch -- plus a real LEFT JOIN example finding topics not yet published in English. The 9th lesson in the PostgreSQL Foundations category.

Intermediate 25 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/joins

Every query so far in this course has read from one table at a time. This project's own real domain — `course`, `category`, `topic`, `topic_translation` — is deliberately split across four tables, exactly the way "PostgreSQL and the Relational Model" first laid it out, and a `JOIN` is how a single query pulls related rows back together across that split. Spring Data JPA's own `@OneToMany` / `join fetch` JPQL, from "Relationships, Fetching, and the N+1 Problem," solves a related problem one layer up — this lesson is about the raw SQL underneath.

Why JOINs: Data Split Across Tables

A `category` row only stores a `course_id` — it doesn't repeat the course's `name`. Getting a category alongside its course's name in one result therefore means combining two tables, row by row, wherever their foreign key relationship connects them. That combining operation is a `JOIN` — nothing more exotic than matching rows from two tables on a condition, almost always a foreign key equality.

INNER JOIN: Only Matching Rows

```
SELECT c.name AS course_name, cat.name AS category_name
FROM category cat
INNER JOIN course c ON cat.course_id = c.id;
```

`INNER JOIN` (often just written `JOIN`, with `INNER` implied) returns only rows that have a match on both sides — a `category` row with no matching `course` (which, thanks to the `NOT NULL REFERENCES` "Constraints and Keys" already covered, can never actually happen here) simply wouldn't appear. `ON` specifies the join condition, almost always equating a foreign key to the primary key it references; `AS` gives each table a short alias (`c`, `cat`) so columns from either side can be referenced unambiguously, especially when — as here — both tables happen to have a column literally named `name`.

This Project's Own topic → category → course Chain, as Raw SQL

Extending the same pattern across all three levels of this project's real content hierarchy:

```
SELECT t.slug, cat.name AS category_name, c.name AS course_name
FROM topic t
INNER JOIN category cat ON t.category_id = cat.id
INNER JOIN course c ON cat.course_id = c.id
WHERE t.slug = 'joins';
```

Run against this project's own real data, this returns exactly one row: `joins`, `PostgreSQL Foundations`, `PostgreSQL` — every piece of this lesson's own place in the course hierarchy, pulled from three separate tables in a single query. Each `INNER JOIN` adds one more table to the combination; the order they're chained in doesn't change the result, only (in principle) how PostgreSQL might choose to execute it internally.

From JPQL join fetch to a Real SQL JOIN

This project's own `TopicRepository` has a real method that already does exactly this, one layer up:

```
@Query("select t from Topic t join fetch t.category c join fetch c.course where t.slug = :
slug")
Optional<Topic> findBySlugWithCategoryAndCourse(String slug);
```

"Relationships, Fetching, and the N+1 Problem" already covered `join fetch` as the technique that avoids issuing separate follow-up queries for `t.category` and `c.course` — what's worth naming explicitly here, at the SQL level, is that this JPQL compiles to essentially the same three-table `INNER JOIN` chain written by hand above. `join fetch` isn't a different kind of join from SQL's `JOIN` — it's Hibernate choosing to express a Java-level "also load this related entity" instruction as a real SQL `JOIN`, instead of a second round-trip query.

LEFT JOIN: Keeping Unmatched Rows Too

`LEFT JOIN` (also written `LEFT OUTER JOIN`) keeps every row from the left-hand table, whether or not it finds a match on the right — when there's no match, the right side's columns simply come back as `NULL`:

```
SELECT t.slug, tt.title
FROM topic t
LEFT JOIN topic_translation tt ON tt.topic_id = t.id AND tt.language = 'en';
```

Every `topic` row appears at least once here, even one with no English `topic_translation` row at all — `tt.title` would simply be `NULL` for it, rather than that `topic` silently disappearing from the results the way an `INNER JOIN` would make it. This is the genuine difference between the two: `INNER JOIN` answers "rows that have a match"; `LEFT JOIN` answers "every row on the left, plus a match where one exists."

A Real LEFT JOIN: Finding Topics Not Yet Published in English

This project's own two-step publish workflow — a topic's Turkish translation published immediately, its English translation following later, as CLAUDE.md documents and this very course's own migrations (`connecting-to-postgresql`'s `V402` then `V403`) demonstrate — is exactly the kind of situation a `LEFT JOIN` is built for:

```
SELECT t.slug
FROM topic t
LEFT JOIN topic_translation en
    ON en.topic_id = t.id AND en.language = 'en' AND en.published = true
WHERE en.id IS NULL;
```

This returns every `topic` slug that has no published English translation yet — a real, useful query against this project's own schema, not a hypothetical one. The `en.id IS NULL` check at the end is the standard idiom for "find rows on the left with no match on the right" — it works precisely because a genuinely unmatched row's right-hand columns come back `NULL`, the same `NULL`-means-missing behavior "SELECT and Filtering" already covered for `IS NULL` generally.

RIGHT JOIN and FULL JOIN

`RIGHT JOIN` is the mirror image of `LEFT JOIN` — it keeps every row from the right-hand table instead of the left. It's rarely needed in practice (and this project's own code never uses it) because swapping the two tables' order in a `LEFT JOIN` produces the identical result — `A LEFT JOIN B` and `B RIGHT JOIN A` return the same rows, just with columns in a different order. `FULL JOIN` (or `FULL OUTER JOIN`) keeps every row from *both* sides regardless of match, filling in `NULL` on whichever side has no counterpart — useful for genuinely symmetric comparisons (like finding every mismatch between two tables that should mirror each other), a case this project's own domain doesn't currently need.

Common Misconceptions

"A JOIN merges two tables into one, permanently." It doesn't — a `JOIN` combines rows for the duration of a single query's result set; the underlying tables (`category`, `course`) remain exactly as separate as "Databases, Schemas, Tables, and Basic SQL Syntax" first showed them. **"`LEFT JOIN` is always slower than `INNER JOIN`."** Not inherently — the performance difference, if any, comes down to indexes and row counts, covered properly in

"Indexes and Query Performance with EXPLAIN," not the join type itself. **"`join fetch` in JPQL is a completely different concept from a SQL JOIN."** It isn't — it's the exact same relational operation, just triggered from the Java/JPQL side instead of written by hand.

Best Practices

- Alias every table in a multi-table query (`t`, `cat`, `c`, as used throughout this lesson) — it keeps column references unambiguous the moment two tables share a column name, like `name` on both `category` and `course`.
- Reach for `LEFT JOIN ... WHERE <right>.id IS NULL` whenever the real question is "which rows are missing a related row" — this project's own "topics without a published English translation" query is a genuine, reusable example of the pattern.
- Default to `INNER JOIN` unless there's a specific reason rows without a match need to survive in the result — it's both the more common case and the easier one to reason about.
- Recognize a JPQL `join fetch` (like this project's own `findBySlugWithCategoryAndCourse`) as literally compiling to a SQL `JOIN` — reasoning about its cost or behavior means reasoning about the same `JOIN` mechanics covered in this lesson, not a separate Java-level concept.

Common Mistakes

- Forgetting the `ON` condition entirely, or writing one that doesn't actually reference a shared key — PostgreSQL won't refuse to run it, but a join with a wrong or missing condition can silently multiply row counts (a cross product) rather than erroring out.
- Using `INNER JOIN` when the real intent was "keep this row even if there's no match" — this project's own English-translation query, written as an `INNER JOIN`, would silently exclude exactly the topics it was meant to find, since an unmatched `topic` row simply wouldn't appear at all.
- Filtering an outer-joined table's column in `WHERE` instead of in the `ON` clause when the intent was to keep unmatched rows — `WHERE en.published = true` after a `LEFT JOIN` silently turns it back into the equivalent of an `INNER JOIN`, since `WHERE` runs after the join and drops any row where that condition isn't true, including the `NULL` rows the `LEFT JOIN` was meant to preserve.
- Assuming column order in a `SELECT` after a multi-table `JOIN` reflects which table a column "belongs to" without checking the alias — ambiguous or misattributed column references are a common source of a query that runs, but reads the wrong value.

Summary, Cheat Sheet, and Glossary

Summary

- A `JOIN` combines rows from two or more tables on a condition — almost always a foreign key equality — for the duration of one query; the underlying tables stay separate.
- `INNER JOIN` returns only rows with a match on both sides; this project's own `topic` → `category` → `course` chain, joined three tables deep, is a real example.
- This project's own `TopicRepository.findBySlugWithCategoryAndCourse`'s JPQL `join fetch` compiles to essentially the same `INNER JOIN` chain — already covered conceptually in "Relationships, Fetching, and the N+1 Problem," not repeated here.
- `LEFT JOIN` keeps every row from the left table even without a match, filling unmatched right-side columns with `NULL` — the standard way to find "rows missing a related row," demonstrated with this project's own real "topics not yet published in English" query.
- `RIGHT JOIN` mirrors `LEFT JOIN` (rarely needed in practice); `FULL JOIN` keeps unmatched rows from both sides.

Cheat Sheet

```
SELECT a.x, b.y
FROM a
INNER JOIN b ON a.b_id = b.id;

SELECT a.x, b.y
FROM a
LEFT JOIN b ON a.b_id = b.id
WHERE b.id IS NULL;   -- rows in a with no matching b
```

Glossary

- **JOIN**: a query operation combining rows from two or more tables based on a condition, typically a foreign key equality.
- **INNER JOIN**: returns only rows with a match on both joined tables.
- **LEFT JOIN**: returns every row from the left table, with `NULL` on the right where no match exists.
- **Alias**: a short name (`t` , `cat` , `c`) given to a table in a query, used to disambiguate columns shared between joined tables.