SELECT and Filtering

The basic shape of SELECT, naming columns explicitly instead of SELECT *, comparison /logical operators, LIKE pattern matching, IN/BETWEEN, and why NULL can only be filtered with IS NULL, not = -- on this project's own real topic/category table. The 7th lesson in the PostgreSQL Foundations category.

Beginner 20 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/select-and-filtering

---

Every `SELECT` used so far in this course was a subquery, buried inside an `INSERT` or an `UPDATE`, looking up exactly one id by exactly one slug. This lesson looks at `SELECT` as the main event — reading rows back, filtering them down, on this project's own real `topic` and `category` tables.

SELECT: The Basic Shape

```
SELECT slug, difficulty, estimated_minutes
FROM topic
WHERE category_id = 1;
```

`SELECT <columns> FROM <table> WHERE <condition>;` — columns first, then the table, then an optional filter. Run against this project's own real data, that statement returns every `topic` row belonging to whichever category has `id = 1`, showing only three of its columns.

Selecting Specific Columns vs. SELECT *

`SELECT *` returns every column a table has, in its current column order — convenient at a `psql` prompt while exploring, but worth avoiding in anything meant to last: it silently changes shape the moment a migration adds a column (a real possibility in a project whose Flyway migrations, as "Databases, Schemas, Tables, and Basic SQL Syntax" and "PostgreSQL Data Types" already showed, alter tables constantly), and it fetches columns nothing downstream needed. Naming columns explicitly, as every other example in this lesson does, keeps a query's output shape stable regardless of what the table grows into later.

WHERE and Comparison Operators

PostgreSQL's comparison operators are what you'd expect from any programming language, with one spelling difference: `=` for equality (not `==`), plus `<>` or `!=` for "not equal" (both work identically), `<`, `>`, `<=`, `>=`.

```
SELECT slug FROM topic WHERE difficulty = 'ADVANCED';
SELECT slug FROM topic WHERE estimated_minutes > 20;
SELECT slug FROM course WHERE slug <> 'postgresql';
```

The first is real against this project's own data — `difficulty` is stored as text (`Difficulty` is mapped with `@Enumerated(EnumType.STRING)`, which "PostgreSQL Data Types" would classify as a plain `VARCHAR` column from SQL's point of view), so it's compared the same way any string column would be, with single quotes around the literal.

Logical Operators: AND, OR, NOT

Multiple conditions combine with `AND`, `OR`, and `NOT`, exactly like Java's `&&`, `||`, and `!`:

```
SELECT slug FROM topic
WHERE difficulty = 'ADVANCED' AND estimated_minutes > 25;

SELECT slug FROM topic
WHERE difficulty = 'BEGINNER' OR difficulty = 'INTERMEDIATE';
```

`AND` binds tighter than `OR`, the same precedence order arithmetic's `*` has over `+` — mixing them without parentheses is a real, common source of a filter silently matching more or fewer rows than intended:

```
-- Almost certainly not what's intended: AND binds before OR,
-- so this reads as "difficulty = 'ADVANCED' AND category_id = 5"
-- OR "difficulty = 'BEGINNER'" (any category at all)
WHERE difficulty = 'ADVANCED' OR difficulty = 'BEGINNER' AND category_id = 5

-- What was probably meant — parentheses make the grouping explicit
WHERE (difficulty = 'ADVANCED' OR difficulty = 'BEGINNER') AND category_id = 5
```

LIKE and Pattern Matching

`LIKE` matches strings against a pattern using two wildcards: `%` matches any sequence of characters (including none), `_` matches exactly one character.

```
SELECT slug FROM topic WHERE slug LIKE 'postgresql%';
SELECT slug FROM topic WHERE slug LIKE '%data-types';
SELECT slug FROM topic WHERE slug LIKE '%postgresql%';
```

The first, run against this project's own real `topic` table, matches slugs like `postgresql-and-the-relational-model` and `postgresql-data-types` — `connecting-to-postgresql` does *not* match it (the `%` is trailing, so it only matches slugs starting with `postgresql`), but would match `WHERE slug LIKE '%postgresql%'` instead, wildcards on both sides. `LIKE` is case-sensitive by default; `ILIKE` is the case-insensitive version, PostgreSQL-specific (not standard SQL).

IN and BETWEEN

`IN` matches against a list of values in one condition, instead of chaining several `OR`s:

```
SELECT slug FROM topic
WHERE difficulty IN ('INTERMEDIATE', 'ADVANCED');
```

equivalent to, but shorter and clearer than, `difficulty = 'INTERMEDIATE' OR difficulty = 'ADVANCED'` — "Inserting, Updating, and Deleting Data" already used this exact operator in a real `DELETE ... WHERE example_name IN (...)`. `BETWEEN` checks an inclusive range in one condition:

```
SELECT slug FROM topic
WHERE estimated_minutes BETWEEN 15 AND 20;
```

equivalent to `estimated_minutes >= 15 AND estimated_minutes <= 20` — both bounds included, which is easy to forget since not every language's range helper is inclusive on both ends.

NULL and Filtering: Why = NULL Doesn't Work

"Constraints and Keys" already established that PostgreSQL never considers one `NULL` equal to another — the same rule applies to filtering, and it trips up `WHERE` far more often than it trips up a `UNIQUE` constraint. This project's own `topic.estimated_minutes` column is nullable (unlike `slug` or `difficulty`), so it's a real column where this matters:

```
-- Returns ZERO rows, always — not "rows where estimated_minutes is NULL"
SELECT slug FROM topic WHERE estimated_minutes = NULL;

-- The correct way to check for NULL
SELECT slug FROM topic WHERE estimated_minutes IS NULL;

-- And its opposite
SELECT slug FROM topic WHERE estimated_minutes IS NOT NULL;
```

`= NULL` isn't false, exactly — in PostgreSQL's three-valued logic (`true` / `false` / `unknown`), comparing anything to `NULL` with `=` evaluates to `unknown`, and `WHERE` only keeps rows

where the condition is `true`, so an `unknown` row is silently dropped either way. `IS NULL` / `IS NOT NULL` are the only correct way to test for `NULL` — they're a distinct piece of syntax from `=` / `<>`, not a special case of them.

Common Misconceptions

"`SELECT *` is a harmless shortcut for `psql`." It's fine for ad hoc exploration, but bringing it into application code or a saved query means a later `ALTER TABLE ADD COLUMN` silently changes that query's result shape, often far from where the migration happened. **"`!=` and `<>` mean different things."** They don't — both are "not equal," `<>` is the SQL-standard spelling and `!=` a widely supported alias; this project's own style has no strong preference between them. **"`LIKE '%text%'` and a full-text search are the same thing."** They're not — `LIKE` does a literal substring/pattern match with no notion of word boundaries, relevance ranking, or stemming; PostgreSQL's actual full-text search (a separate, more advanced feature) is out of scope for this course.

Best Practices

- Name columns explicitly instead of `SELECT *` in anything beyond a one-off `psql` check — it keeps a query's shape stable as this project's own migrations keep adding, renaming, and altering columns over time.
- Reach for `IN (...)` instead of a chain of `OR`s comparing the same column to different literals — "Inserting, Updating, and Deleting Data"'s real `DELETE ... WHERE example_name IN (...)` is a genuine example of this being both shorter and clearer.
- Parenthesize `AND` / `OR` combinations explicitly whenever both appear in the same `WHERE` clause, even when the default precedence would technically produce the intended result — it removes any need for the reader to recall which operator binds tighter.
- Check any nullable column (`estimated_minutes`, in this project) with `IS NULL` / `IS NOT NULL`, never `=` / `<>` against `NULL` — the difference between "no rows found" and "a query that's silently wrong" often comes down to exactly this.

Common Mistakes

- Writing `WHERE column = NULL` (or `<> NULL`) expecting it to filter on nullability, and being confused when the query returns nothing, or unexpectedly excludes rows that should have matched.
- Forgetting `LIKE`'s wildcard placement matters directionally — `'postgresql%'` (starts with) and `'%postgresql'` (ends with) return genuinely different rows, and neither is a general "contains" search without wildcards on both ends.
- Mixing `AND` and `OR` in one `WHERE` clause without parentheses and trusting operator precedence to "obviously" mean what was intended — it usually compiles and runs without error, which is exactly what makes the resulting wrong row count easy to miss.

- Treating `IN (...)` as doing anything more than the equivalent chain of `OR`s it replaces — it offers no automatic deduplication or type coercion beyond what a single `=` comparison would already do for each value.

Summary, Cheat Sheet, and Glossary

Summary

- `SELECT <columns> FROM <table> WHERE <condition>;` is the full shape; naming columns explicitly instead of `SELECT *` keeps a query's result shape stable as this project's schema evolves.
- Comparison operators (`=`, `<>` / `!=`, `<`, `>`, `<=`, `>=`) work as expected; `AND` / `OR` / `NOT` combine conditions, with `AND` binding tighter than `OR` — parenthesize explicitly when mixing them.
- `LIKE` matches a pattern with `%` (any sequence) and `_` (exactly one character), case-sensitive by default; `ILIKE` is PostgreSQL's case-insensitive variant.
- `IN (...)` replaces a chain of `OR`s on the same column; `BETWEEN a AND b` is an inclusive range check.
- `NULL` is never equal to anything, including another `NULL`, under `=` / `<>` — `IS NULL` / `IS NOT NULL` are the only correct way to filter on it, a direct continuation of the same rule "Constraints and Keys" already established for `UNIQUE`.

Cheat Sheet

```
SELECT a, b FROM t WHERE a = 1;
SELECT a, b FROM t WHERE a <> 1 AND b > 5;
SELECT a, b FROM t WHERE a LIKE 'foo%';
SELECT a, b FROM t WHERE a IN (1, 2, 3);
SELECT a, b FROM t WHERE a BETWEEN 1 AND 10;
SELECT a, b FROM t WHERE a IS NULL;
SELECT a, b FROM t WHERE a IS NOT NULL;
```

Glossary

- **Wildcard**: a special character in a `LIKE` pattern — `%` for any sequence of characters, `_` for exactly one.
- **Three-valued logic**: PostgreSQL's `true` / `false` / `unknown` evaluation model, the reason `= NULL` never matches anything.
- **ILIKE**: PostgreSQL's case-insensitive equivalent of `LIKE`, not part of standard SQL.
- **Predicate**: the general term for any condition in a `WHERE` clause that evaluates to true, false, or unknown for a given row.