Inserting, Updating, and Deleting Data

INSERT/UPDATE/DELETE, with this project's own real INSERT ... SELECT and UPDATE ... WHERE migrations, RETURNING, ON CONFLICT (upsert), and how Flyway guarantees this project's migrations never need ON CONFLICT. The 6th lesson in the PostgreSQL Foundations category.

Intermediate 25 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/inserting-updating-and-deleting-data

---

Every lesson in this course so far has been read-only — looking at structure (`CREATE TABLE`) and looking at types, never actually changing a row. This lesson is where that changes: `INSERT`, `UPDATE`, and `DELETE`, read through this project's own real migrations, which have been writing rows this exact way since `V1`.

INSERT: The Basic Shape

The simplest form names the table, the columns, and the values:

```
INSERT INTO course (name, slug, sort_order)
VALUES ('PostgreSQL', 'postgresql', 5);
```

This is real — the exact statement that created this course's own `course` row. Column order in the parentheses must match the value order that follows; any column left out (like `id`, generated by the `BIGSERIAL` covered in "PostgreSQL Data Types") takes its default.

This Project's Own INSERT ... SELECT Pattern

A plain `VALUES` list only works when every value is a literal. Almost every migration in this project instead needs a foreign key — a `category_id` or `topic_id` — and doesn't know that generated number in advance. The pattern this project uses everywhere is `INSERT ... SELECT`, looking the id up by slug at insert time:

```
INSERT INTO topic (category_id, slug, difficulty, estimated_minutes, sort_order)
SELECT id, 'connecting-to-postgresql', 'BEGINNER', 15, 2
FROM category
WHERE slug = 'postgresql-foundations';
```

Instead of a `VALUES` clause, the columns are populated from a `SELECT` — `id` comes from whichever row `category` the `WHERE slug = ...` matches, and the rest are still literals. This is the same `SELECT` syntax "SELECT and Filtering," two lessons ahead, covers properly — for now, treat it as "insert the result of this query as one row." It's what lets a migration reference `postgresql-foundations`'s `category_id` without ever hardcoding a specific number that might differ between a fresh database and one that's run every migration since `V1`.

UPDATE: Changing Existing Rows

`UPDATE` modifies existing rows rather than creating new ones, and — critically — needs a `WHERE` clause to avoid rewriting every row in the table:

```
UPDATE topic_translation
SET published = true
WHERE language = 'en'
  AND topic_id = (SELECT id FROM topic WHERE slug = 'connecting-to-postgresql');
```

This is the real statement that published this very course's second lesson in English — the two-step publish pattern ("PostgreSQL Overview" published in Turkish immediately, English flipped later) that CLAUDE.md documents as a project-wide convention runs on nothing more exotic than this. `SET` can update multiple columns at once, comma-separated (`SET published = true, seo_title = '...'`), and the `WHERE` clause can be as specific as needed — here, a subquery (a `SELECT` nested inside another statement) finds the right `topic_id` by slug, exactly the same technique the `INSERT ... SELECT` pattern above uses.

DELETE: Removing Rows

`DELETE` removes whole rows, and, like `UPDATE`, needs a `WHERE` clause for the same reason — omitting it deletes every row in the table:

```
DELETE FROM code_example
WHERE example_name IN ('CardBase', 'CardDemo')
  AND topic_id = (SELECT id FROM topic WHERE slug = 'component-composition');
```

This is a real migration from the React course, removing two example rows that were no longer embedded in that lesson's markdown after a content change — `DELETE` doesn't touch the `code_example` table's structure at all, only the two rows matching this condition. `IN (...)` here matches any of a list of values, rather than one exact value with `=` — worth recognizing now, ahead of "SELECT and Filtering" covering it properly.

RETURNING: Getting a Row Back From a Write

An `INSERT`, `UPDATE`, or `DELETE` normally reports only how many rows were affected — `RETURNING` makes it hand back the actual row data, in the same statement, without a separate `SELECT` afterward:

```
INSERT INTO course (name, slug, sort_order)
VALUES ('PostgreSQL', 'postgresql', 5)
RETURNING id;
```

This immediately returns the new `id` PostgreSQL just generated via `BIGSERIAL` — useful anywhere application code needs that generated value right away, without a round trip to query for it. `RETURNING` works identically on `UPDATE` and `DELETE`, returning the row as it looked *after* the update or *before* the delete. This project's own migrations never use it — a Flyway migration runs once, unattended, and has no calling code waiting to receive a value back — but it's exactly what Hibernate relies on internally whenever it needs the database-generated `id` of a row it just inserted through a repository's `save(...)`.

INSERT ... ON CONFLICT: Upsert

`ON CONFLICT` tells `INSERT` what to do instead of failing when a row would violate a `UNIQUE` or `PRIMARY KEY` constraint — the constraint mechanics "Constraints and Keys" already covered are exactly what `ON CONFLICT` reacts to:

```
INSERT INTO category (course_id, name, slug, sort_order)
VALUES (5, 'PostgreSQL Foundations', 'postgresql-foundations', 1)
ON CONFLICT (course_id, slug) DO NOTHING;
```

`ON CONFLICT (course_id, slug)` names the exact constraint being watched for — here, this project's own real `uq_category_course_slug` composite `UNIQUE` from "Constraints and Keys." `DO NOTHING` silently skips the insert if a matching row already exists, instead of raising a constraint-violation error. The alternative, `DO UPDATE`, updates the conflicting row instead of skipping it:

```
INSERT INTO category (course_id, name, slug, sort_order)
VALUES (5, 'PostgreSQL Foundations', 'postgresql-foundations', 1)
ON CONFLICT (course_id, slug)
DO UPDATE SET name = EXCLUDED.name;
```

`EXCLUDED` refers to the row that was about to be inserted — `EXCLUDED.name` is the new `'PostgreSQL Foundations'` value, distinguishing it from `category.name`, the *existing* row's current value. This combined "insert, or update if it's already there" behavior is commonly called an **upsert**.

Why This Project's Migrations Never Use ON CONFLICT

Every `INSERT` shown in this lesson so far comes from a real Flyway migration, and none of them use `ON CONFLICT` — worth understanding *why*, not just noting it. Flyway guarantees each numbered migration runs exactly once per database, in order, and is checksummed against modification — so a migration inserting `postgresql` / `postgresql-foundations` can safely assume neither row exists yet; there's no conflict to handle because Flyway itself is the mechanism preventing one. `ON CONFLICT` earns its place in code that might run more than once against data that might already be there — a data-loading script re-run by hand, an idempotent seed script, or (closer to application code) a `save(...)` call meant to insert-or-update depending on whether a natural key already exists. This project's Java code never reaches for it either, for the same underlying reason: `JpaRepository.save(...)` already decides insert-vs-update based on whether its `@Id` is `null`, which "Entities and the Repository Abstraction" already covered — a different mechanism solving a related problem one layer up.

Common Misconceptions

"`UPDATE` / `DELETE` without `WHERE` only affects the 'current' row." There's no such concept in SQL — omitting `WHERE` targets every row in the table, immediately, with no confirmation prompt. **"`RETURNING` runs a second query."** It doesn't — it's the same single statement, returning data it already computed while performing the write, not an extra round trip. **"`ON CONFLICT` catches any error during `INSERT`."** It only catches a `UNIQUE` / `PRIMARY KEY` / `EXCLUDE` constraint violation on the specific column(s) named — a `NOT NULL` violation or a foreign key violation still fails the statement outright.

Best Practices

- Write the `WHERE` clause of an `UPDATE` or `DELETE` first, mentally, before the `SET`/columns — and consider testing it as a `SELECT` with the identical condition first, to see exactly which rows would be affected before committing to changing them.
- Prefer `INSERT ... SELECT` (this project's own pattern throughout its migrations) over hardcoding a foreign key's numeric id — it stays correct regardless of exactly which id a `category` or `topic` ended up with on a given database.
- Reach for `RETURNING` instead of a follow-up `SELECT` whenever code needs a value a write just produced (a generated id, a computed default) — it's one round trip instead of two, and atomic with the write itself.
- Treat `ON CONFLICT` as a signal, not a default habit — needing it usually means the same operation might run more than once against the same data, which is worth naming explicitly (an idempotent script, a natural-key upsert) rather than reaching for reflexively.

Common Mistakes

- Running an `UPDATE` or `DELETE` in `psql` without a `WHERE` clause, intending to test on "just this one row" — nothing in SQL enforces that intention; every unqualified row is affected the instant the statement runs.
- Forgetting that `INSERT ... SELECT`'s `SELECT` can return zero rows (a slug that doesn't match anything) — the `INSERT` then silently inserts zero rows too, with no error, which is a much quieter failure than a typo in a `VALUES` literal would be.
- Using `ON CONFLICT DO UPDATE` without realizing `EXCLUDED` refers to the row that *would have been* inserted, not the existing row already in the table — writing `SET name = name` instead of `SET name = EXCLUDED.name` silently keeps the old value forever.
- Assuming `DELETE FROM table` (with no `WHERE`) is equivalent to `TRUNCATE TABLE table` — both empty the table, but `DELETE` does it row by row (slower on a large table, and it fires any triggers) while `TRUNCATE` is a much faster structural operation; the distinction matters once table sizes stop being trivial.

Summary, Cheat Sheet, and Glossary

Summary

- `INSERT INTO table (columns) VALUES (...)` adds a new row; this project's own migrations almost always use `INSERT ... SELECT` instead, looking a foreign key's id up by slug rather than hardcoding a number.
- `UPDATE table SET column = value WHERE ...` and `DELETE FROM table WHERE ...` both require a `WHERE` clause to avoid affecting the entire table — this project's real publish-English migrations are a live `UPDATE ... WHERE` example.
- `RETURNING` hands back the affected row's data in the same statement, avoiding a separate follow-up `SELECT` — Hibernate relies on the same idea internally for generated ids, even though this project's own migrations never use it directly.
- `INSERT ... ON CONFLICT (columns) DO NOTHING / DO UPDATE ...` is an upsert, reacting to the exact `UNIQUE` / `PRIMARY KEY` constraint named — this project's migrations never need it, because Flyway itself guarantees each migration runs exactly once.
- `EXCLUDED` refers to the row that was about to be inserted inside an `ON CONFLICT DO UPDATE` clause, distinct from the existing row already in the table.

Cheat Sheet

```
INSERT INTO t (a, b) VALUES (1, 2);
INSERT INTO t (a, b) SELECT id, 'x' FROM other WHERE slug = 'y';

UPDATE t SET a = 1 WHERE id = 5;
DELETE FROM t WHERE id = 5;

INSERT INTO t (a, b) VALUES (1, 2)
RETURNING id;

INSERT INTO t (a, b) VALUES (1, 2)
ON CONFLICT (a) DO NOTHING;

INSERT INTO t (a, b) VALUES (1, 2)
ON CONFLICT (a) DO UPDATE SET b = EXCLUDED.b;
```

Glossary

- **Upsert**: a write that inserts a new row, or updates the existing one if a conflicting key is already present — PostgreSQL's version of this is `INSERT ... ON CONFLICT ... DO UPDATE`.
- **RETURNING**: a clause on `INSERT` / `UPDATE` / `DELETE` that hands back the affected row's data as part of the same statement.
- **EXCLUDED**: inside `ON CONFLICT DO UPDATE`, the pseudo-table referring to the row that was about to be inserted.
- **Subquery**: a `SELECT` nested inside another statement — used above both in `INSERT ... SELECT` and in a `WHERE ... = (SELECT ...)` condition.