Indexes and Query Performance with EXPLAIN

EXPLAIN/EXPLAIN ANALYZE, Seq Scan vs. Index Scan, CREATE INDEX with this project's own real idx_topic_category B-tree index, partial/expression indexes, and revisiting OFFSET's cost with keyset pagination. The 3rd lesson in the Advanced PostgreSQL category.

Advanced 30 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/indexes-and-query-performance-with-explain

---

"Sorting, Limiting, and Pagination" mentioned `OFFSET`'s cost growing with the offset, and "Constraints and Keys" mentioned `PRIMARY KEY` automatically building an index, both deferred here. This lesson is where indexes and query performance are finally taught for real — not as abstract DBA trivia, but the way this project's own migrations actually use them: query, `EXPLAIN`, read the plan, understand the scan type, `EXPLAIN ANALYZE`.

EXPLAIN: Seeing the Query Plan

`EXPLAIN` shows how PostgreSQL intends to execute a query, without actually running it:

```
EXPLAIN SELECT * FROM topic WHERE category_id = 1;
```

```
Seq Scan on topic  (cost=0.00..1.14 rows=10 width=44)
  Filter: (category_id = 1)
```

`Seq Scan` (sequential scan) means PostgreSQL reads every row of `topic`, top to bottom, checking each one against `category_id = 1` — no shortcuts. `cost=0.00..1.14` is PostgreSQL's own internal cost estimate (not seconds — an abstract unit reflecting mostly disk I/O and CPU work), `rows=10` is its estimate of how many rows will match, and `width=44` estimates the average row size in bytes. This is exactly what "PostgreSQL and the Relational Model"'s query-to-database story never had a tool to actually look at — `EXPLAIN` is that tool.

Seq Scan vs. Index Scan

A `Seq Scan` isn't inherently bad — on a small table, like nearly every one of this project's own tables today, scanning every row is often genuinely the cheapest option, cheaper than

the overhead of consulting an index at all. An `Index Scan` is the alternative: instead of reading every row, PostgreSQL uses an index to jump directly to the rows that match, at the cost of maintaining that index on every write. Whether PostgreSQL chooses one or the other for a given query depends on its own cost estimate, based on table size, how selective the condition is, and whether a usable index exists in the first place — which is exactly why this project's own real tables, at their current small size, would show a `Seq Scan` even on columns this lesson will index below, and that's the correct choice PostgreSQL is making, not a sign something is missing.

Creating a B-Tree Index, and Watching the Plan Change

`CREATE INDEX` is the same statement this project's own `V1__init_schema.sql` already uses four times:

```
CREATE INDEX idx_topic_category ON topic (category_id);
```

This real index exists precisely because `topic.category_id` is a foreign key ("Constraints and Keys" already covered why a foreign key needs `REFERENCES`) that gets filtered and joined on constantly — every `WHERE category_id = ...`, and every `JOIN category ... ON t.category_id = cat.id` from "JOINs," benefits from it. Without `USING <method>` specified, `CREATE INDEX` defaults to a **B-tree** — a balanced tree structure that keeps values in sorted order, letting PostgreSQL locate a matching row (or range of rows) in roughly logarithmic time instead of scanning every row linearly; it's the right default for equality and range conditions (`=`, `<`, `>`, `BETWEEN`), and by far the most common index type in practice, including in every index this project's own schema defines. On a table large enough for it to matter, running `EXPLAIN` again after this index exists would show `Index Scan using idx_topic_category` in place of `Seq Scan` — this project's own tables are simply too small today for PostgreSQL to judge the index worth using yet, which is itself a real, useful thing to be able to read from `EXPLAIN`'s output rather than assume.

EXPLAIN ANALYZE: What Actually Happened

`EXPLAIN` alone only estimates; `EXPLAIN ANALYZE` actually runs the query and reports what really happened:

```
EXPLAIN ANALYZE SELECT * FROM topic WHERE category_id = 1;
```

```
Seq Scan on topic  (cost=0.00..1.14 rows=10 width=44)
                    (actual time=0.012..0.018 rows=10 loops=1)
  Filter: (category_id = 1)
```

The second line is new — `actual time` is real elapsed milliseconds, `rows=10` here is the real count returned (not an estimate), and `loops=1` counts how many times this step ran (more

than once for something like the inner side of certain join strategies). Comparing the estimated `rows` from plain `EXPLAIN` against the actual `rows` from `EXPLAIN ANALYZE` is one of the most useful things this pair of tools offers — a big gap between them means PostgreSQL's own statistics about the table are stale or misleading, which can itself cause it to pick a worse plan than it otherwise would.

Partial Indexes

An index doesn't have to cover every row — a **partial index** adds a `WHERE` clause to the index definition itself, indexing only the rows that match:

```
CREATE INDEX idx_topic_translation_published_en
    ON topic_translation (topic_id)
    WHERE language = 'en' AND published = true;
```

This is a genuinely useful hypothetical index for this project's own real query pattern from "JOINs"'s `LEFT JOIN ... WHERE en.id IS NULL` (finding topics not yet published in English) — smaller than an index over every `topic_translation` row regardless of language or publish state, and faster to maintain on every write, since only rows matching the `WHERE` condition need updating in it at all. A partial index is a genuine trade-off: it only helps queries whose own condition matches (or is implied by) the index's `WHERE` clause — a query filtering on `language = 'tr'` gets no benefit from an index defined `WHERE language = 'en'`.

Expression Indexes

An index can also be built on the *result of an expression* rather than a raw column value — useful the moment a query filters on a computed or transformed version of a column:

```
CREATE INDEX idx_topic_slug_lower ON topic (LOWER(slug));
```

"Databases, Schemas, Tables, and Basic SQL Syntax" already established that this project's own unquoted identifiers are folded to lowercase automatically, so this specific example is hypothetical for this project's real schema — but the general pattern is real and common: a query like `WHERE LOWER(email) = 'user@example.com'` (case-insensitive lookup) can only use an index if the index itself was built on `LOWER(email)`, not on `email` directly — a plain index on `email` doesn't help a query filtering on a transformed version of it, since the index stores the raw values, not the computed ones.

Revisiting OFFSET's Cost: Keyset Pagination

"Sorting, Limiting, and Pagination" named `OFFSET`'s cost — scanning and discarding every skipped row — without a way to fix it yet. **Keyset pagination** (sometimes called "seek pagination") replaces `OFFSET` with a `WHERE` condition on the last row already seen:

```
-- Instead of OFFSET 300 LIMIT 20 (must scan and discard 300 rows)
SELECT slug, sort_order FROM topic
WHERE sort_order > 300
ORDER BY sort_order
LIMIT 20;
```

With an index on `sort_order`, this `WHERE sort_order > 300` becomes an `Index Scan` that jumps directly to the right starting point, regardless of how deep into the results "page 16" happens to be — unlike `OFFSET 300`, whose cost keeps growing the further into the results a page sits. The trade-off is real: keyset pagination can't jump to an arbitrary page number the way `OFFSET` can (it can only move forward from a known row), which is exactly why `OFFSET`-based pagination, simpler and more flexible for small tables like this project's own, remains the right default until a table's size and access pattern genuinely call for the alternative.

Common Misconceptions

"An index always makes a query faster." Not automatically — on a small table (like nearly every one of this project's own), a `Seq Scan` is often genuinely cheaper, and every index also costs something on every `INSERT` / `UPDATE` / `DELETE` to that table, a cost `EXPLAIN` on a `SELECT` never shows. **"More indexes are always better."** Each one adds write overhead and storage, whether or not it's ever chosen by the planner — an unused index is pure cost with no query ever benefiting from it. **"`EXPLAIN` and `EXPLAIN ANALYZE` show the same thing, just with different formatting."** `EXPLAIN` alone never runs the query — it estimates; `EXPLAIN ANALYZE` genuinely executes it (including any side effects, for a non-`SELECT` statement), which is worth remembering before running it on something that writes data.

Best Practices

- Reach for `EXPLAIN` before assuming a slow query needs an index at all — this project's own tables, at their current size, are a real example of a case where `Seq Scan` is already the right plan, and adding an index would only add write overhead with zero read benefit.
- Index columns that are actually filtered or joined on frequently, following this project's own real `idx_topic_category` / `idx_category_course` pattern (both indexing a foreign key column that's joined on constantly) rather than guessing.
- Reach for a partial index the moment a query pattern consistently filters to a small, stable subset of a table (like this project's own hypothetical "published English translations only" case) — it stays smaller and cheaper to maintain than a full-table index.
- Compare `EXPLAIN`'s estimated `rows` against `EXPLAIN ANALYZE`'s actual `rows` when a query's performance is genuinely surprising — a large gap between them is itself a useful diagnostic, pointing at stale table statistics rather than the query's own logic.

Common Mistakes

- Adding an index to a small table and being confused when `EXPLAIN` still shows `Seq Scan` — this isn't a bug or a wasted index necessarily; it's PostgreSQL correctly judging the index not worth using yet at the table's current size.
- Running `EXPLAIN ANALYZE` on a write statement (`UPDATE` / `DELETE`) in a context where the actual write is unwanted, forgetting that `ANALYZE` genuinely executes the statement rather than only estimating it — wrapping it in a transaction that's rolled back afterward is the safe way to inspect a write's plan without keeping its effect.
- Building a plain index on a column, then filtering on a transformed version of it (`LOWER(column)`, `column + 1`) and being surprised the index isn't used — the index has to be built on the same expression the query actually filters on.
- Reaching for keyset pagination as a default rather than a targeted fix, then losing the ability to jump to an arbitrary page number that `OFFSET`-based pagination — this project's own current approach — still supports.

Summary, Cheat Sheet, and Glossary

Summary

- `EXPLAIN` shows PostgreSQL's intended query plan and cost estimate without running the query; `EXPLAIN ANALYZE` actually runs it and reports real elapsed time and row counts.
- `Seq Scan` reads every row; `Index Scan` uses an index to jump directly to matching rows — which one PostgreSQL picks depends on its own cost estimate, not a fixed rule, and a `Seq Scan` on a small table (like this project's own) is often genuinely the cheaper choice.
- `CREATE INDEX` defaults to a B-tree, the same statement this project's own `V1__init_schema.sql` already uses for `idx_topic_category` and three other real indexes on foreign key columns.
- A partial index (`CREATE INDEX ... WHERE ...`) indexes only a subset of rows; an expression index (`CREATE INDEX ... (expression)`) indexes a computed value rather than a raw column — both only help a query whose own condition matches what was indexed.
- Keyset pagination (`WHERE sort_order > <last seen>` instead of `OFFSET`) fixes the cost "Sorting, Limiting, and Pagination" already named, at the cost of losing arbitrary-page jumps — `OFFSET` remains the simpler, right default until a table's size genuinely requires the alternative.

Cheat Sheet

```
EXPLAIN SELECT ...;
EXPLAIN ANALYZE SELECT ...;

CREATE INDEX idx_name ON table (column);
CREATE INDEX idx_name ON table (column) WHERE condition;
CREATE INDEX idx_name ON table (LOWER(column));

-- Keyset pagination
SELECT * FROM t WHERE sort_key > :last_seen ORDER BY sort_key LIMIT :page_size;
```

Glossary

- **Query plan**: PostgreSQL's own chosen strategy for executing a query, shown by `EXPLAIN`.
- **Seq Scan**: reads every row of a table in order, checking each against the query's condition.
- **Index Scan**: uses an index to locate matching rows directly, without reading every row.
- **B-tree**: a balanced tree structure, PostgreSQL's default index type, efficient for equality and range conditions.
- **Keyset pagination**: paging through results using a `WHERE` condition on the last-seen row instead of `OFFSET`, avoiding the cost of scanning and discarding skipped rows.