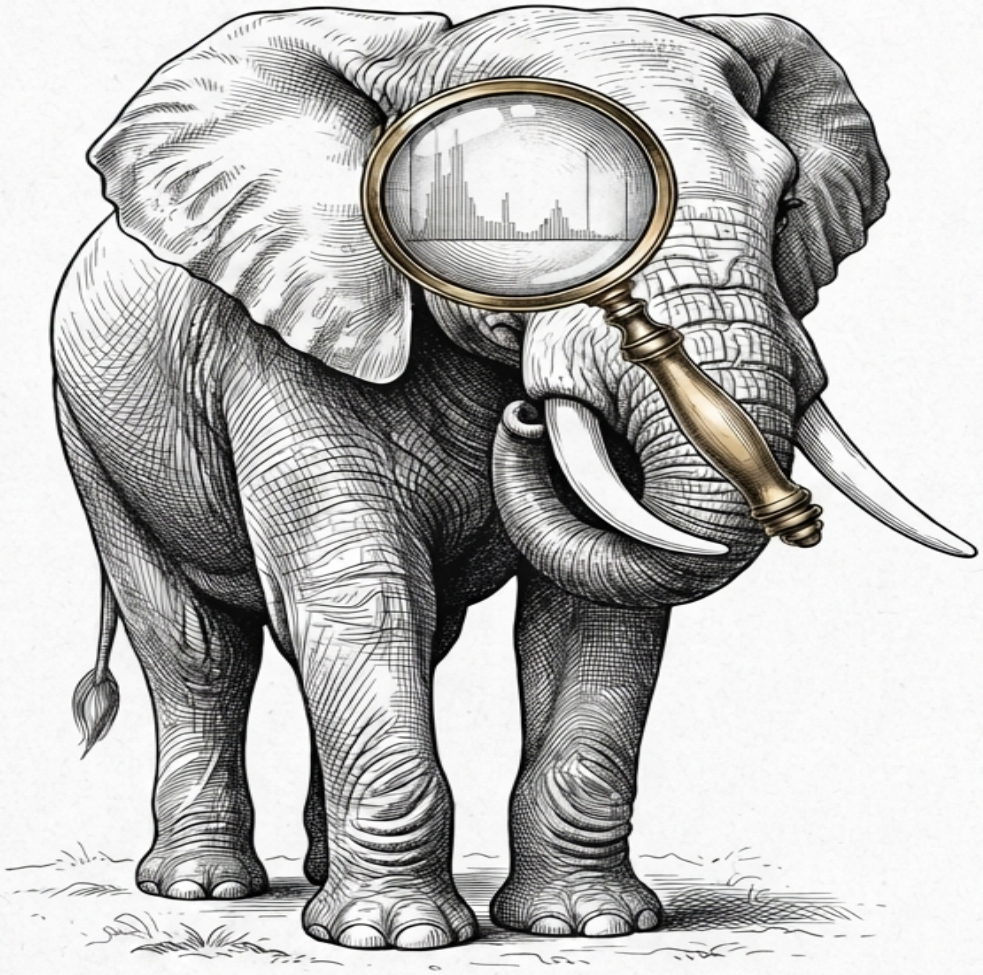
# The Planner's Guessing Game Inside PostgreSQL's ANALYZE



The sampling math, the source code, and the research papers behind every row-count estimate.

Sumit Kumar

# Acknowledgments

This book was written in the hours between bedtime stories and the next morning's coffee. My wife kept everything running while I disappeared into source code and mailing list archives. My daughters reminded me, every single evening, that there is a world outside `pg_statistic`. My mother taught me, long before I ever read a line of C, that patience and persistence are the only tools that matter. This book exists because they let me have the time it took.

# Preface

This book is organized as 12 chapters, each covering one statistic or one piece of machinery the planner leans on, in the order you need to understand them. Chapters that need more than one idea to tell their story are split into numbered sections, so "Section 3.2" means the second section of Chapter 3. Cross-references throughout the book point back or forward by chapter and section number, "see Chapter 2" or "covered in Section 10.7," rather than by a bare title, so you can always find your place regardless of which chapter you are reading.

One convention worth settling here, once, instead of repeating every time it comes up: whenever this book names a function, a struct field, or a file path in a code-styled snippet like `eqsel` or `src/backend/commands/analyze.c`, that is real PostgreSQL source code, not pseudocode or a simplification, unless the text explicitly says otherwise. You can open that exact file at that exact path in a PostgreSQL source checkout and find the function being discussed. This book does not repeat "this is from the PostgreSQL source" at every mention, that convention is declared here, once, and holds for the rest of the book.

A second convention, about where this book's claims come from. Every statement in this book traces back to one of four kinds of source: the PostgreSQL source code itself, read directly from a checkout the way the paragraph above describes. The official PostgreSQL documentation. The published research papers PostgreSQL's own source comments cite by name, Haas and Stokes, Chaudhuri, Motwani and Narasayya, Vitter, Manku and Motwani, Ioannidis and Christodoulakis, among others. And the public **pgsql-hackers mailing list archives**, PostgreSQL's own developer mailing list, searchable at postgresql.org, where the original design discussion behind a feature, or the sole record of a still-open limitation, is sometimes the only explanation for why the code behaves as it does. That last source is named consistently as "the pgsql-hackers mailing list archives" everywhere it appears in this book, rather than under several different descriptions, so you can always tell which of the four kinds of source a given claim rests on. Every specific message cited this way is linked directly, so you can read the original discussion yourself rather than take this book's summary of it on faith. Despite that care, a technical book covering this much source code and history will not get every detail right. Where this book turns out to be wrong, treat the four sources above as the correction, not this book's own restatement of them.

A full table of contents follows this preface. A glossary of every term, function, and abbreviation this book uses, and a references section listing every source cited by name, sit at the very end, after the last chapter.

## Scope: What This Book Leaves Out, and Why

This book answers one question: how does the planner turn a `WHERE` clause, a `GROUP BY`, or a join condition into a row-count guess. That question has a precise boundary, and PostgreSQL has a lot of things called "statistics" that sit outside it. Naming that boundary here, once, is more useful than discovering it by accident three chapters in.

**Cost estimation is a different subject from row estimation, and this book stops at the border between them.** Once the planner has a selectivity and a row count, the numbers this whole book is about, it hands them to a separate cost model that turns them into `seq_page_cost`, `random_page_cost`, and the rest of the cost arithmetic `EXPLAIN` reports. Chapter 8 follows correlation as far as the point where it enters that cost model, the `csquared` interpolation between `min_IO_cost` and `max_IO_cost`, because correlation itself is a statistic `ANALYZE` computes. But the book does not teach the cost model itself: no `cost_seqscan`, no `cost_index`'s start-up term, no `cost_sort`, and no quicksort-versus-external-merge-sort choice that `work_mem` governs. Those are real, and they matter for performance work, but they belong to a book about the cost model, not a book about statistics.

**Cumulative activity statistics are not planner inputs, so most of them stay out.** `pg_stat_user_tables` and `pg_stat_user_indexes` track what has actually happened to a table, how many sequential scans ran, how many index scans, how many rows were fetched or returned, and that is genuinely useful data. But the planner's row estimates never read `seq_scan`, `idx_scan`, or anything from `pg_stat_user_indexes` when it plans a query. Those counters answer "what did queries do to this table," not "how many rows will the next query touch," so they sit outside this book's question entirely. `pg_statio_*`, `pg_stat_io`, `track_io_timing`, and the background writer, checkpointer, and WAL statistics are the same story one level further out, real I/O accounting, not row estimation. The one exception is `n_live_tup`, `n_dead_tup`, and `n_mod_since_analyze` in Chapter 12, and they appear there for a narrow reason: autovacuum reads them to decide when to trigger `VACUUM` and `ANALYZE`, and that scheduling decision is close enough to this book's subject, when statistics go stale and get refreshed, to earn a place. The planner itself still never consults them for a row estimate. It reads `pg_class.reltuples` for that, a different number entirely, and Chapter 12 draws that distinction on purpose.

**Cluster-wide and server monitoring statistics are a different book's subject.** Replication lag, WAL generation, checkpoint timing, archiver activity, connection and SSL statistics, these describe how the server is running, not how many rows a query will return. None of it feeds a selectivity or a cardinality estimate anywhere in the planner, so none of it appears here.

**Some real limitations in PostgreSQL's own statistics system are named but not taught as mechanisms, because no finished mechanism exists to demonstrate.** Two are worth knowing about. First, there is no statistic anywhere for how "TOASTed" a column's values are, so the planner has no way to account for the extra out-of-line fetches a lossy GIN or GiST index scan can trigger. This has been an open, acknowledged gap since at least 2011, not an oversight in this book. Second, `CREATE STATISTICS` only ever describes columns inside a single table, so two correlated columns on opposite sides of a join have no equivalent

mechanism to lean on, the planner falls back to the plain independence assumption Chapter 2 introduces, regardless of how strong the real correlation is. Extending extended statistics across a join has been discussed on the pgsql-hackers mailing list, with patches proposed but not yet committed to any released version, so there is nothing finished here to walk through.

**This book targets PostgreSQL 17, and later additions stay out on purpose.** PostgreSQL 18 added a way to inject statistics values by hand instead of running `ANALYZE`: `pg_restore_attribute_stats` and `pg_restore_relation_stats` write a supplied set of values into `pg_statistic` and `pg_class` directly, and `pg_clear_attribute_stats` and `pg_clear_relation_stats` remove them again. An earlier, differently-named pair, `pg_set_attribute_stats` and `pg_set_relation_stats`, was committed and then reverted before release, so those two names never actually shipped in any version. The final functions exist specifically to support `pg_dump` / `pg_restore`'s new `--statistics` options, letting a restored database skip a full re-`ANALYZE`. All four are close cousins of everything this book covers, but naming them here would misrepresent what 17 actually ships.

**Statistics on foreign tables, and a function's own row-count hint, are real and adjacent, but outside this book's single-local-table frame.** `postgres_fdw` can either ask a remote server directly for a row estimate (`use_remote_estimate`) or run `ANALYZE` against a foreign table to pull a sample over the wire and populate `pg_statistic` exactly as if the table were local, reusing every mechanism this book already builds. `pg_proc.procost` and `pg_proc.prorows` serve a similar purpose for function calls, where the planner has no table to sample at all. These are hints the function's author attaches by hand when creating the function, through `CREATE FUNCTION`'s `COST` and `ROWS` clauses, telling the planner how expensive the call is and, for a set-returning function specifically, how many rows it will produce. `prorows` only applies to a set-returning function, defaulting to `1000`; an ordinary scalar function's `prorows` is `0`. `procost` applies to every function, defaulting to `1` unit for C and internal functions and `100` for everything else, and for a set-returning function it means cost per row returned, not a flat per-call cost. Both are genuine statistics mechanisms in the same spirit as everything else here. They are left out because both reuse mechanisms this book already covers, hand-set hints and stored estimates, without introducing new ideas, and the boundary had to be drawn somewhere.

---

## Prerequisites

Every example in this book is meant to be run, not read past, so a few things need to be in place before Chapter 1 starts.

- **A running PostgreSQL 17 instance**, reachable through `psql`, with permission to create tables in some database you can experiment freely in. None of the examples touch anything outside tables this book creates itself.
- **Superuser access on that instance**, or a role with superuser privileges available to switch to. A handful of examples query `pg_statistic` directly rather than through the friendlier `pg_stats` view, and `pg_statistic` is readable only by superusers, exactly the restriction Section 5.6 explains.

- **A checkout of the PostgreSQL 17 source tree**, matching the instance's own version. The Preface's own convention holds throughout: every file path and function name this book names in a code-styled snippet is real, and you should be able to open that exact file at that exact path and find the function being discussed. Later PostgreSQL releases can move a function or rename a variable, so a 17 checkout is what actually lines up with this book, not whatever checkout happens to be lying around.
- **A C compiler** (`gcc` or `clang`), only for the one standalone reservoir-sampling program in Section 3.7, which you build and run outside the database entirely.
- **A plain-text SQL editor or `psql` itself**, since several examples set a session-level option (`SET client_min_messages`, `SET enable_indexonlyscan`, and similar) partway through and expect you to reset it afterward, exactly as each example shows.

Nothing here needs a specific operating system or a specific way of installing community PostgreSQL. A local build from source, a package-manager install, or a self-managed cloud instance all work identically, since every example is plain SQL and, where noted, plain C, run against the server, not against any particular way of standing one up.

---

## Table of Contents

Every SQL statement you type begins as plain text and passes through several stages before any row moves. PostgreSQL's parser turns the text into a syntax tree, the analyzer resolves names and types, and the rewriter expands views and applies rules. What emerges is a clean internal structure, a `Query`, that says exactly what you asked for. From here, the planner takes over, and its job is the harder one: deciding *how* to actually retrieve those rows.

**A note about the examples:** every `EXPLAIN` output in this book was captured from a real PostgreSQL 17 session. However, because `ANALYZE` works from a random sample, the `rows=` estimates you see on your own instance may differ slightly, `rows=99861` where this book shows `rows=99900`, for example. This is sampling variance at work, not a mistake. Chapter 3 explains exactly why it happens.

Take a plain example, `SELECT * FROM orders WHERE status = 'shipped'`. There is an index on `status`, so the planner could look `'shipped'` up in the index and fetch just the rows it points to. Or it could skip the index entirely and read the whole table straight through, checking `status` on each row as it passes. Which is faster? It depends completely on how many orders are actually shipped. If only a few hundred match, the index wins in a walk, it jumps right to them. But if most of the table is shipped, the index turns into the slow path, because chasing thousands of matches one at a time is more work than a single clean sweep of the table.

So the whole decision rests on one number, how many rows will match, and that is where things get interesting. The number is nowhere in the query, and the planner will not run the query to find it, because that would throw away the very savings it is trying to plan for. So it does the only thing left. It guesses. Everything in this section is about where that guess comes from and how much you can trust it.

The guess is easiest to believe when you watch it flip a plan in front of you. Build a small table so you can run each step yourself:

```
DROP TABLE IF EXISTS demo CASCADE;

CREATE TABLE demo (id int, status text);

INSERT INTO demo
SELECT g, CASE WHEN g % 1000 = 0 THEN 'rare' ELSE 'common' END
FROM generate_series(1, 100000) g;

CREATE INDEX ON demo (status);

ANALYZE demo;
```

Now `status` holds `'common'` on 99,900 rows and `'rare'` on exactly 100 of them. Ask PostgreSQL how it would run each of these, and it answers differently:

```
EXPLAIN SELECT * FROM demo WHERE status = 'rare';

 Index Scan using demo_status_idx on demo  (cost=0.29..8.90 rows=100 width=13)
   Index Cond: (status = 'rare'::text)

EXPLAIN SELECT * FROM demo WHERE status = 'common';

 Seq Scan on demo  (cost=0.00..1791.00 rows=99900 width=13)
   Filter: (status = 'common'::text)
```

Same table, same index, same query shape. For `'rare'` the planner uses the index; for `'common'` it ignores the index and reads the whole table. Nothing in the query text told it to. The only thing that differed was `rows=100` versus `rows=99900`, and those numbers were produced before either query ran a single row. The planner looked at the value you asked for, decided how many rows it would probably match, and chose the plan that fits that count. For a hundred rows an index is a bargain. For nearly a hundred thousand it is a waste, and a plain scan wins.

One caveat before we go on. Every `rows=` in this section is an estimate drawn from a sample, not an exact count, so your own numbers may differ by a little, `rows=99861` where the text shows `99900`, and they can even shift between runs. That wobble is the sampling at work, not a mistake, and Chapter 3, "Sampling: How ANALYZE Reads a Little and Knows a Lot," explains exactly where it comes from.

That is the one idea to hold onto for this entire section: **the planner is a guesser, and every plan it picks is chosen to fit a guessed row count.** The guess came from somewhere. It is not in the query and it is not read from the table at planning time. It came from a small summary of `demo.status` that `ANALYZE` computed and stored, and you can look at the very number the planner used:

```
SELECT most_common_vals, most_common_freqs
FROM pg_stats
WHERE tablename = 'demo' AND attname = 'status';

 most_common_vals | most_common_freqs
------------------+-------------------
 {common,rare}    | {0.999,0.001}
```

The planner multiplied `0.001` by the table's row count to get about 100, and `0.999` to get about 99,900. The estimate was arithmetic on a stored summary. This whole section is the story of where that summary comes from, what is in it, and how the planner turns it into the numbers that decide your plans.