Production query plans without production data

2026-03-08 · 7 min · Radim Marek

TABLE OF CONTENTS

In the previous article we covered how the PostgreSQL planner reads `pg_class` and `pg_statistic` to estimate row counts, choose join strategies, and decide whether an index scan is worth it. The message was clear: when statistics are wrong, everything else goes with it.

Streaming replication provides bit-to-bit replication, so all replicas share the same statistics with primary server.

But there was one thing we didn't talk about. Statistics are specific to the database cluster that generated them. The primary way to populate them is `ANALYZE` which requires the actual data.

PostgreSQL 18 changed that. Two new functions: `pg_restore_relation_stats` and `pg_restore_attribute_stats` write numbers directly into the catalog tables. Combined with `pg_dump --statistics-only`, you can treat optimizer statistics as a deployable artifact. Compact, portable, plain SQL.

The feature was driven by the upgrade use case. In the past, major version upgrades used to leave `pg_statistic`

empty, forcing you to run `ANALYZE`. Which might take hours on large clusters. With PostgreSQL 18 upgrades now transfer statistics automatically. But that's just the beginning. The same logic lets you export statistics from production and inject them anywhere - test database, local debugging, or as part of CI pipelines.

The problem

Your CI database has 1,000 rows. Production has 50 million. The planner makes completely different decisions for each. Running `EXPLAIN` in CI tells you nothing about the production plan. This is the core premise behind RegreSQL. Catching query plan regressions in CI is far more reliable when the planner sees production-scale statistics.

Same applies to **debugging**. A query is slow in production and you want to reproduce the plan locally, but your database has different statistics, and planner chooses the predictable path. Porting production stats can provide you that snapshot of thinking planner has to do in production, without actually going to production.

pg_restore_relation_stats

The first of function behind portable PostgreSQL statistics is `pg_restore_relation_stats`. It writes table-level data directly into `pg_class` in form of variadic name/value pairs.

```
SELECT pg_restore_relation_stats(
    'schemaname', 'public',
```

```
    'relname', 'orders',
    'relpages', 123513::integer,
    'reltuples', 50000000::real,
    'relallvisible', 123513::integer,
    'relallfrozen', 120000::integer
);
```

But that's just an example. Let's modify some real statistics to see the full value. We will create a small table, inject fake production-like statistics and watch the planner to change its mind.

```
CREATE TABLE test_orders (
    id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    customer_id integer NOT NULL,
    amount numeric(10,2) NOT NULL,
    status text NOT NULL DEFAULT 'pending',
    created_at date NOT NULL DEFAULT CURRENT_DATE
);

INSERT INTO test_orders (customer_id, amount, status, create
SELECT
    (random() * 9999 + 1)::int,
    (random() * 5000 + 5)::numeric(10,2),
    (ARRAY['pending','shipped','delivered','cancelled'])[flo
    '2024-01-01'::date + (random() * 365)::int
FROM generate_series(1, 10000);

CREATE INDEX ON test_orders (created_at);
CREATE INDEX ON test_orders (status);
ANALYZE test_orders;
```

When you check the current statistics, it has predictable data.

```
SELECT relname, relpages, reltuples
FROM pg_class WHERE relname = 'test_orders';
```

```
   relname    | relpages | reltuples
--------------+----------+-----------
 test_orders |       74 |     10000
(1 row)
```

With 10,000 rows across 74 pages, the planner picks a sequential scan.

```
EXPLAIN SELECT * FROM test_orders WHERE created_at > '2024-0
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on test_orders  (cost=0.00..199.00 rows=5891 width
   Filter: (created_at > '2024-06-01'::date)
(2 rows)
```

Now inject production-scale table stats:

```
SELECT pg_restore_relation_stats(
    'schemaname', 'public',
    'relname', 'test_orders',
    'relpages', 123513::integer,
    'reltuples', 50000000::real,
    'relallvisible', 123513::integer
);
```

And you might be surprised by the result.

```
EXPLAIN SELECT * FROM test_orders WHERE created_at > '2024-0
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on test_orders  (cost=0.00..448.45 rows=17649 widt
   Filter: (created_at > '2024-06-01'::date)
```

The planner is still using the sequential plan. Only the estimated number of rows has changed. Why? If you remember from previous article, it's where column level statistics come into play. Histogram bounds still match the original 10,000 rows we inserted.

pg_restore_attribute_stats

This function writes column-level statistics into `pg_statistic` the same catalog that ANALYZE populates with MCVs, histograms, and correlation.

In previous section, we left the planner stuck on a sequential scan despite believing the table has 50 million rows. The missing piece is column-level statistics. Let's pick up where we left off and inject histogram bounds for `created_at`.

```
SELECT pg_restore_attribute_stats(
    'schemaname', 'public',
    'relname', 'test_orders',
    'attname', 'created_at',
    'inherited', false::boolean,
    'null_frac', 0.0::real,
    'avg_width', 4::integer,
    'n_distinct', -0.05::real,
    'histogram_bounds', '{2019-01-01,2019-07-01,2020-01-01,2
    'correlation', 0.98::real
);
```

Now the planner knows the data spans 5 years. A query filtering on the last 6 months of 2024 covers a narrow slice.

```
EXPLAIN SELECT * FROM test_orders WHERE created_at > '2024-0
```

```
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Index Scan using test_orders_created_at_idx on test_orders
   Index Cond: (created_at > '2024-06-01'::date)
```

Histogram bounds divide the non-MCV portion of the data into equal-population buckets. If `most_common_vals` accounts for most of the data, the histogram covers only the remaining tail. The number of buckets is controlled by `default_statistics_target` (default 100, meaning 101 bounds).

And that's a plan flip! The histogram tells the planner the data spans 2019–2024, so `> '2024-06-01'` matches a narrow tail. A small fraction of 50 million rows. The index scan that was ignored before is now the obvious choice. Table-level stats set the scale, column-level stats shaped the selectivity, and together they changed the plan.

> The `correlation` statistic tells the planner how closely the physical row order matches the column's sort order. A value near 1.0 means sequential access patterns - making index scan cheaper because the next row is likely on the same or adjacent page. For time-series data like `created_at` where rows are inserted chronologically, correlation is typically very high.

Injecting a skewed distribution

The same function handles MCV lists. In production, your `status` column isn't uniform, 95% of orders are delivered, 1.5% are pending.

```
SELECT pg_restore_attribute_stats(
    'schemaname', 'public',
    'relname', 'test_orders',
```

```
    'attname', 'status',
    'inherited', false::boolean,
    'null_frac', 0.0::real,
    'avg_width', 9::integer,
    'n_distinct', 5::real,
    'most_common_vals', '{delivered,shipped,cancelled,pendin
    'most_common_freqs', '{0.95,0.015,0.015,0.015,0.005}'::r
);
```

You can see

```
EXPLAIN SELECT * FROM test_orders WHERE status = 'pending';
```

```
                                QUERY PLAN
-----------------------------------------------------------
 Bitmap Heap Scan on test_orders  (cost=8.93..90.42 rows=599
   Recheck Cond: (status = 'pending'::text)
   ->  Bitmap Index Scan on test_orders_status_idx  (cost=0.
         Index Cond: (status = 'pending'::text)
(4 rows)
```

and compare it with

```
EXPLAIN SELECT * FROM test_orders WHERE status = 'delivered'
```

```
                          QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test_orders  (cost=0.00..448.45 rows=28458 widt
   Filter: (status = 'delivered'::text)
(2 rows)
```

Same column, same operator, different plans. The planner uses a bitmap index scan for `pending` (1.5% rare enough to justify the index) and a sequential scan for `delivered` (95% being most of the table). The selectivity ratios from the MCV list drive the plan choice.

You might have noticed the row estimates (599 and 28,458) are lower than you'd expect for a 50-million-row table. The planner checks the actual physical file size. Our table is only 74 pages on disk, not the 123,513 we injected. Hence the planner scales `reltuples` down proportionally. The absolute numbers shrink, but the *ratios* between them stay correct, and it's the ratios that determine plan shape. When you use `pg_dump --statistics-only` in practice, you're typically restoring into a database with comparable data volume, so the estimates align naturally.

pg_dump

The functions we covered are the mechanics. For operational use `pg_dump` provides everything you need. PostgreSQL 18 added three flags.

Flag	Effect
`--statistics`	dump the statistics (you have to request it explicitely)
`--statistics-only`	dump only the statistics, not schema or data
`--no-statistics`	do not dump statistics

When you export the statistics for your production database

```
pg_dump --statistics-only -d production_db > stats.sql
```

you will see the output is series of `SELECT pg_restore_relation_stats(...)` and `SELECT pg_restore_attribute_stats(...)` calls. Exactly as we explained above.

The full workflow to turn your production data into testable plans might look like this:

```
# 1. dump schema from production
pg_dump --schema-only -d production_db > schema.sql

# 2. dump statistics from production
pg_dump --statistics-only -d production_db > stats.sql

# 3. create test database with schema
createdb test_db
psql -d test_db -f schema.sql

# 4. load fixture data (optional; masked, minimal)
psql -d test_db -f fixtures.sql

# 5. inject production statistics
psql -d test_db -f stats.sql

# 6. query plans now match production
psql -d test_db -c "EXPLAIN SELECT * FROM test_orders WHERE
```

> Statistics dumps are tiny. A database with hundreds of tables and thousands of columns produces a statistics dump under 1MB. The production data might be hundreds of GB. The statistics that describe it fit in a text file.

Keeping injected statistics alive

Now you might ask yourself, where's the catch? And there's a big one, the autovacuum will eventually kick in and run `ANALYZE`. Which will overwrite your injected statistics with real numbers and you are back where you started.

To prevent this, disable autovacuum analyze on the tables you've injected.

```
-- disable autovacuum
ALTER TABLE test_orders SET (autovacuum_enabled = false);

-- or set analyze threshold so high it nevers kicks-in
ALTER TABLE test_orders SET (autovacuum_analyze_threshold =
```

Be careful here.

If you're also writing data to these tables in dev: running migrations, loading fixtures, testing inserts, the injected statistics will drift further from reality with every write. The planner will plan based on a production distribution that no longer reflects the local data.

For read-only query plan testing this is exactly what you want. For integration tests that modify data, you may need to re-inject statistics after each test run.

And please, never ever do this in production!

What's not covered?

As we have seen earlier, it's not worth trying to inject `relpages` as the planner checks the actual file size and

scales it proportionally. This limits the number of absolute rows planner might estimate. I.e. to get comparable numbers to production environment you still would have to create comparable data volume (which isn't a problem when talking about the primary use case of this feature - restoring backups).

It's also worth to note that `CREATE STATISTICS` used for multivariate correlations, distinct counts across column groups and MCV lists for column combinations are not covered within PostgreSQL 18. Those still require `ANALYZE` after restore. PostgreSQL 19 will close this gap with `pg_restore_extended_stats()`.

Security

The restore functions require the `MAINTAIN` privilege on the target table. This is the same privilege needed for `ANALYZE`, `VACUUM`, `REINDEX`, and `CLUSTER` as it was introduced in PostgreSQL 17.

The easiest way to grant it for automation:

```
GRANT pg_maintain TO ci_service_account;
```

This grants `MAINTAIN` on all tables in the database. Enough for a CI pipeline to inject statistics without needing superuser.

your@email.com Subscribe

postgresql expert

← PREVIOUS
PostgreSQL Statistics:
Why queries run slow