

boringSQL

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT ...
-> Hash Join (cost=123.45..678.90 rows=42)
    Seq Scan on orders (cost=0.00..123.00)
    Hash (cost=... rows=...)
-> Index Scan using customers_pkey ...
```

It works on my database

Regression testing of SQL queries

Radim Marek

2026.pgconf.eu

Who's actually writing your SQL?

- Developers who learned just enough
- ORMs generating queries you've never seen
- LLMs, faster than anyone can review

A cleaner query

Before

```
SELECT count(*)  
FROM orders o  
WHERE NOT EXISTS (  
    SELECT 1 FROM refunds r WHERE r.order_id = o.id  
);
```

After

```
SELECT count(*)  
FROM orders o  
WHERE o.id NOT IN (SELECT order_id FROM refunds);
```

Same rows. Tests pass. Reads like English. Ship it.

What production saw

PG 18	CI: 10 K ORDERS, 500 REFUNDS	PRODUCTION: 5 M ORDERS, 400 K REFUNDS
NOT EXISTS	Hash Anti Join, 1.4 ms	Hash Anti Join, 0.47 s
NOT IN	hashed SubPlan, 1.4 ms	SubPlan, killed at 90 s

Pop quiz

```
SELECT o.*, u.name
FROM orders o
JOIN users u ON u.id = o.user_id
WHERE o.created_at > now() - interval '30 days'
ORDER BY o.created_at DESC
OFFSET 2500
LIMIT 100;
```

Will this use indexes? Which ones?

You can't tell from reading SQL

SAME QUERY	PLAN	BUFFERS
CI statistics	Seq Scan, Hash Join	14
Production statistics	Index Scan, Nested Loop	10,432

plan = SQL × statistics × execution

- SQL: what you wrote. EXPLAIN shows it.
- Statistics: what ANALYZE sampled. EXPLAIN guesses from it.
- Execution: how your application runs it. Only ANALYZE shows it.

| 1. SQL

Time lies. Buffers mostly don't.

MACHINE	TIME	BUFFERS
Hetzner ARM64	154 ms	24,108
Hetzner AMD64	119 ms	24,108
M4 Pro VM	98 ms	24,108

Buffers still have noise

- Cold cache vs warm cache
- Hint bits set on first read
- Visibility map: index-only scans that aren't
- Parallel worker scheduling

Equal warmups, an absolute floor, and a **measured** noise floor

RegreSQL

github.com/boringsql/regresql

```
regresql init postgres://localhost/my_app
regresql discover      # think git status
regresql add           # pick the queries, write bindings
regresql update        # expected result sets: correctness
regresql baseline      # expected buffers and plans: performance
regresql test          # in CI
```

The cleaner query, in CI

```
mode: plan choice (injected stats) | pg_regresql: loaded | GUCs: not | PG major: match
```

```
✓ 1 passing
```

```
x 1 failing
```

```
unrefunded_orders.cost (plan changed)
```

```
Cost (info): 11984407962.97 (baseline: 101351.09)
```

```
Plan shape changed
```

```
⚠ Join type changed: [Anti] → []
```

Same rows: we test the correctness (most of the time).

Why not `auto_explain` and `pg_stat_statements`?

- They tell you after the incident
- `regresql` tells you in the pull request



2. Statistics

The planner doesn't care about rows.

It cares about statistics.

I found a PostgreSQL 19 regression

What the harness said: 18 → 19

```
113 queries · 0 correctness · 7 shape · 9 buffer · 0 spill
```

```
perf    13d    buffers +186.4%, join strategy changed
```

```
perf    30c    buffers +280.7%
```

```
perf    29c    buffers +170.1%    ←
```

```
perf    24a    buffers +73.4%
```

```
...
```

Query 13d

	PG 18	PG 19
Buffers	5.37 M	15.4 M
Execution	1.85 s	3.6 s
Result	identical	identical

PG 19 delays one small, very selective join and carries twice the rows through the nested loops.

My notes, at the time

Deterministic (buffers stable across runs) and not a stats artifact (`reLtuples` match within 0.2% between the two servers).

| I was wrong.

Same server. Same data. ANALYZE, 20 times.

29c, from the same list. PG 18 alone.

ANALYZE samples 30,000 rows. Two plans cost almost the same. The sample picks the winner.

```
PG 18, query 29c, 20 × (ANALYZE; EXPLAIN ANALYZE)
```

12 different plans

18 K buffers	1 of 20 runs	35 ms
125 K buffers	5 of 20 runs	66 ms
2.3 M buffers	5 of 20 runs	780 ms
3.4 M buffers	1 of 20 runs	1.9 s

The noise was the finding

- Comparing PG 18 to PG 19: ANALYZE noise is a nuisance
- One production: ANALYZE noise is an incident

Production re-rolls on every autoanalyze

- ▶ ~10% of a table changes → ANALYZE → new sample → maybe a new plan
- ▶ No deploy. No code change.

Not just a benchmark

A real application: 131 GB, top 50 statements = 82% of execution time

	QUERIES	FLIP IN 20 RUNS	FLIP WITH $\geq 2\times$ BUFFERS
JOB	113	57	25
A real application	53	5	0

Run it on yours

```
psql -X -d restored_copy \  
    -v query="$(cat q.sql)" -v buffers=1 \  
    -f coinflip.sql
```

```
coinflip|app-query|20|2|17|4406786|6735943|1.53
```

github.com/boringsql/coinflip

Your numbers

	DATABASES	QUERIES	FLIP	FLIP WITH $\geq 2\times$ BUFFERS
My personal stats gallery	71	8,772	1,551	98
A real application	1	53	5	0

Two uses of statistics

	PIN	SAMPLE
How	inject one set, identical everywhere	re-ANALYZE, watch the plan
Answers	did my change break it?	is this query a coin flip?
Tool	regresql test --stats	coinflip.sql

Portable statistics (PostgreSQL 18)

```
pg_dump --statistics-only prod > stats.sql
```

- pg_restore_relation_stats: relpages, reltuples
- pg_restore_attribute_stats: MCVs, histograms, n_distinct
- Not in 18: CREATE STATISTICS. 19 adds them.

PG 18 today. 14–17: pg_regresql backports the restore functions, Q4/2026

boringsql.com/posts/portable-stats

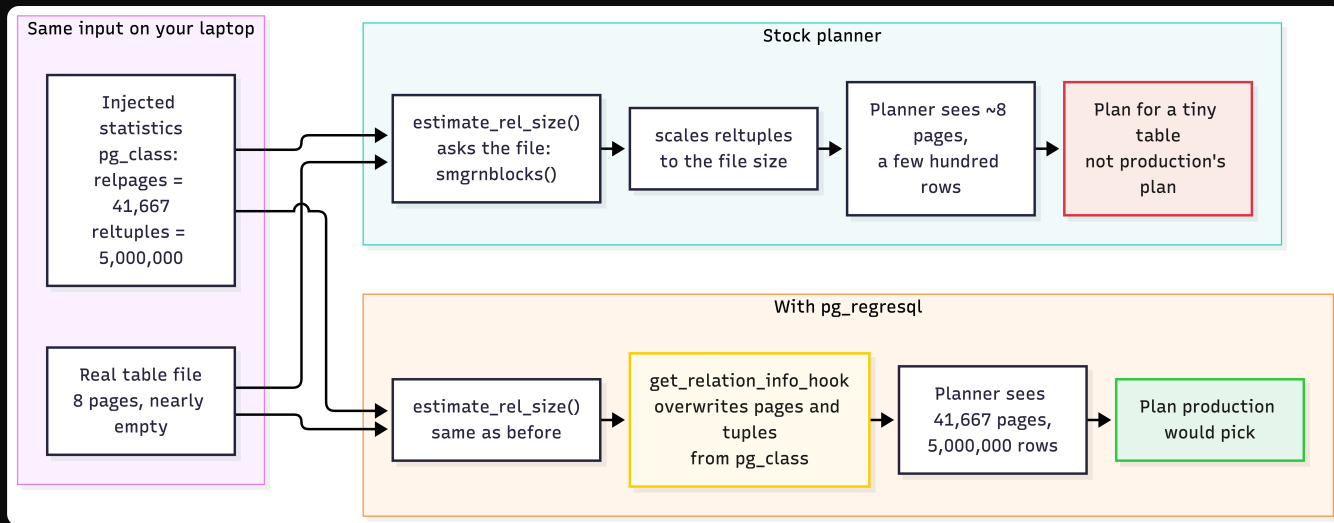
Statistics contain your data

- Most-common values of `users.email` are emails
- Histogram bounds of `orders.amount` are amounts
- Masking statistics like fixture rows: TBD
- Until then: drop PII columns from the dump. Plans on them won't match.

github.com/boringsql/fixtures

pg_regresql

- Problem: the planner reads the real file size and rescales your statistics
- Fix: the extension makes it trust pg_class instead
- Result: your laptop's planner chooses like production's



Two modes, two questions

	PLAN CHOICE	BUFFERS
Data	any, even empty	production-like fixture data
Statistics	production, injected	from that data
Answers	which plan would production pick?	how much work does it do?



3. Execution

The sixth execution. Or the eleventh.

psql is not your application



- Custom plan: planned for the value you passed
- Generic plan: planned once without the value, reused

Watch

```
PREPARE q(text) AS SELECT count(*) FROM orders WHERE status = $1;  
-- bound to 'paid' every time, 60% of the table
```

exec 1	Parallel Seq Scan	41,667 buffers	143 ms
...			
exec 5	Parallel Seq Scan	41,667 buffers	144 ms
exec 6	Parallel Index Only Scan	11,504 buffers	112 ms

It got better. That's not the point.

- Custom plan, the real value: 60% of the table → Parallel Seq Scan
- Generic plan, no value: 1.67 M rows, a third of the table → Parallel Index Only Scan
- The right number picked the plan that reads 3.6× more

The other direction

```
PREPARE q(timestamptzt) AS SELECT ... WHERE created_at > $1;
```

No value for a range: one third of the table

8 CONCURRENT CLIENTS, 3 CPUS	CUSTOM PLAN	GENERIC PLAN
p50 latency	8.6 ms	14.0 ms
p99 latency	23.7 ms	31.2 ms
Parallel workers wanted / launched	none	22,808 / 10,294

When does your driver get there?

DRIVER	SERVER-SIDE PREPARE	GENERIC PLAN POSSIBLE FROM
pgx v5	default	6th call
pgJDBC	from the 5th call	~10th call
psycopg 3	after 5 calls	~11th call
psycopg2, Django default	never	never
PgBouncer 1.21+	only with max_prepared_statements	

Counted per connection

The plan cache decides which plan runs

Not you, and not EXPLAIN.

regresql test --generic: asks the cache what it would settle on

Fails when an index read turns into a seq scan

A real application: 35 of 46 queries settle on the generic plan, none lose an index

What PostgreSQL could do

- Let a test session trust pg_class sizes: pg_regresql is a hook today
- A seedable ANALYZE sample, for tests
- Generic and custom plan counts in pg_stat_statements: in 19, thank you

plan = SQL × statistics × execution

- SQL → regresql test
- Statistics → pin to compare, sample to find coin flips
- Execution → test the plan the cache settles on

Monday morning

```
brew install boringsql/boringsql/regresql      # github.com/boringsql/regresql
sudo pgxn install pg_regresql                  # pgxn.org/dist/pg_regresql
pg_dump --statistics-only prod > stats.sql      # drop PII columns
regresql init postgres://localhost/my_app      # same major as prod
regresql snapshot capture                      # then point regress.yaml at it
regresql add sql/ && regresql baseline && regresql update
regresql test --generic
```

regress.yaml: snapshot.path: snapshots/default.dump, stats.default: ../stats.sql

Then, on a restored copy: coinflip.sql on your top 20 queries

On 14–17: coinflip.sql works today

Thank you

- boringsql.com
- radim@boringsql.com
- github.com/boringsql/regresql

ORM? qShape turns pg_stat_statements into regresql queries