

It works on my database

Regression testing of SQL queries

boringSQL

radim@boringsql.com
Skeptic Turned Believer

boringSQL

**What was the root cause of your
last database-bound incident?**

Did I guess it right?

	Root Cause
60%	Query-Related Issues
20%	Schema, Migrations & Data Issues
15%	Configuration & Resource Issues
5%	Infrastructure

Lies, damned lies, and statistics

What's broken?

**80% of incidents
are query &
schema related**

We test our SQL, right?

SQL is scary 🙈

doesn't feel like "real code"

runs somewhere else

might be invisible

developers never "really learned" SQL

We test around it, not through it

SQL testing

Syntax correctness?

Schema being correct?

Contract maintained?

Knowing how the query will behave?

Who's actually writing your SQL?

Developers who learned just enough

ORMs generating queries you've never seen

LLMs!

84% of developers now use LLMs in some way

LLM might solve the part of problem
Ask for a query. Get a query.

So what's the problem?

Rate of change might accelerate

But LLM does not see it all

And at least I don't have
to talk bad about SWE

A simple change

Before

```
SELECT id, user_id, status, total_amount, created_at
FROM orders
WHERE created_at > now() - interval '1 day'
LIMIT 50;
```

After

```
SELECT id, user_id, status, total_amount, created_at
FROM orders
WHERE created_at > now() - interval '1 day'
ORDER BY total_amount DESC
LIMIT 50;
```

Looks reasonable. Ship it.

**Your tests
passed! Ship it**

What actually happened

```
Limit (cost=32293.96..32299.78 rows=50 width=31) (actual time=98.631..103.719 rows=50.00 loops=1)
  Buffers: shared hit=14479 read=2258 written=26
  -> Gather Merge (cost=32293.96..32694.49 rows=3439 width=31) (actual time=98.628..103.714 rows=50.00 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    Buffers: shared hit=14479 read=2258 written=26
    -> Sort (cost=31293.94..31297.52 rows=1433 width=31) (actual time=75.454..75.454 rows=40.33 loops=3)
      Sort Key: total_amount DESC
      Sort Method: top-N heapsort Memory: 31kB
      Buffers: shared hit=14479 read=2258 written=26
      Worker 0: Sort Method: top-N heapsort Memory: 31kB
      Worker 1: Sort Method: top-N heapsort Memory: 30kB
      -> Parallel Seq Scan on orders (cost=0.00..31246.33 rows=1433 width=31) (actual
time=0.399..75.174 rows=926.33 loops=3)
        Filter: (created_at > (now() - '1 day'::interval))
        Rows Removed by Filter: 665740
        Buffers: shared hit=14405 read=2258 written=26

Planning Time: 0.348 ms
Execution Time: 103.796 ms
```

SQL is fundamentally different

With Python, what you write is what runs

```
def get_user(id):  
    for user in all_users:  
        if user.id == id:  
            return user
```

With SQL, you write WHAT, not HOW

```
SELECT * FROM users WHERE id = $1;
```

For that one query, the planner might choose:

- Index Scan on primary key
- Index Scan on a different index
- Bitmap Index Scan
- Sequential Scan
- Parallel Sequential Scan

The planner is smart

It looks at table statistics, available indexes, memory settings, data patterns

And it makes the best choice.

PostgreSQL is very good at guessing - usually 🤖

Same query, different reality

Factor	Your Laptop	CI	Production
Rows	100	1,000	10,000,000
Distribution	Uniform	Uniform	Heavily skewed
Memory	16GB	4GB	128GB

**It works on
my database**

We solved this for code

"It works on my machine" → Docker

"It works in staging" → Feature flags,
gradual rollouts

"It works with my data" → ???

What about code review?

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?

How many rows will it scan?

You can't tell from reading SQL

"Just run EXPLAIN before merging"

We need to test two things

Logical correctness: Does it return the right data?

Performance correctness: Does it return them efficiently?

PERFORMANCE

What you going
to measure?

The timing trap

"Just fail the test if it's slow!"

Run	Time
1	154 ms
2	119 ms
3	105 ms

COST?

Cost is the planner's pre-execution estimate based on statistics and cost parameters

BUFFERS FTW!

**Buffers, reported with
ANALYZE, reflect actual pages
accessed during execution**

Expect that one little thing...

Buffers don't lie

Machine	Time	Buffers
Hetzner ARM64	154 ms	24,108
Hetzner AMD64	119 ms	24,108
M4 Pro VM	98 ms	24,108

Why not cost? that's estimate

**If your query
suddenly reads just
10% more buffers,
that's a regression**

**If you have reproducible
start point (data)**

Introducing RegreSQL

<https://github.com/boringsql/regresql>

SQL Query regression
testing framework

Getting started

```
regresql init postgres://regresql:can123do@192.168.139.28/my_demo
```

regresql/regress.yaml

```
-- default
root: .
pguri: postgres://regresql:can123do@192.168.139.28/my_demo

-- optional configuration
snapshot:
  path: snapshots/test_data.dump
  schema: regresql/schema.sql

analyze:
  enabled: true
```

Flow

- regresql fixturize
- regresql snapshot
- regresql discover (thing `git status`)
- regresql add/remove
- edit plan in your favourite editor (hx)
- regresql update -- generate `expected` data (reference)
- regresql baseline -- generate `baseline` (reference)
- regresql test

discover

```
regresql discover
SQL files in project:
  [ ] internal/services/auth/create_provider_link.sql (1 query)
  [ ] internal/services/auth/delete_failed_login_attempt_by_user_id.sql
(1 query)
  [ ] internal/services/auth/delete_provider_link.sql (1 query)
  [ ] internal/services/auth/get_provider_link.sql (1 query)
  [+]
internal/services/auth/select_failed_login_attempt_by_user_email.sql (1
query)
  [ ] internal/services/auth/select_failed_login_attempt_by_user_id.sql
(1 query)
  [ ] internal/services/auth/upsert_failed_login_attempt.sql (1 query)
```

Plan

```
# Test cases for select_failed_login_attempt_by_user_email query
# Parameters: $1=base_lockout_time, $2=lockout_multiplier, $3=email

# User with failed login attempts - should return user_id, attempts, locked_until
user_with_failed_attempts:
  arg1: 5          # base_lockout_time in seconds
  arg2: 2          # lockout_multiplier (exponential)
  arg3: "user0@example.com"

# User without failed login attempts - should return empty result
user_without_failed_attempts:
  arg1: 5
  arg2: 2
  arg3: "user5001@example.com" # index beyond failed_login_attempts count

# Case insensitive email lookup - should still find the user
case_insensitive_email:
  arg1: 5
  arg2: 2
  arg3: "USER0@EXAMPLE.COM"

# Nonexistent user email - should return empty result
nonexistent_user:
  arg1: 5
  arg2: 2
  arg3: "nobody@nowhere.test"
```

RESULTS:

✓ 8 passing
0.00s total

WARNINGS:

select_failed_login_attempt_by_user_email_select_failed_login_attempt_by_user_email.case_insensitive_email.buffer (56 <= 56 * 102%)

⚠ Sequential scan detected on table 'user_account'

Suggestion: Consider adding an index if this table is large or this query is frequently executed

⚠ Nested loop join with sequential scan detected

Suggestion: Add index on join column to avoid repeated sequential scans

select_failed_login_attempt_by_user_email_select_failed_login_attempt_by_user_email.nonexistent_user.buffer (56 <= 56 * 102%)

⚠ Sequential scan detected on table 'user_account'

Suggestion: Consider adding an index if this table is large or this query is frequently executed

⚠ Nested loop join with sequential scan detected

Suggestion: Add index on join column to avoid repeated sequential scans

select_failed_login_attempt_by_user_email_select_failed_login_attempt_by_user_email.user_with_failed_attempts.buffer (56 <= 56 * 102%)

⚠ Sequential scan detected on table 'user_account'

Suggestion: Consider adding an index if this table is large or this query is frequently executed

⚠ Nested loop join with sequential scan detected

Suggestion: Add index on join column to avoid repeated sequential scans

select_failed_login_attempt_by_user_email_select_failed_login_attempt_by_user_email.user_without_failed_attempts.buffer (56 <= 56 * 102%)

⚠ Sequential scan detected on table 'user_account'

Suggestion: Consider adding an index if this table is large or this query is frequently executed

⚠ Nested loop join with sequential scan detected

Suggestion: Add index on join column to avoid repeated sequential scans

What data to test?

I thought that's easy
part; turns out it's NOT

Fixturize

github.com/boringSQL/fixturize

Production query plans without production data

`pg_restore_relation_stats`

`pg_restore_attribute_stats`

<https://boringsql.com/posts/portable-stats/>

pg_regresql extension

```

psql -t -A -c "
-- Relation stats
SELECT format(
  'SELECT pg_restore_relation_stats(''schemaname'', %L, ''relname'', %L, ''relpages'', %s, ''reltuples'', %s::real, ''relallvisible'', %s);',
  n.nspname,
  c.relname,
  c.relpages,
  c.reltuples,
  c.relallvisible
)
FROM pg_class c
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE c.relkind IN ('r', 'm')
      AND n.nspname NOT IN ('pg_catalog', 'information_schema', 'pg_toast')

UNION ALL

-- Attribute stats
SELECT format(
  'SELECT pg_restore_attribute_stats(''schemaname'', %L, ''relname'', %L, ''attname'', %L, ''inherited'', %L::boolean, ''null_frac'', %s::real,
  ''avg_width'',
  %s,
  ''n_distinct'', %s::real);',
  s.schemaname,
  s.tablename,
  s.attname,
  s.inherited,
  s.null_frac,
  s.avg_width,
  s.n_distinct
)
FROM pg_stats s
WHERE s.schemaname NOT IN ('pg_catalog', 'information_schema', 'pg_toast');
" accounts_production > /tmp/stats.sql

```

test --stats stats.sql

RegreSQL 2.0

v2.0.0-alpha5

full release in coming weeks

What about ORMs?

The N+1 in the room

regresql-activererecord - NEXT

ETA Q1/2026

```
# rake regresql:export
Regresql: Exporting queries to /Users/radim/tmp/rails-demo

Define exports in lib/tasks/regresql.rake:

namespace :regresql do
  task export: :environment do
    # automatic plan generation
    Regresql.export('users/recent', User.where('created_at > ?', 1.day.ago))

    # manual bindings
    Regresql.export('users/active',
      User.where(active: true)
    ) do |q|
      q.binding :default, active: true
      q.binding :inactive, active: false
    end
  end
end

Regresql: Exporting queries to /Users/radim/tmp/rails-demo

Exported: /Users/radim/tmp/rails-demo/posts/published.sql
Exported: /Users/radim/tmp/rails-demo/posts/by_tag.sql
Exported: /Users/radim/tmp/rails-demo/regresql/plans/posts/by_tag_by_tag.yaml
Exported: /Users/radim/tmp/rails-demo/posts/by_category.sql
Exported: /Users/radim/tmp/rails-demo/regresql/plans/posts/by_category_by_category.yaml
Exported: /Users/radim/tmp/rails-demo/posts/with_comment_counts.sql
Exported: /Users/radim/tmp/rails-demo/authors/with_post_counts.sql
Exported: /Users/radim/tmp/rails-demo/tags/popular.sql
Exported: /Users/radim/tmp/rails-demo/posts/with_all_tags__post.sql
Exported: /Users/radim/tmp/rails-demo/posts/with_all_tags__tagging.sql
Exported: /Users/radim/tmp/rails-demo/posts/with_all_tags__tag.sql
Exported: /Users/radim/tmp/rails-demo/comments/recent.sql
Exported: /Users/radim/tmp/rails-demo/categories/hierarchy.sql
Exported: /Users/radim/tmp/rails-demo/posts/search.sql
Exported: /Users/radim/tmp/rails-demo/regresql/plans/posts/search_search.yaml
Exported: /Users/radim/tmp/rails-demo/posts/by_date_range.sql
Exported: /Users/radim/tmp/rails-demo/regresql/plans/posts/by_date_range_by_date_range.yaml
Exported: /Users/radim/tmp/rails-demo/authors/top_contributors.sql
```

**What ORMs
do you use?**

Beyond?

- Query capture, ORM support, more CI/CD, automations
- PlanAdvisors integration
- better developer experience
- Improve the contract governance aspect of the tool

Thank you!

Find 1 - 4 queries to start
with, and let me know
radim@boringsql.com