Logging with Purpose: A Framework for Finding and Fixing Slow Queries in Postgres

Ryan Booz

Solutions Engineer
pganalyze

@ryanbooz

/in/ryanbooz

www.softwareandbooz.com

youtube.com/@ryanbooz

github.com/ryanbooz

Really? Another configuration talk?

Another configuration talk?

Postgres has many internal tools for **observability**

Learning how to use them is still **surprisingly hard**

~400* settings in a default install of Postgres 18

*I did a simple COUNT of pg_settings, not deep analysis. ;-)

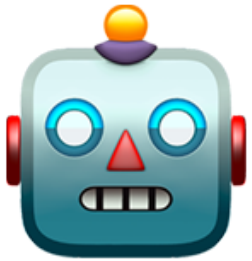
Another configuration talk?

Teams still **struggle** to find slow queries

Either **too little signal**… or **too much noise**

Another configuration talk?

pganalyze



AI still can’t solve solve all these configuration problems...

… at least not yet

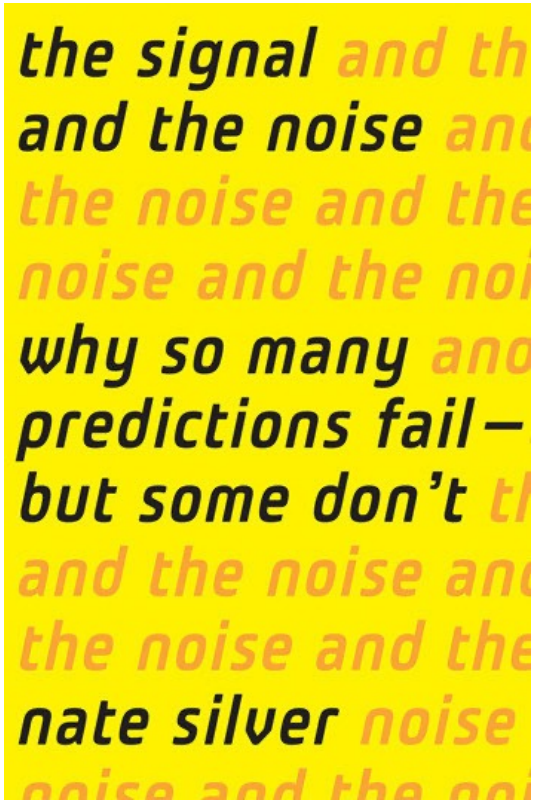
Tuning Framework

- Revisit the basics
- Offer some guiding principles
- Figure out what's missing (audience participation 😉)

The signal and the noise

pganalyze

Unrelated, but still a good book.



The signal and the noise

pganalyze

Under-instrumentation = blind to problems

- `log_min_error_statement` = `[FATAL|PANIC]`
- `log_min_duration_statement` = `-1`
- `pg_stat_statements` not installed
- `auto_explain` not enabled
- `track_io_timing` not enabled*

It's like hearing Crickets?

*YMMV

The signal and the noise

pganalyze

Over-instrumentation = noise everywhere

- `log_statement` = all
- `log_min_duration_statement` = [0-100]*
- `pg_stat_statements.max` = 50,000-100,00*
- `auto_explain.log_min_duration` = [0-100]*
- `log_duration` = on
- `pg_audit` is enabled broadly

Overwhelmed by the flood of information!

*YMMV

pganalyze

pganalyze

Find queries worth fixing

(Not every slow query matters)

Quick Disclaimer:

(primarily for those reading the slides without in-person context)

The following is based on experience, and primarily, observations while helping other Postgres users and customers over the years. Some you may agree with - and some you may not. That's totally OK! Let's engage and help me make the information better.

Primarily this is about bringing information to the surface for many users that just won't hear or think about it in normal day-to-day operations (especially if they are primarily developers).

A rising tide lifts all ships… right? 😃

log_min_error_statement = [FATAL/PANIC]

"All statements generating an error at or above this level"

- Often "turned up" when an app is in error and lots of similar errors are filling logs
- Misses "why is this query failing?" questions unnecessarily

Recommendation: Keep at ERROR and fix your application

log_min_duration_statement = -1

"Minimum execution time above which all statements will be loggged"

- Hosted environments don't turn this on by default
- Often missed configuration with high upside value
- Only requires a reload, not a restart

pganalyze

log_min_duration_statement = [0-100]*

- Fastest way to find potentially problematic queries
- Too low and logs become overwhelmed with low-value queries
- Thoughtfully consider your initial threshold

Recommendation: One approach is to start with a value that looks at the distribution of your current timings in pg_stat_statements – or 1,000(ms) if you don't know and tune up/down as necessary

*YMMV

Find a reasonable starting value (one approach)


```
SELECT
  threshold_ms,
  SUM(calls) FILTER (WHERE mean_exec_time > threshold_ms) AS queries_logged,
  ROUND(100.0 * SUM(calls) FILTER (WHERE mean_exec_time > threshold_ms) /
     NULLIF(SUM(calls), 0), 6) AS pct_of_traffic
FROM pg_stat_statements,
(VALUES (50), (100), (200), (400), (1000), (2000)) AS t(threshold_ms)
WHERE calls > 10
GROUP BY threshold_ms
ORDER BY threshold_ms;
```

Modify as necessary based on what you find

threshold_ms	queries_logged	pct_of_traffic
1	187	0.003529
5	173	0.003265
10	173	0.003265
50	65	0.001227
100	65	0.001227
500	65	0.001227

pganalyze

track_io_timing = off

"Collects timing statistics for database I/O activity"

- Distinguishes CPU vs. I/O-bound queries
- More helpful EXPLAIN (ANALYZE,BUFFERS) plans
  - Shows read/write times of buffers
- Better reasoning with pg_stat_statements
  - Lower values = CPU-bound
  - Higher values = I/O-bound
- Does incur overhead, but typically far less than feared (1-5%)*

Recommendation: Turn it on with proper monitoring to verify load

*YMMV

pg_stat_statements not installed

- Most hosted providers don't include in **preload_shared_libraries**
- Primary tool to surface patterns when sampled regularly
- **Cumulative values** requires tooling to sample deltas over time
- Only path to get full text of executed queries
- **Requires a restart**
  - Plan accordingly
  - Make it a default part of your configuration settings

pganalyze

pg_stat_statements.max = 20,000-50,000+*

"Maximum number of unique statement texts stored and tracked"

- <=PG18 are stored in text file on disk
  - PG19 may move some of this to memory (🤞)
- Slow to read and maintain the larger it grows
- Can the app be modified to improve unique query counts?
- Modify IN queries to ANY? (<=PG17)
  - PG18+ changes "IN (1,2,3)" into "ANY('{1,2,3}')" automatically!

*YMMV

pg_stat_statements recommendation

- Enable it – no questions
- Initial settings:
  - pg_stat_statements.max = 10,000
  - pg_stat_statements.track = all
  - track_io_timing = on (discussed earlier)
- Monitor the "dealloc" column in **pg_stat_statements_info** view
  - If it goes up, raise pg_stat_statements.max – until you're in the 30k-50k range
  - Above that, consider spreading out DBs to multiple servers or rewriting queries with ANY, etc.

auto_explain not installed and configured

- (Most) hosted providers don't include in **preload_shared_libraries**
- Only method for getting runtime EXPLAIN of queries
- Impossible to answer "why" questions after the fact
- Can't track plan changes over time for the same query
- Aurora 14.10, 15.5, and later
  - Enable `aurora_compute_plan_id`
- Postgres 16+
  - pg_stat_plans extension is a new way to track per-plan statistics
  - https://github.com/pganalyze/pg_stat_plans

pganalyze

auto_explain.log_min_duration = [0-100]*

"Log EXPLAIN plans when execution time is above threshold (ms)"

- auto_explain must be enabled in `preload_shared_libraries`
  - requires a restart
- Like `log_min_statement_duration`, too low will overwhelm logs with low-value plans
- Best set with some guidance from pg_stat_statements and appropriate monitoring software

*YMMV

auto_explain recommendations

- Aurora 14.10, 15.5, and later
  - Enable `aurora_compute_plan_id`
- Baseline settings:
  - auto_explain.format = json
  - auto_explain.log_min_duration = 1,000
  - auto_explain.log_analyze = on
  - auto_explain.log_buffers = on
  - auto_explain.log_verbose = on
  - auto_explain.log_nested_statements = on
  - auto_explain.log_timing = off
  - auto_explain.sample_rate = 1

pganalyze

log_statement = all

“Logs all statements”

- Logs all statements to Postgres log
- Significant increase in log volume
- Generally, no actionable value

Recommendation: NEVER set to ‘all’. Use either ‘none’ or maybe ‘ddl’

pganalyze

log_duration = on

"Logs duration of all statement executions"

- By itself doesn't provide much benefit or value
- Other means for logging statements or EXPLAIN provide better visibility

Recommendation: always keep "off"

pgAudit = broadly configured

- Increasingly used with regulatory requirements (SOC2,HIPPA,etc.)
- Not just, "*this query ran*", but "*this user accessed these specific tables/columns with this operation type*"
- Increased log volume
- Few monitoring solutions currently provide parsing capabilities
- Consider focused, object-level auditing

Global Configuration Considerations

Global Configuration Consideration

pganalyze

Premature Optimization = Unnecessary Resource usage

- `autovacuum_vacuum_scale_factor` < 0.1 (0.2)
- `autovacuum_analyze_scale_factor` < 0.05 (0.1)
- `default_statistics_target` > 200 (100)
- `max_connections` > 1,000 (100)
- `work_mem` >16MB (4MB)

(default in parenthesis)

autovacuum_vacuum_scale_factor <= 0.10

- While more frequent vacuum is often better, too frequent on all tables steals resources
- Added contention with planning, locks, and scanning visibility maps
- Monitoring software may interpret lots of vacuum activity as unhealthy table bloat rather than improper settings

Recommendation: Keep at 20% and tune specific tables

autovacuum_analyze_scale_factor <= 0.05

- More frequent analyze runs doesn't typically improve chosen plan for most queries
- Resource usage is correlated with table width, index patterns, etc.
- When everything is "up-to-date" artificially, outliers become harder to spot

Recommendation: Keep at 10% and tune specific tables

default_statistics_target > 200

- Defaults to 100
- Some other platforms (SQL Server) use 200, but it's not configurable
- Generally unproven for most workloads to increase
- Focus on per-column updates for large DW situations
- Often using focused extended (correlated) statistics would serve you better

Recommendation: Don't modify globally and target specific high-value query issues

max_connections >= 1,000

- Application connection issues usually precipitate increases
- Recall that memory usage is a function of `connections/queries * work_mem`
- Focus on a connection pooler instead (assuming your application is compatible)
  - pgbouncer
  - pgdog
  - Pgpool-II

Recommendation: at least 200 and maybe up to 1,000 to start

work_mem >= 16MB

"Sets the maximum memory to be used for query workspaces"

- Default of 4MB is often too low for modern workloads and hardware
- Default is often the "silent" query performance killer without effective monitoring
- Query memory usage is a function of `queries/connections * work_mem`
- Monitor for "external" operations in EXPLAIN (or pg_stat_statements)

Recommendation: 8-16MB if you have at least a few GB of RAM

Query examples

External Operations


```
EXPLAIN (ANALYZE,buffers)
SELECT r.rental_id, r.customer_id, r.store_id, r.inventory_id,
       p.amount, p.payment_date, c.full_name, c.email
FROM bluebox.rental r
JOIN bluebox.payment p ON p.rental_id = r.rental_id
JOIN bluebox.customer c ON c.customer_id = r.customer_id
WHERE r.store_id <= 20
   OR c.store_id >= 100 AND c.store_id <= 105
ORDER BY p.amount DESC, r.customer_id, p.payment_date;
```

Using EXPLAIN plans


```
QUERY PLAN
-----------------------------------------------------------------------------------------------+
Gather Merge  (cost=392889.74..489751.53 rows=830186 width=75) (actual time=2178.674..2419.667
rows=1020401 loops=1)                                                                          |
  Workers Planned: 2
  Workers Launched: 2
  Buffers: shared hit=119728 read=49379, temp read=65755 written=65843
  ->  Sort  (cost=391889.71..392927.45 rows=415093 width=75) (actual time=2142.694..2203.032 rows=340134
loops=3)
        Sort Key: p.amount DESC, r.customer_id, p.payment_date
        Sort Method: external merge  Disk: 29144kB
        Buffers: shared hit=119728 read=49379, temp read=65755 written=65843
          Worker 0:  Sort Method: external merge  Disk: 31176kB
          Worker 1:  Sort Method: external merge  Disk: 30408kB
        ->  Parallel Hash Join  (cost=160672.20..334710.15 rows=415093 width=75) (actual
             time=1655.021..1926.128 rows=340134 loops=3)
              Hash Cond: (p.rental_id = r.rental_id)
```

Using pg_stat_statements


```
SELECT calls, mean_exec_time,
       temp_blks_read, temp_blks_written
FROM pg_stat_statements
WHERE queryid = 1212311850664953035;
```

```
Name              |Value     |
------------------+----------+
calls             |1         |
mean_exec_time    |1965.98996|
temp_blks_read    |65754     |
temp_blks_written|65847     |
```

Using pg_stat_statements


```
work_mem_needed = current_work_mem +
        (temp_blks_written × 8KB / num_workers)
```

```
SELECT pg_size_pretty(4098 + (65847*8192/2)::numeric) = 247MB
```

Name	Value
calls	1
mean_exec_time	1965.98996
temp_blks_read	65754
temp_blks_written	65847


```
SET work_mem='256MB'

EXPLAIN (ANALYZE,buffers)
SELECT r.rental_id, r.customer_id, r.store_id, r.inventory_id,
       p.amount, p.payment_date, c.full_name, c.email
FROM bluebox.rental r
JOIN bluebox.payment p ON p.rental_id = r.rental_id
JOIN bluebox.customer c ON c.customer_id = r.customer_id
WHERE r.store_id <= 20
   OR c.store_id >= 100 AND c.store_id <= 105
ORDER BY p.amount DESC, r.customer_id, p.payment_date;
```


```
-- Set role specific memory for targeted jobs
ALTER ROLE nightly_etl_role SET work_mem = '256MB';

-- Later...
EXPLAIN (ANALYZE,buffers)
SELECT r.rental_id, r.customer_id, r.store_id, r.inventory_id,
       p.amount, p.payment_date, c.full_name, c.email
FROM bluebox.rental r
JOIN bluebox.payment p ON p.rental_id = r.rental_id
JOIN bluebox.customer c ON c.customer_id = r.customer_id
WHERE r.store_id <= 20
   OR c.store_id >= 100 AND c.store_id <= 105
ORDER BY p.amount DESC, r.customer_id, p.payment_date;
```

Wrong Index Due to ORDER BY

- The planner chooses an index to satisfy ORDER BY
- Requires scanning many rows before filtering
- A better plan would filter first, then sort the small result.

Fix: Add "+0" to the ORDER BY column to disable index usage

- Look for:
  - `Index scan on PK when you have filtered columns`
  - `Filter (not Index Cond)`
  - `Lots of rows removed`

Inefficient Nested Loop

- A correlated subquery forces the planner to re-execute the subquery for every row in the outer query
- This causes repeated scans of the same tables.

Fix: Use a MATERIALIZED CTE to compute the result once

- Look for:
  - "SubPlan" with high "loops=" count
  - Same table being scanned repeatedly inside the loop
  - Massive buffer counts relative to actual rows returned
  - Correlation: subquery references columns from outer query

🎉 THANK YOU! 🎉

What Questions do you have?

Ryan Booz

Solutions Engineer
pganalyze

@ryanbooz

/in/ryanbooz

www.softwareandbooz.com

youtube.com/@ryanbooz