



Postgres Tuning in the Age of AI: Context Is Everything



Ryan Booz

Solutions Engineer, pganalyze · @ryanbooz · pganalyze.com

Someone added a store-average column to the customer list



```
# reports/customers.py    (added in the Q3 release)
```

```
store_avg = (Rental.objects
    .filter(customer__store_id=OuterRef("store_id"),          # correlate to the row
            rental_period__startswith__gte=cutoff,          # repeat the date filter
            inventory__film__overview__icontains=term)      # repeat the film filter
    .order_by().values("customer__store_id")
    .annotate(store_revenue=Sum("payment__amount"),
              customers=Count("customer_id", distinct=True))
    .annotate(avg_spend=ExpressionWrapper(F("store_revenue") / F("customers"),
                                          output_field=DecimalField()))
    .values("avg_spend")[:1])
```

```
qs = (Customer.objects
    .filter(rentals__rental_period__startswith__gte=cutoff,
            rentals__inventory__film__overview__icontains=term)
    .annotate(rental_count=Count("rentals"),
              revenue=Sum("rentals__payment__amount"),
              store_avg_spend=Subquery(store_avg))
    .filter(rental_count__gt=4)
    .order_by("-revenue")[:200])
```

store_avg_spend: what a typical customer at this customer's own store spent, same window, same films.
The filters appear twice because they have to. OuterRef carries the correlation, not the WHERE clause.

Which resulted in a slow query

```
SELECT c.customer_id, c.full_name, c.store_id,
       count(r.rental_id) AS rentals,
       sum(p.amount)      AS revenue,
       max(upper(r.rental_period))::date AS last_return,
       (SELECT avg(spend) FROM (
         SELECT sum(p2.amount) AS spend
         FROM bluebox.rental    r2
         JOIN bluebox.customer  c2 ON c2.customer_id = r2.customer_id
         JOIN bluebox.payment   p2 ON p2.rental_id   = r2.rental_id
         JOIN bluebox.inventory i2 ON i2.inventory_id = r2.inventory_id
         JOIN bluebox.film      f2 ON f2.film_id    = i2.film_id
         WHERE c2.store_id = c.store_id
              AND lower(r2.rental_period) >= now() - interval '2 years'
              AND f2.overview ILIKE '%adventure%'
         GROUP BY r2.customer_id) sc) AS store_avg_spend
FROM bluebox.customer  c
JOIN bluebox.rental    r ON r.customer_id = c.customer_id
JOIN bluebox.payment   p ON p.rental_id   = r.rental_id
JOIN bluebox.inventory i ON i.inventory_id = r.inventory_id
JOIN bluebox.film      f ON f.film_id    = i.film_id
WHERE lower(r.rental_period) >= now() - interval '2 years'
      AND f.overview ILIKE '%adventure%'
GROUP BY c.customer_id, c.full_name, c.store_id
HAVING count(r.rental_id) > 4
ORDER BY revenue DESC, last_return DESC
LIMIT 200;
```

Which resulted in a slow query

```
SELECT c.customer_id, c.full_name, c.store_id,
       count(r.rental_id) AS rentals,
       sum(p.amount)      AS revenue,
       max(upper(r.rental_period))::date AS last_return,
       (SELECT avg(spend) FROM (
         SELECT sum(p2.amount) AS spend
         FROM bluebox.rental    r2
         JOIN bluebox.customer  c2 ON c2.customer_id = r2.customer_id
         JOIN bluebox.payment   p2 ON p2.rental_id   = r2.rental_id
         JOIN bluebox.inventory i2 ON i2.inventory_id = r2.inventory_id
         JOIN bluebox.film      f2 ON f2.film_id    = i2.film_id
        WHERE c2.store_id = c.store_id
              AND lower(r2.rental_period) >= now() - interval '2 years'
              AND f2.overview ILIKE '%adventure%'
        GROUP BY r2.customer_id) sc) AS store_avg_spend
FROM bluebox.customer c
JOIN bluebox.rental    r ON r.customer_id = c.customer_id
JOIN bluebox.payment   p ON p.rental_id   = r.rental_id
JOIN bluebox.inventory i ON i.inventory_id = r.inventory_id
JOIN bluebox.film      f ON f.film_id    = i.film_id
WHERE lower(r.rental_period) >= now() - interval '2 years'
      AND f.overview ILIKE '%adventure%'
GROUP BY c.customer_id, c.full_name, c.store_id
HAVING count(r.rental_id) > 4
ORDER BY revenue DESC, last_return DESC
LIMIT 200;
```

Which resulted in a slow query

```
SELECT c.customer_id, c.full_name, c.store_id,
       count(r.rental_id) AS rentals,
       sum(p.amount)      AS revenue,
       max(upper(r.rental_period))::date AS last_return,
       (SELECT avg(spend) FROM (
         SELECT sum(p2.amount) AS spend
         FROM bluebox.rental    r2
         JOIN bluebox.customer  c2 ON c2.customer_id = r2.customer_id
         JOIN bluebox.payment   p2 ON p2.rental_id   = r2.rental_id
         JOIN bluebox.inventory i2 ON i2.inventory_id = r2.inventory_id
         JOIN bluebox.film      f2 ON f2.film_id     = i2.film_id
         WHERE c2.store_id = c.store_id
              AND lower(r2.rental_period) >= now() - interval '2 years'
              AND f2.overview ILIKE '%adventure%'
         GROUP BY r2.customer_id) sc) AS store_avg_spend
FROM bluebox.customer  c
JOIN bluebox.rental    r ON r.customer_id = c.customer_id
JOIN bluebox.payment   p ON p.rental_id   = r.rental_id
JOIN bluebox.inventory i ON i.inventory_id = r.inventory_id
JOIN bluebox.film      f ON f.film_id     = i.film_id
WHERE lower(r.rental_period) >= now() - interval '2 years'
      AND f.overview ILIKE '%adventure%'
GROUP BY c.customer_id, c.full_name, c.store_id
HAVING count(r.rental_id) > 4
ORDER BY revenue DESC, last_return DESC
LIMIT 200;
```

Which resulted in a slow query

```
SELECT c.customer_id, c.full_name, c.store_id,
       count(r.rental_id) AS rentals,
       sum(p.amount)      AS revenue,
       max(upper(r.rental_period))::date AS last_return,
       (SELECT avg(spend) FROM (
         SELECT sum(p2.amount) AS spend
         FROM bluebox.rental    r2
         JOIN bluebox.customer  c2 ON c2.customer_id = r2.customer_id
         JOIN bluebox.payment   p2 ON p2.rental_id   = r2.rental_id
         JOIN bluebox.inventory i2 ON i2.inventory_id = r2.inventory_id
         JOIN bluebox.film      f2 ON f2.film_id    = i2.film_id
         WHERE c2.store_id = c.store_id
              AND lower(r2.rental_period) >= now() - interval '2 years'
              AND f2.overview ILIKE '%adventure%'
         GROUP BY r2.customer_id) sc) AS store_avg_spend
FROM bluebox.customer  c
JOIN bluebox.rental    r ON r.customer_id = c.customer_id
JOIN bluebox.payment   p ON p.rental_id   = r.rental_id
JOIN bluebox.inventory i ON i.inventory_id = r.inventory_id
JOIN bluebox.film      f ON f.film_id    = i.film_id
WHERE lower(r.rental_period) >= now() - interval '2 years'
      AND f.overview ILIKE '%adventure%'
GROUP BY c.customer_id, c.full_name, c.store_id
HAVING count(r.rental_id) > 4
ORDER BY revenue DESC, last_return DESC
LIMIT 200;
```

Which resulted in a slow query

```
SELECT c.customer_id, c.full_name, c.store_id,
       count(r.rental_id) AS rentals,
       sum(p.amount)      AS revenue,
       max(upper(r.rental_period))::date AS last_return,
       (SELECT avg(spend) FROM (
         SELECT sum(p2.amount) AS spend
         FROM bluebox.rental    r2
         JOIN bluebox.customer  c2 ON c2.customer_id = r2.customer_id
         JOIN bluebox.payment   p2 ON p2.rental_id   = r2.rental_id
         JOIN bluebox.inventory i2 ON i2.inventory_id = r2.inventory_id
         JOIN bluebox.film      f2 ON f2.film_id    = i2.film_id
         WHERE c2.store_id = c.store_id
              AND lower(r2.rental_period) >= now() - interval '2 years'
              AND f2.overview ILIKE '%adventure%'
         GROUP BY r2.customer_id) sc) AS store_avg_spend
FROM bluebox.customer  c
JOIN bluebox.rental    r ON r.customer_id = c.customer_id
JOIN bluebox.payment   p ON p.rental_id   = r.rental_id
JOIN bluebox.inventory i ON i.inventory_id = r.inventory_id
JOIN bluebox.film      f ON f.film_id    = i.film_id
WHERE lower(r.rental_period) >= now() - interval '2 years'
      AND f.overview ILIKE '%adventure%'
GROUP BY c.customer_id, c.full_name, c.store_id
HAVING count(r.rental_id) > 4
ORDER BY revenue DESC, last_return DESC
LIMIT 200;
```

Which resulted in a slow query

```
SELECT c.customer_id, c.full_name, c.store_id,
       count(r.rental_id) AS rentals,
       sum(p.amount)      AS revenue,
       max(upper(r.rental_period))::date AS last_return,
       (SELECT avg(spend) FROM (
         SELECT sum(p2.amount) AS spend
         FROM bluebox.rental    r2
         JOIN bluebox.customer  c2 ON c2.customer_id = r2.customer_id
         JOIN bluebox.payment   p2 ON p2.rental_id   = r2.rental_id
         JOIN bluebox.inventory i2 ON i2.inventory_id = r2.inventory_id
         JOIN bluebox.film      f2 ON f2.film_id    = i2.film_id
         WHERE c2.store_id = c.store_id
              AND lower(r2.rental_period) >= now() - interval '2 years'
              AND f2.overview ILIKE '%adventure%'
         GROUP BY r2.customer_id) sc) AS store_avg_spend
FROM bluebox.customer c
JOIN bluebox.rental    r ON r.customer_id = c.customer_id
JOIN bluebox.payment   p ON p.rental_id   = r.rental_id
JOIN bluebox.inventory i ON i.inventory_id = r.inventory_id
JOIN bluebox.film      f ON f.film_id     = i.film_id
WHERE lower(r.rental_period) >= now() - interval '2 years'
      AND f.overview ILIKE '%adventure%'
GROUP BY c.customer_id, c.full_name, c.store_id
HAVING count(r.rental_id) > 4
ORDER BY revenue DESC, last_return DESC
LIMIT 200;
```

Here's what AI said when all it had was the SQL



```
## 1. **Add Missing Indexes**

-- Index for date range filter on rental_period
CREATE INDEX idx_rental_period_lower ON bluebox.rental
USING BTREE (lower(rental_period));

-- Index for text search on film overview
CREATE INDEX idx_film_overview_trgm ON bluebox.film
USING GIN (lower(overview) gin_trgm_ops);
-- Requires: CREATE EXTENSION IF NOT EXISTS pg_trgm;

-- Composite indexes for joins
CREATE INDEX idx_rental_customer_inventory ON
    bluebox.rental(customer_id, inventory_id, rental_id);
CREATE INDEX idx_payment_rental ON bluebox.payment(rental_id, amount);
CREATE INDEX idx_inventory_film ON bluebox.inventory(inventory_id, film_id);
```

IT DEPENDS!

Previous response was on an earlier, but simpler version of this correlated subquery.

The same response did name the correlated subquery, further down the list.

It was right on the newer version... most of the time



125 runs · five models · one prompt · the SQL and nothing else · median 4 recommendations per answer

what the 125 SQL-only runs did	of 125
"rewrite the correlated subquery" was #1	119
recommended an index somewhere in the list	125
the index was #2	111
the index it wanted on lower(rental_period) already exists, as a BRIN	117

*Two graders read all 500 responses independently; they agree on whether the subquery was #1 in 97% of runs.

In a pinch, we'll often reach for the index first because it (usually) doesn't require an app deploy

But the index was already there – and it wasn't the main issue

Three things this talk is not about



Configuration autotuning

Bayesian optimization over `shared_buffers` and `work_mem` is a different problem with different tools.

Useful, but not this talk

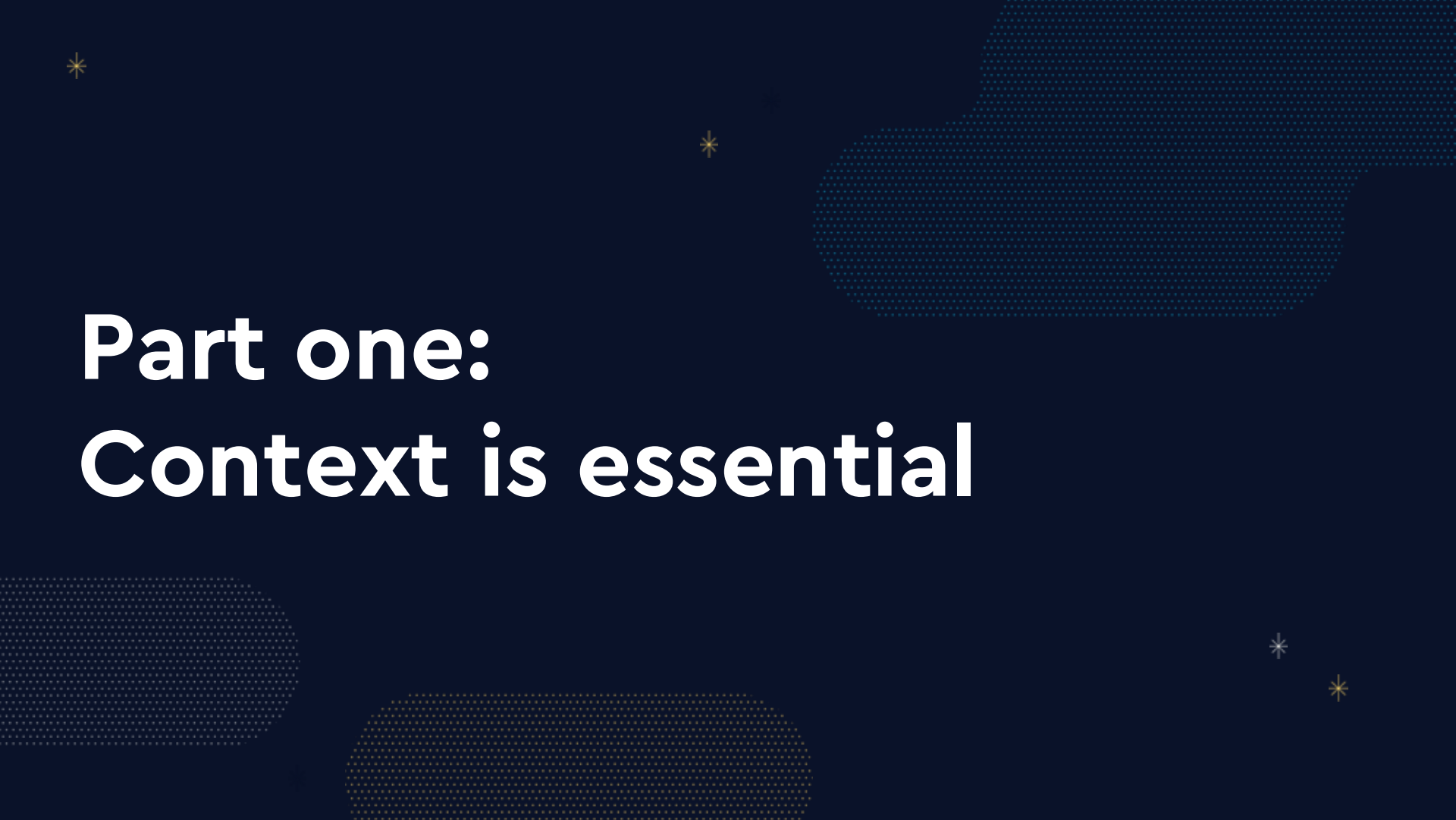
"Will AI replace the DBA?"

It won't, and the interesting question isn't that one. Instead, we're looking at what changes about the work.

MCP security

Real, serious, and thoroughly covered elsewhere.

One hint, though, for production work, don't just use a direct Postgres MCP connection and let AI have at it.



Part one: Context is essential

**~400 settings in a
default install of
Postgres 18.**

The problem was never a shortage of knobs.

The settings

- `log_min_duration_statement = -1`
- `pg_stat_statements` not installed
- `auto_explain` not enabled
- `track_io_timing = off`

What you lose

- No history. You can't tell slow from slower
- No plans. You can't answer "why" after the fact
- No I/O split. CPU-bound and I/O-bound look identical
- No query text. You can't even name the problem

The settings

- `log_statement = all`
- `log_min_duration_statement = 0-100`
- `pg_stat_statements.max = 50,000-100,000`
- `auto_explain.log_min_duration = 0-100`
- `log_duration = on`

What you lose

- Signal buried in volume
- Real cost in I/O and storage
- Dashboards nobody reads
- The same blindness, arrived at from the other direction

Four initial tools for better context



pg_stat_statements

Workload history. What runs, how often, how long.

auto_explain

Runtime plans. The only way to answer "why" after the fact.

track_io_timing

Splits CPU-bound from I/O-bound. **Run pg_test_timing before you enable it**

log_min_duration_statement

The fastest way to find candidates. Reload, not restart.

Two kinds of history

They answer different questions.

Neither substitutes for the other.

pg_stat_statements - sampled over time

- Is this query worth my afternoon?
- Calls x total time, ranked against everything else on the database.
- No plans. No per-execution detail. It can often tell you WHICH query, rarely why.

auto_explain - captured and kept

- Is it always like this?
- One plan you ran by hand is a photograph.
- Fifty captured overnight is a film, and only the film shows a plan change.

Picking a starting "duration" threshold for sampling



```
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;
```

Start where the distribution tells you to, or 1,000ms if you have nothing

Then tune

auto_explain: suggested baseline settings



```
shared_preload_libraries = 'pg_stat_statements,auto_explain'

auto_explain.format           = json
auto_explain.log_min_duration = 1000
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
```

log_timing = off is deliberate

On high-cardinality workloads the timing overhead is the risk, not the logging.

Log statements and EXPLAIN plans?



*PostgreSQL 13 and later. Before 13, log_min_duration_statement is all or nothing.

setting	example	what it does	PG13+	default
log_min_duration_statement	1000	always logged, never sampled		-1
log_min_duration_sample	100	the floor of the sampled band	yes	-1
log_statement_sample_rate	0.05	how much of that band you keep	yes	1.0

- Over 1000 ms: always logged
- Between 100 and 1000 ms: five percent logged
- Under 100 ms: nothing

A snapshot can't see a regression.

Slow queries are often the result of changes:

- a plan flip
- stats drift
- a table that tripled

Point-in-time context is structurally blind to all three.

Part two:
How I'd actually
approach this

- Does this query even matter?
- Is it consistently slow, or only sometimes?
- What shape is the plan, and is it what I expected?
- Where is the time actually going?
- Wrong estimate, or genuinely large?
- Query problem, data problem, stats problem, or config problem?
- What should this change, and how will I know?

Does this query even matter?



```
SELECT calls, mean_exec_time, total_exec_time,  
       100*total_exec_time/sum(total_exec_time) OVER () AS pct, query  
FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 6;
```

calls	mean_ms	pct	query
5803	2,179	24.05	SELECT s.store_id ... ORDER BY random()
285	44,331	24.02	SELECT c.customer_id, c.full_name, ...
7191	1,755	23.99	SELECT s.store_id, count(i.inventory_id)
7079	887	11.94	WITH film_rentals AS (
12020	322	7.36	SELECT f.title, f.rating, f.popularity

Ours ran 285 times

The queries either side of it ran 5,803 and 7,191

All three cost this database the same

Is it consistently slow, or only sometimes?



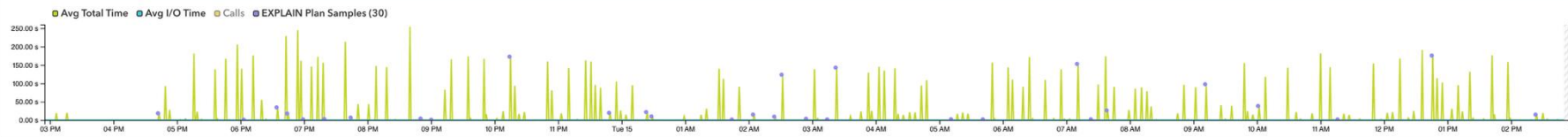
51 captured plans · one query · one evening

plan fingerprint	plans	runtime	cost estimate
07a142bc	20	3.6 s - 283.2 s	13.0 M - 22.0 M
72b33f08	19	1.2 s - 260.1 s	9.1 M - 13.4 M
056d6d04	11	18.8 s - 29.8 s	22.0 M - 22.6 M
338b56e5	1	3.4 s - 3.4 s	0.3 M

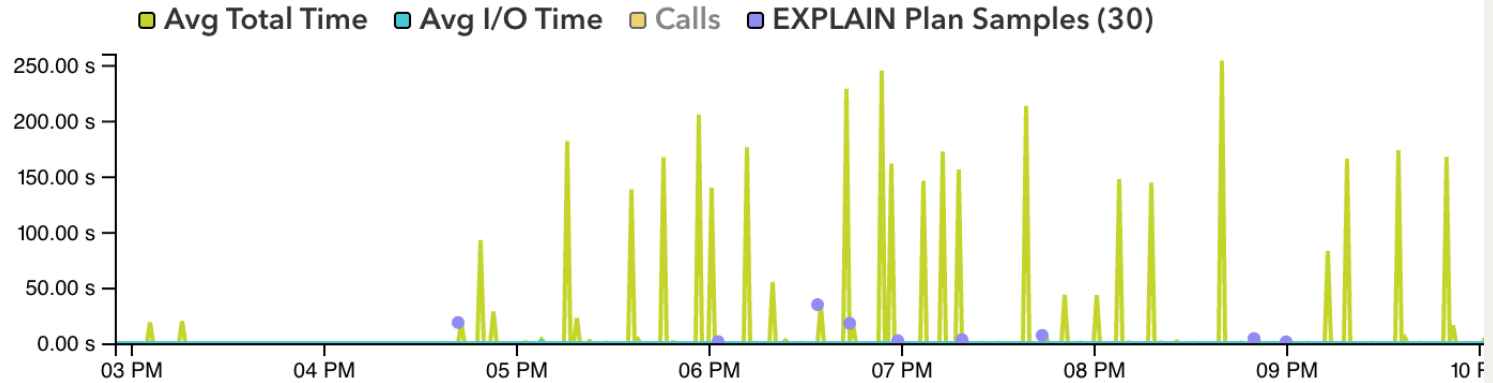
Nothing changed but the word in the ILIKE.

Run EXPLAIN by hand and you get exactly one of these, with no way to know the other three exist.

Avg Time & Calls



Avg Time & Calls



What shape is the plan, and is it what I expected?



```
Limit (actual time=2313.317..164747.520 rows=200 loops=1)
  Buffers: shared hit=105767682, temp read=1712 written=1715
  -> Result (actual time=2313.315..164747.375 rows=200 loops=1)
    -> Sort (actual rows=200 loops=1)
      Sort Method: top-N heapsort  Memory: 69kB
    -> Finalize GroupAggregate (actual rows=569 loops=1)
      Filter: (count(r.rental_id) > 4)
      Rows Removed by Filter: 110846
    -> Gather Merge (actual rows=147601 loops=1)
      Workers Planned: 2  Workers Launched: 2
    -> Sort Method: external merge  Disk: 5320kB
      -> Parallel Hash Join (rows=57542)
        ... inventory / film / rental / payment ...

SubPlan 1
  -> Aggregate (actual time=814.265..814.265 rows=1 loops=200)
    Buffers: shared hit=104670330

Execution Time: 164748.840 ms
```

Where is the time actually going?

Rows, loops, buffers, or spill. Pick one before you read further.

loops=200, and 99% of the buffers



SubPlan 1

```
-> Aggregate (actual time=814.265..814.265 rows=1 loops=200)
    Buffers: shared hit=104670330
    -> GroupAggregate (actual rows=637 loops=200)
        -> Nested Loop (actual rows=993 loops=200)
            -> Hash Join (actual rows=1002 loops=200)
                -> Gather (actual rows=174162 loops=200)
                    Workers Planned: 2    Workers Launched: 2
                    -> Index Scan using rental_inventory_id_idx
                        on rental r2 (loops=15,262,800)
                -> Seq Scan on customer c2 (actual rows=1061 loops=200)
                    Filter: (c2.store_id = c.store_id)
                    Rows Removed by Filter: 185679
```

Whole query: 105,767,682 buffers, 200 rows returned

SubPlan 1: 104,670,330 buffers = 99%

185,679 rows discarded per loop, 200 loops. 15.3 million index scans on rental_inventory_id_idx

Buffers relative to rows returned is the tell, not execution time

The same query, four hours earlier



```
SubPlan 1      -- the 22.2 s capture, same query, same evening
-> Aggregate (actual time=103.939..103.939 rows=1 loops=200)
    Buffers: shared hit=21692195
-> GroupAggregate (actual rows=800 loops=200)
    -> Nested Loop (actual rows=1568 loops=200)
        -> Seq Scan on customer c2 (actual rows=1062 loops=200)
            Rows Removed by Filter: 185678
        -> Index Scan using rental_customer_id_idx
            on rental r2 (rows=16 loops=212,366)
        -> Index Scan using inventory_pk
            on inventory i2 (rows=1 loops=3,310,109)
```

No Gather. It drives from customer, not from inventory.

104 ms per loop instead of 814. 22.2 s instead of 164.7 s.

Same SQL, same server, same evening. The planner rated this one the most expensive of the four, and it is the only one that never got slow.

Wrong estimate, or genuinely large?



worst runtime seen, across 51 captures of one query, in one evening

the plan the planner rated CHEAPEST

260.1 s

cost 9.1 M · 228 M buffers · 19 captures

the plan it rated MOST EXPENSIVE

29.8 s

cost 22.6 M · 23.6 M buffers · 11 captures

The cost estimate went down 60%. The worst runtime went up 9x.

Same statistics, same tables, same night. The estimate pointed the wrong way.

You can only answer it when plans were captured and kept while nobody was watching.

Query problem

The SQL asks for work it doesn't need

Fix: rewrite

Data problem

The work is real and the volume grew

Fix: less data, or accept it

Stats problem

Estimates are wrong, so the plan is wrong

Fix: ANALYZE, extended statistics, targets

Config problem

The plan is right and the memory isn't

Fix: work_mem (global or per-role?)

What should this change, and how will I know?



```
WITH matched AS MATERIALIZED (  
    SELECT r.rental_id, r.customer_id, c.store_id  
    FROM bluebox.rental      r  
    JOIN bluebox.customer  c ON c.customer_id = r.customer_id  
    JOIN bluebox.inventory i ON i.inventory_id = r.inventory_id  
    JOIN bluebox.film      f ON f.film_id     = i.film_id  
    WHERE lower(r.rental_period) >= now() - interval '2 years'  
    AND f.overview ILIKE $1  
) , per_customer AS MATERIALIZED (  
    SELECT m.customer_id, m.store_id,  
           count(*) AS rentals, sum(p.amount) AS revenue  
    FROM matched m JOIN bluebox.payment p ON p.rental_id = m.rental_id  
    GROUP BY 1, 2  
) , store_avg AS MATERIALIZED (  
    SELECT store_id, avg(revenue) AS avg_spend  
    FROM per_customer GROUP BY store_id  
)  
SELECT pc.*, c.full_name, sa.avg_spend  
FROM per_customer pc  
JOIN bluebox.customer c ON c.customer_id = pc.customer_id  
JOIN store_avg      sa ON sa.store_id   = pc.store_id  
WHERE pc.rentals > 4  
ORDER BY pc.revenue DESC LIMIT 200;
```

Faster with no LIMIT effect



same query · same parameters · same minute · LIMIT 200

the original
21,443 ms
23.7 M buffers · cost 22.6 M

the rewrite
3,176 ms
1.0 M buffers · cost 0.34 M

buffers read, by how many rows you asked for	LIMIT 10	LIMIT 50	LIMIT 200
original	3.0 M	7.2 M	23.7 M
rewrite	1,017,498	1,017,498	1,017,498

6.8x faster and 23x fewer buffers on the same parameters

The original's buffers scales with the rows you asked for

It also uncovered the next bottleneck



```
CTE per_customer
```

```
-> HashAggregate (actual time=2870.156..2995.352 rows=140215 loops=1)
```

```
    Planned Partitions: 8    Batches: 9
```

```
    Memory Usage: 8273kB    Disk Usage: 15496kB
```

```
-> Hash Join (actual rows=274340 loops=1)
```

```
    -> Seq Scan on bluebox.payment p
```

```
        (actual time=0.007..478.210 rows=4,796,658)
```

```
    -> Hash Buckets: 131072 Batches: 4 Memory Usage: 5882kB
```

```
temp read=18435 written=21771
```

```
work_mem = 8MB
```

```
3.0 s of the 3.2 s total is right here.
```

The rewrite traded a correlated subquery for a full scan of a 4.8M-row table and a hash aggregate that will not fit in 8MB of work_mem. Nine batches, spilling 15 MB to disk.

This was less visible before. The SubPlan dominated the buffer count and drowned everything else out.

Caution: the proposed fix is conditional



rows returned	original	rewrite	winner
10	2,610 ms	3,006 ms	original
50	6,277 ms	3,038 ms	rewrite
200	21,443 ms	3,176 ms	rewrite

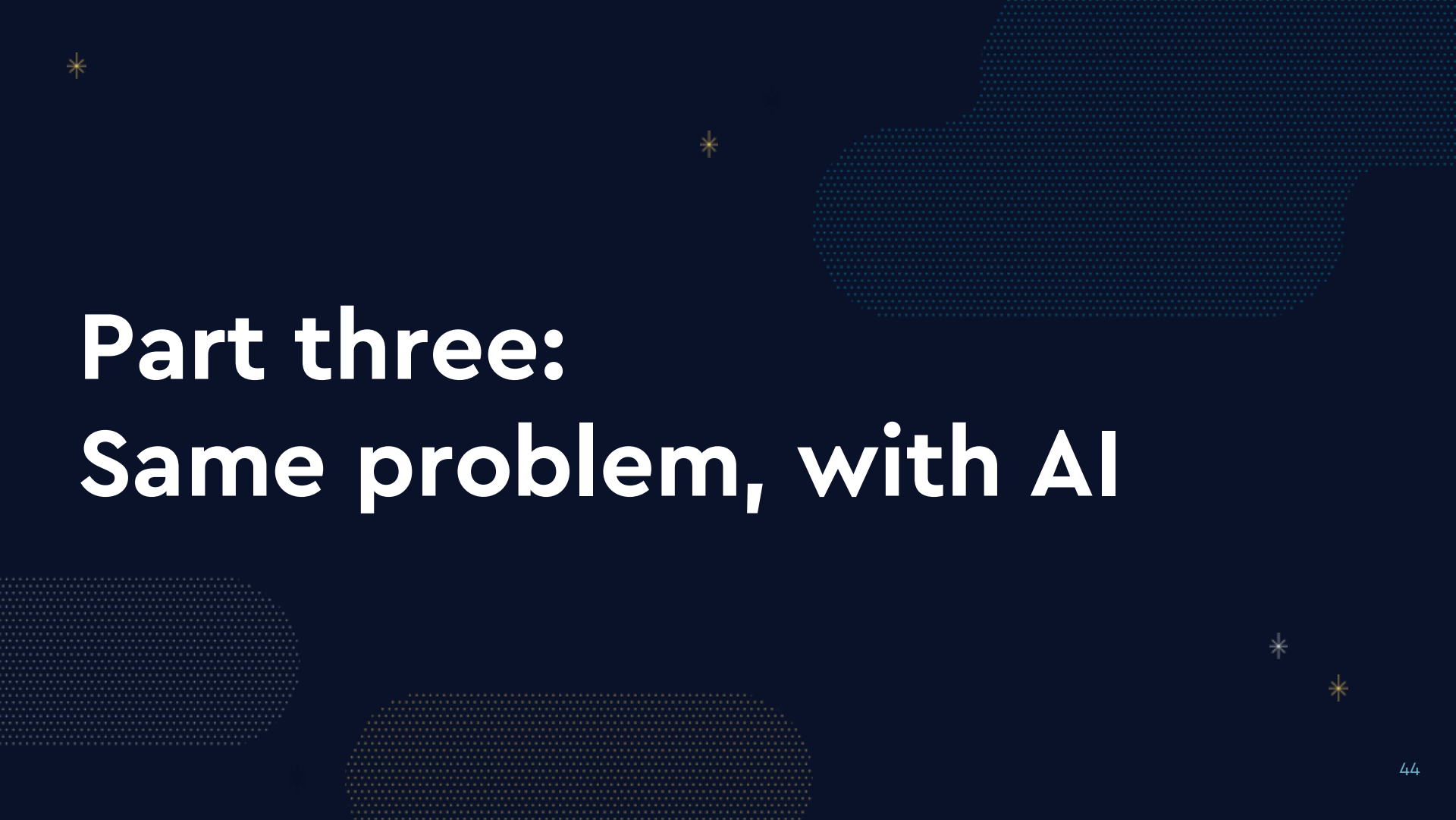
The rewrite pays a fixed cost: scan 4.8M payments, aggregate, spill. About 3 seconds, always.

The correlated subquery pays about 95 ms per output row.

The crossover sits near 15 rows.



**That's not a bad
process... but it's slow
and error prone**



Part three: Same problem, with AI

What most of us typically do

1

I had a slow query.

So I pasted the SQL into a chat window.

2

I added an EXPLAIN ANALYZE.

Better advice. But one plan, run once, by me, just now.

3

I added the schema and indexes.

Better again. It stopped inventing columns.

4

I added pg_stat_statements.

Now it knows the query matters, and how often.

5

I added table stats and config.

Now it can reason about estimates instead of guessing.

Back to the answer we started with

```
## 1. **Add Missing Indexes**

-- Index for date range filter on rental_period
CREATE INDEX idx_rental_period_lower ON bluebox.rental
USING BTREE (lower(rental_period));

-- Index for text search on film overview
CREATE INDEX idx_film_overview_trgm ON bluebox.film
USING GIN (lower(overview) gin_trgm_ops);
-- Requires: CREATE EXTENSION IF NOT EXISTS pg_trgm;

-- Composite indexes for joins
CREATE INDEX idx_rental_customer_inventory ON
    bluebox.rental(customer_id, inventory_id, rental_id);
CREATE INDEX idx_payment_rental ON bluebox.payment(rental_id, amount);
CREATE INDEX idx_inventory_film ON bluebox.inventory(inventory_id, film_id);
```

It started with the premise that "my query is slow" had to do with the WHERE clause and recommended indexes to deal with that alone.

What each additional context brought

+ EXPLAIN ANALYZE

- Found the SubPlan.
Correct diagnosis.
- Still couldn't say whether the query mattered, or whether this plan was typical.

+ schema and indexes

- Stopped asking for the two indexes that were already there. Kept asking for the one that wasn't.
- Started reasoning about join order instead of syntax.

+ pg_stat_statements

- Ranked it against the rest of the workload.
- First point at which it could argue the fix was worth deploying.

Context reduced poor index recommendations



Sonnet 5.5, 25 runs per rung	SQL	+plan	+schema	+pgss	+stats
index on lower(rental_period) already there, as a BRIN	24	16	3	3	0
indexes on the join keys all five already there	25	4	0	4	1
trigram on film.overview this one genuinely isn't there	25	25	23	23	21
raise work_mem and this plan really does spill	7	24	25	24	25

Counts are runs out of 25. Two graders read all 500 responses independently; they agree on 93% of index positions.

It wants an index on lower(rental_period).

That index is already on the table, as a BRIN.

Context reduced poor index recommendations



Sonnet 5.5, 25 runs per rung	SQL	+plan	+schema	+pgss	+stats
index on lower(rental_period) already there, as a BRIN	24	16	3	3	0
indexes on the join keys all five already there	25	4	0	4	1
trigram on film.overview this one genuinely isn't there	25	25	23	23	21
raise work_mem and this plan really does spill	7	24	25	24	25

Counts are runs out of 25. Two graders read all 500 responses independently; they agree on 93% of index positions.

Same story for the join keys.

All five already exist.

Context reduced poor index recommendations



Sonnet 5.5, 25 runs per rung	SQL	+plan	+schema	+pgss	+stats
index on lower(rental_period) already there, as a BRIN	24	16	3	3	0
indexes on the join keys all five already there	25	4	0	4	1
trigram on film.overview this one genuinely isn't there	25	25	23	23	21
raise work_mem and this plan really does spill	7	24	25	24	25

Counts are runs out of 25. Two graders read all 500 responses independently; they agree on 93% of index positions.

Every rung recommends about four things.

What changes is which index it asks for.

The same query, hidden behind a function

what the model was given	recommended it	its rank	the #1 recommendation
the SQL, subquery visible	25 / 25	#1	rewrite the subquery
a function call, body not shown	14 / 25	#4	go and get the body
that same function call + the plan	25 / 25	#2	go and get the body
Haiku 4.5, function call only	5 / 25	#3	go and get the body
Haiku 4.5, + the same plan	25 / 25	#2	go and get the body

```
SELECT * FROM bluebox.customer_revenue_report('%secret%');
```

The same query, hidden behind a function

what the model was given	recommended it	rewrite rank	the #1 recommendation
the SQL, subquery visible	25 / 25	#1	rewrite the subquery
a function call, body not shown	14 / 25	#4	go and get the body
that same function call + the plan	25 / 25	#2	go and get the body
Haiku 4.5, function call only	5 / 25	#3	go and get the body
Haiku 4.5, + the same plan	25 / 25	#2	go and get the body

```
SELECT * FROM bluebox.customer_revenue_report('%secret%');
```

Put the same query behind a function and it falls apart

Nothing is hidden from the database

The same query, hidden behind a function

what the model was given	recommended it	its rank	the #1 recommendation
the SQL, subquery visible	25 / 25	#1	rewrite the subquery
a function call, body not shown	14 / 25	#4	go and get the body
that same function call + the plan	25 / 25	#2	go and get the body
Haiku 4.5, function call only	5 / 25	#3	go and get the body
Haiku 4.5, + the same plan	25 / 25	#2	go and get the body

```
SELECT * FROM bluebox.customer_revenue_report('%secret%');
```

It is ranked second, not first, because first is still "go and get the function body".

That is the model being right.

It asked the right question, then answered anyway



It asked for the function body

Recommendation #1 in 50 runs out of 50.

"Don't tune blind. Get the function body first, then look at the plan."

That is exactly the right first move.

Then it answered anyway

Four more recommendations about a query it could not see.

It asked for an index in 25 of 25 runs, ranked #2, without one column name to put it on.

Only 44% said plainly they could not tell.

What this measures, and what it doesn't



What I can claim

When the problem is legible, context barely changes the diagnosis. It mostly suppresses wrong advice.

When it is not legible, nothing in the SQL tier recovers it. The plan does, completely.

What I can't claim

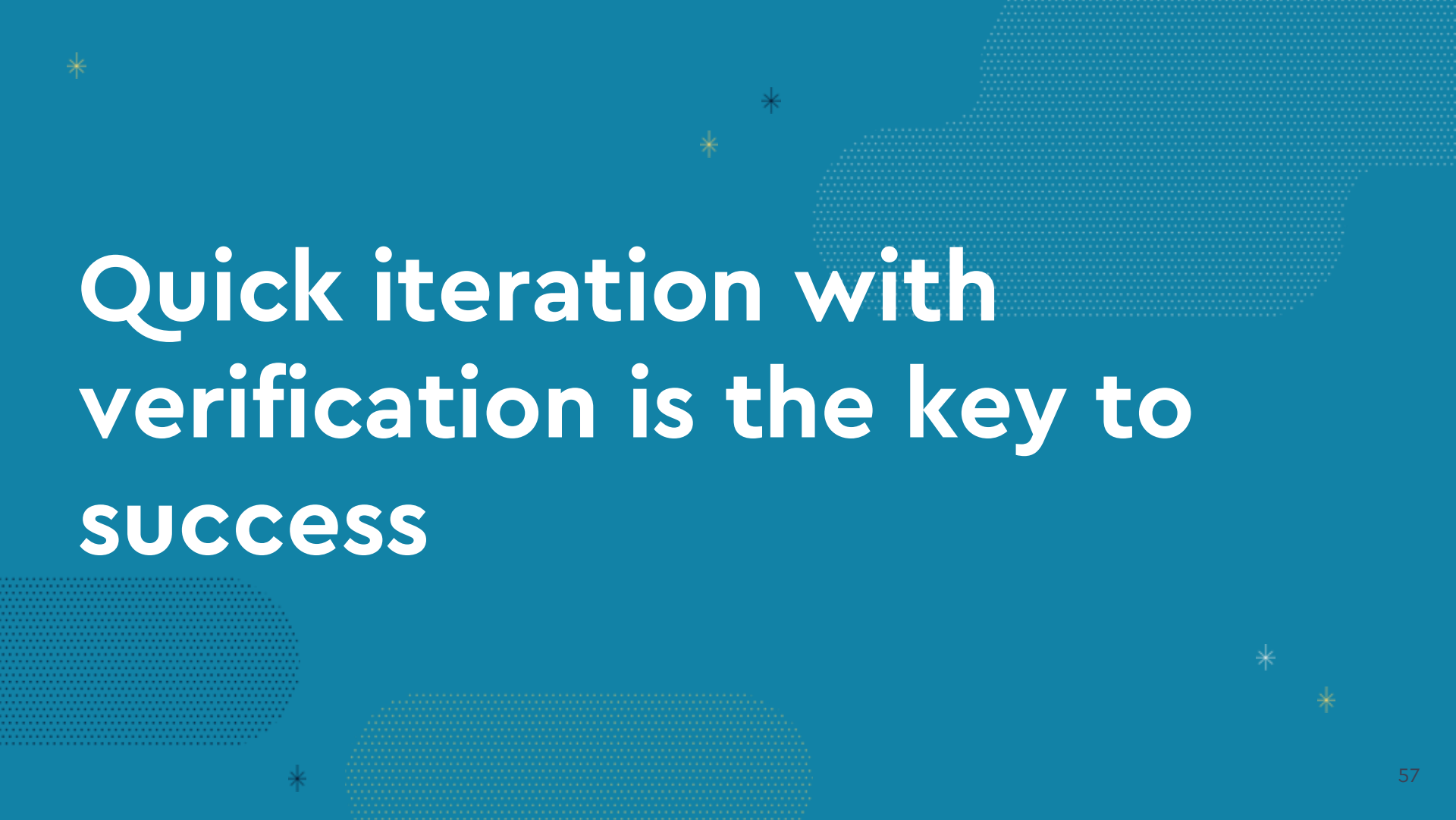
How any of this scales to a genuinely gnarly query.

I made a simple query hard to see. Real problem queries are hard to see AND hard to reason about.

I only varied the first one.



That's arguably a better
result... but still slow
and error prone



Quick iteration with
verification is the key to
success



Part four: AI with history and metrics

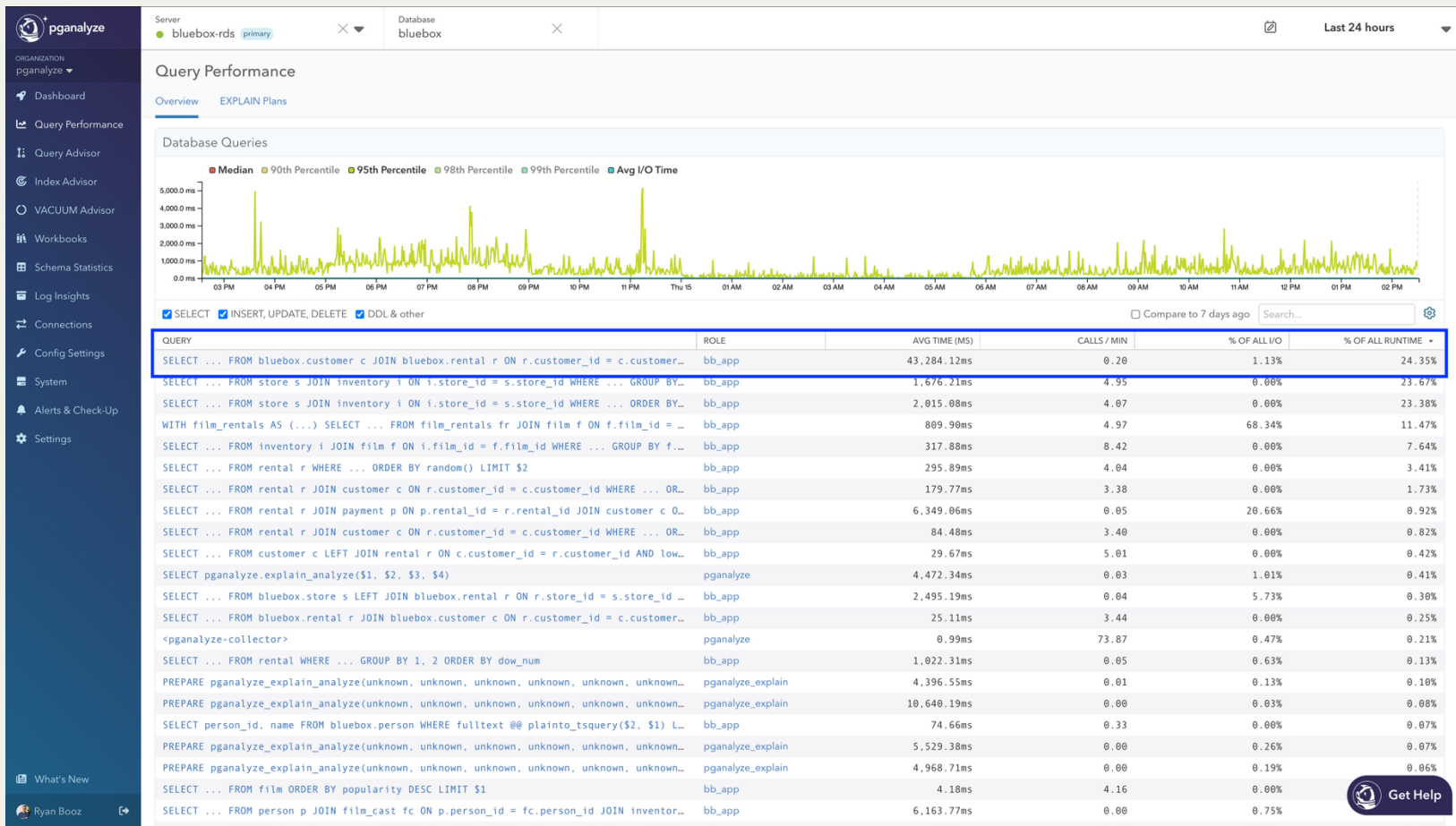
pganalyze MCP Server

AI tools never touch Postgres

- They read what the collector already gathered: stored metrics, query samples, previously captured plans.

Only Workbooks execute queries through the collector

- AI stops to ask for/confirm parameter values, because parameters change plans.
- Most other Postgres MCP hands the agent a live connection and a snapshot.
- But remember - a snapshot can't see a regression.





Claude Code v2.1.296
Sonnet 5.5 · Claude Team
~/Library/CloudStorage/OneDrive-Personal/Conference Talks/Postgres Tuning in the Age of AI - Context is Everything

Updated to latest. Got 50 features, 423 bugfixes, and 128 other changes.
code.claude.com/docs/en/changelog for details

Get to finished work sooner with Opus 5.5. Switch anytime with /model.

> /mcp
└ MCP dialog dismissed

> I've noticed a query on my bluebox-rds database that's running slowly and seems to have been introduced recently. The runtime varies widely and for certain parameters the query appears to query significant numbers of buffers that are multitudes larger than the total amount of data in those tables.

Please investigate query 44097450 and provide suggestions on how to mitigate the issue.

▶▶ auto mode on (shift+tab to cycle) ✓ Update installed · Restart to update



I've noticed a query on my bluebox-rds database that's running slowly and seems to have been introduced recently. The runtime varies widely and for certain parameters the query appears to query significant numbers of buffers that are multitudes larger than the total amount of data in those tables.

Please investigate query 44097450 and provide suggestions on how to mitigate the issue.



```

> claude
Claude Code v2.1.300
Sonnet 5.5 · Claude Team
~/Library/CloudStorage/OneDrive-Personal/Conference Talks/Postgres Tuning in the Age of AI - Context is Everything

Get to finished work sooner with Opus 5.5. Switch anytime with /model.

> /mcp
  MCP dialog dismissed

> I've noticed a query on my bluebox-rds database that's running slowly and seems to have been introduced recently. The runtime varies widely and for certain parameters the query appears to query significant numbers of buffers that are multitudes larger than the total amount of data in those tables.

Please investigate query 44097450 and provide suggestions on how to mitigate the issue.

Called pganalyze 2 times (ctrl+o to expand)

• This org has no server named bluebox-rds, so I'll check the staging pganalyze connection.

Called pganalyze staging 4 times (ctrl+o to expand)

• Bash(cd /Users/ryanbooz/.claude/projects/*/da5bd230*/tool-results/ && f=$(ls mcp-claude_ai_pganalyze_staging-get_query_explains-*.txt) && head -c 3000 $f; echo; grep...)
  {
    "explainId": "5oqn4j3p6bff5hs7pop2m5o5oa",
    ... +211 lines (ctrl+o to expand)
  }
  Shell cwd was reset to /Users/ryanbooz/Library/CloudStorage/OneDrive-Personal/Conference Talks/Postgres Tuning in the Age of AI - Context is Everything

• Bash(cd /Users/ryanbooz/.claude/projects/*/da5bd230*/tool-results/ && python3 -I -c "
  import json,glob...
  [ jl4awa7o5fgq7bedds2vl7qt3q 1792023836 250471 14539427.0 229722831 0 07a142
    vjj7evsvibc2tinvr6e4vpdkoy 1792021986 226897 14539493.0 170263062 0 07a142
    zbrni7laajhnnjtu7o5apm6igy 1792021369 217557 21428158.0 197744710 0 07a142
    ... +48 lines (ctrl+o to expand)
  ]
  Shell cwd was reset to /Users/ryanbooz/Library/CloudStorage/OneDrive-Personal/Conference Talks/Postgres Tuning in the Age of AI - Context is Everything
  Allowed by auto mode classifier

• Up to 229M buffers hit (~1.7 TB) against a 2.4 GB database. I'm pulling the worst plan to find the node responsible.

Called pganalyze staging (ctrl+o to expand)

• Query 44097450 on bluebox-rds (database bluebox, staging pganalyze org) is slow because of its correlated subquery. It recomputes the same store-level average for every output row.

I read the plan history (about 50 auto_explain plans from the last 7 days) and the worst plan in full. I haven't run the rewrite below, and I didn't check when the query first appeared.

What the data shows

```

Bluebox-rds query 44097450 slowdown

Called pganalyze staging 3 times (ctrl+o to expand)

- Bash(cd /Users/ryanbooz/.claude/projects/*/f980ebf3*/tool-results/ && head -c 2500 mcp-claude_ai_pganalyze_staging-get_query_explains-*.txt; echo; grep -nE '"(id|occ...)' [{ { "explainId": "6rk6it5yyzh57azdszszsuxugzy", ... +162 lines (ctrl+o to expand) L Shell cwd was reset to /Users/ryanbooz/Library/CloudStorage/OneDrive-Personal/Conference Talks/Postgres Tuning in the Age of AI - Context is Everything L Allowed by auto mode classifier

Called pganalyze staging 2 times (ctrl+o to expand)

- The slow plan comes from a correlated subquery that Postgres re-runs for each of the 200 output rows. It accounts for about 99% of the buffer reads. This is from the plan history on bluebox-rds / bluebox (staging pganalyze, database 10086556). I haven't tested any fixes yet.

What the plans show

Auto_explain captured about 30 plans for this query over the last 7 days. Runtimes range from about 1 s to 190 s, and the slowest plan runs 175 s with 197.7M shared buffers hit. At 8 KB each, that's about 1.5 TB of buffer traffic against a 2.4 GB database.

Part of the plan	Time	Buffers hit
Outer query (joins, aggregate, sort)	about 1.3 s	1.76M
SubPlan 1 (store_avg_spend), loops=200	about 870 ms per loop, about 174 s total	195.9M

- The subquery re-runs the whole 5-table join 200 times. Each run scans inventory, builds the film hash, does about 32M index probes into rental and scans customer to filter on store_id.
- The subquery only depends on c.store_id. The store_avg_spend result is identical for every customer in the same store. Postgres doesn't cache it across rows, so it recomputes it for each of the 200 output rows.
- The runtime varies because the parameters change the work. The filters are ILIKE \$2 on film.overview and \$1::interval on rental age. A common word like %love% (832 films) or a long window matches many film and rental rows. A selective pattern or short window makes each loop cheap. The cost is also multiplied by how many rows survive HAVING count > \$5, up to the LIMIT.
- Other costs are smaller. ILIKE '%...%' forces a sequential scan of film on every loop, about 64 ms times 401. customer is also seq-scanned for each store filter, about 16 ms times 200.

I couldn't confirm that the query is new or find what changed. I didn't check when this fingerprint first appeared, and I have no deploy history.

Fixes, in order of impact

- Compute the store average once instead of per row. This should remove the 195M-buffer subplan, leaving a query of about 1-2 s,

```
* Bluebox-rds query 44097450 slowdown
```

1. Compute the store average once instead of per row. This should remove the 195M-buffer subplan, leaving a query of about 1–2 s, similar to the outer part.

```
WITH spend AS (
  SELECT c2.store_id, r2.customer_id, sum(p2.amount) AS spend
  FROM bluebox.rental r2
  JOIN bluebox.customer c2 ON c2.customer_id = r2.customer_id
  JOIN bluebox.payment p2 ON p2.rental_id = r2.rental_id
  JOIN bluebox.inventory i2 ON i2.inventory_id = r2.inventory_id
  JOIN bluebox.film f2 ON f2.film_id = i2.film_id
  WHERE lower(r2.rental_period) >= now() - $1::interval
  AND f2.overview ILIKE $2
  GROUP BY c2.store_id, r2.customer_id
), store_avg AS (
  SELECT store_id, avg(spend) AS store_avg_spend FROM spend GROUP BY store_id
)
SELECT c.customer_id, c.full_name, c.store_id,
       count(r.rental_id) AS rentals,
       sum(p.amount) AS revenue,
       max(upper(r.rental_period))::date AS last_return,
       sa.store_avg_spend
FROM bluebox.customer c
JOIN bluebox.rental r ON r.customer_id = c.customer_id
JOIN bluebox.payment p ON p.rental_id = r.rental_id
JOIN bluebox.inventory i ON i.inventory_id = r.inventory_id
JOIN bluebox.film f ON f.film_id = i.film_id
JOIN store_avg sa ON sa.store_id = c.store_id
WHERE lower(r.rental_period) >= now() - $3::interval
  AND f.overview ILIKE $4
GROUP BY c.customer_id, c.full_name, c.store_id, sa.store_avg_spend
HAVING count(r.rental_id) > $5
ORDER BY revenue DESC, last_return DESC
LIMIT $6;
```

If the app always passes the same values for \$1/\$3 and \$2/\$4, the spend CTE repeats the outer query's join. You could then derive the store average from one pass using a window function. The shape above is the safe version that works whatever the parameter values are.

2. Add indexes to cut the remaining scans. They help if the rewrite doesn't get the query fast enough.

- CREATE EXTENSION pg_trgm; CREATE INDEX ON bluebox.film USING gin (overview gin_trgm_ops); supports ILIKE '%.%'. film is small, so the benefit is modest and mostly applies to selective patterns.
- CREATE INDEX ON bluebox.rental ((lower(rental_period))); supports the time filter, which is currently applied as a post-filter on rental_inventory_id_idx.
- CREATE INDEX ON bluebox.customer (store_id); removes the repeated 185k-row seq scan of customer.

3. Mitigations while the fix ships.

- Set statement_timeout for the bb_app role. The 175 s runs are unbounded now.
- Cap the allowed \$1 window and the pattern length or selectivity in the app.
- Cache store_avg_spend in the app per (store, window, pattern), since it's one row per store.

Suggested next step

Suggested next step

I can create a pganalyze Workbook for this query, using parameter sets taken from a slow sample and a fast sample. It would run `EXPLAIN ANALYZE` on the baseline and on the rewrite above, so you can compare buffers and runtime before changing anything. I'd also check when this query fingerprint first appeared to pin down the "recently introduced" part. Want me to proceed?

ORGANIZATION
pganalyze

Dashboard

Query Performance

Query Advisor

Index Advisor

VACUUM Advisor

Workbooks

Schema Statistics

Log Insights

Connections

Config Settings

System

Alerts & Check-Up

Settings

What's New

Ryan Booz

Server
bluebox-rds primary

Database
bluebox

Fix correlated subquery in customer revenue report (#44097450)

Overview

Compare Plans

Parameter Sets

Activity

All Query Plans

Choose a query variant to see its query text and settings, or create a new variant. Variants can be used to test Postgres planner behavior or rewrite queries to improve performance. [Learn more.](#)

Query Plans

Filter by Parameter Set...

PLAN	VARIANT	PARAMETER SET	EST. COST	RUNTIME
☐ 72b33f0	Baseline	Parameter Set 1	13,922,621	188,594.44ms
☐ 056d6d0	Baseline	Parameter Set 4	23,111,052	46,212.64ms
☐ 338b56e	Baseline	Parameter Set 5	263,672	2,565.83ms

New Query Variant

All Query Plans

Baseline

Query Advisor

Inefficient Nested Loop

Found a Nested Loop with a high row misestimate on the outer table.

Try rewriting the query with a materialized CTE to use a different query plan.

New Variant

Found in 338b56e

Inefficient Nested Loop

Found a Nested Loop with a high row misestimate on the outer table.

Try rewriting the query with a materialized CTE to use a different query plan.

New Variant

Found in 338b56e

Baseline Query

#44097450

Owner
Ryan Booz

Workbook description
INVESTIGATION NOTES - query 44097450 (customer revenue report), bluebox-rds / bluebox, 2026-10-14 PROBLEM Runtime

Compare selected plans

No plans selected

Get Help

ORGANIZATION
pganalyze

Dashboard

Query Performance

Query Advisor

Index Advisor

VACUUM Advisor

Workbooks

Schema Statistics

Log Insights

Connections

Config Settings

System

Alerts & Check-Up

Settings

What's New

Ryan Booz

Server
bluebox-rds primary

Database
bluebox

Fix correlated subquery in customer revenue report (#44097450)

OverviewCompare PlansParameter SetsActivity

All Query Plans

Choose a query variant to see its query text and settings, or create a new variant. Variants can be used to test Postgres planner behavior or rewrite queries to improve performance. [Learn more.](#)

Query Plans

Filter by Parameter Set...

PLAN	VARIANT	PARAMETER SET	EST. COST	RUNTIME
☐ 72b33f0	Baseline	Parameter Set 1	13,922,621	188,594.44ms
☐ 056d6d0	Baseline	Parameter Set 4	23,111,052	46,212.64ms
☐ 338b56e	Baseline	Parameter Set 5	263,672	2,565.83ms
☐ 3f13bcb	Materialized CTE: store avg once per store	Parameter Set 1	223,057	3,473.42ms
☐ f2e751c	Materialized CTE: store avg once per store	Parameter Set 4	513,844	3,202.55ms
☐ b1dfb51	Materialized CTE: store avg once per store	Parameter Set 5	4,732	1,391.40ms
☐ f1640a1	Materialized CTE + work_mem 64MB	Parameter Set 1	199,932	5,529.30ms
☐ cc9f01d	Materialized CTE + work_mem 64MB	Parameter Set 4	434,868	9,073.11ms
☐ b1dfb51	Materialized CTE + work_mem 64MB	Parameter Set 5	4,732	2,867.80ms
☐ a6d4b89	Top-N first, store avg only for returned stores	Parameter Set 1	210,518	6,520.74ms
☐ 329f8b	Top-N first, store avg only for returned stores	Parameter Set 4	468,691	7,091.72ms
☐ 644f2c3	Top-N first, store avg only for returned stores	Parameter Set 5	4,636	2,158.96ms

New Query Variant

All Query Plans

Baseline

Materialized CTE: store avg once per store

Materialized CTE + work_mem 64MB

Top-N first, store avg only for returned stores

Query Advisor

Inefficient Nested Loop

Found a Nested Loop with a high row misestimate on the outer table.

Try rewriting the query with a materialized CTE to use a different query plan.

New Variant

Found in 338b56e

Inefficient Nested Loop

Found a Nested Loop with a high row misestimate on the outer table.

Try rewriting the query with a materialized CTE to use a different query plan.

New Variant

Found in 338b56e

Baseline Query

#44097450

Owner

Ryan Booz

Workbook description

INVESTIGATION NOTES - query 44097450 (customer revenue report), bluebox-rds / bluebox, 2026-10-14 PROBLEM Runtime

Get Help

Compare selected plans

No plans selected

ORGANIZATION
pganalyze

Dashboard

Query Performance

Query Advisor

Index Advisor

VACUUM Advisor

Workbooks

Schema Statistics

Log Insights

Connections

Config Settings

System

Alerts & Check-Up

Settings

What's New

Ryan Booz

Server
bluebox-rds primary

Database
bluebox

Fix correlated subquery in customer revenue report (#44097450)

OverviewCompare PlansParameter SetsActivity

All Query Plans
Choose a query variant to select

Query Plans
PLAN

☐ 72b330

☐ 056d6d0

☐ 338b56e

☐ 3f13bcb

☐ 2e751c

☐ b1dfb51

☐ f1640a1

☐ cc901d

☐ b1dfb51

☐ a6d4b89

☐ 329f8b

☐ 6442c3

Fix correlated subquery in customer revenue report (#44097450)

Name

Fix correlated subquery in customer revenue report (#44097450)

Description (Optional)

INVESTIGATION NOTES - query 44097450 (customer revenue report), bluebox-rds / bluebox, 2026-10-14

PROBLEM
Runtime varies 1 s to ~190 s; shared buffers hit up to ~230M vs ~2.4 GB total DB size. 56 auto_explain plans from the last 7 days compared (all shared hit, 0 read: CPU/cache cost, not I/O).

ROOT CAUSE
The correlated scalar subquery (store_avg_spend) is correlated only on c.store_id (~196 stores) but is re-executed once per output row (SubPlan loops=200). Each execution ~945 ms / ~1.13M buffers (5-table join, seq scan of inventory, ~36M probes into rental in total). 226M of the 228M buffers in the slowest plan are in SubPlan 1. Cost scales with (rows surviving HAVING/LIMIT) x (per-store scan size set by the ILIKE pattern + interval). Fast plan: %legend%, 2 rows returned, 1.0 s / 0.92M buffers. Slow plan: %family%, 200 rows, 191.8 s / 228.6M buffers.

Secondary: ILIKE '%x%' on film.overview forces a seq scan (repeated in every subplan run); rental_inventory_id_idx probes discard ~2/3 of rows on the lower(rental_period) filter; occasional sort spills (external merge ~5 MB).

BASELINE (id 563), parameter sets: 375 %family% 1.5y | 378 %war% 2y | 379 %revenge% 2y (LIMIT 200, HAVING > 4) %family% 188.6 s / 229M buf | %war% 46.2 s / 23.7M | %revenge% (5 rows) 2.6 s / 3.0M

VARIANTS TRIED (single run each; differences <= ~1 s are within noise; row counts matched baseline in all cases)

1) 569 Materialized CTE: store_avg once per store (LEFT JOIN store_avg)
family 4.8 s / 4.2M | war 9.6 s / 3.1M | revenge 2.8 s / 1.4M
Big win on 200-row cases; slightly slower on the 5-row case because the CTE computes all 196 stores up front.

2) 570 Top-N first, store_avg only for stores in the top rows (RECOMMENDED)
family 2.9 s / 4.2M | war 4.0 s / 3.1M | revenge 1.0 s / 1.2M
Best or tied-best everywhere, no index dependency. ~65x faster on the worst case.

3) 571 Materialized CTE + GIST containment: rental_period <@ tstzrange(now()-interval, far_future) instead of lower(rental_period) >= ...
family 4.6 s / 4.2M | war 3.4 s / 3.1M | revenge 1.3 s / 1.4M
NO benefit: planner never used rental_rental_period_idx (GIST); it still goes through rental_inventory_id_idx and applies the predicate as a Filter (a 1.5-2y window is not selective).

Update

Edit name and description

Delete workbook

40.2 s / 23.7M | %revenge% (5 rows) 2.6 s / 3.0M

VARIANTS TRIED (single run each; differences <= ~1 s are within noise; row counts matched baseline in all cases)

1) 569 Materialized CTE: store_avg once per store (LEFT JOIN store_avg)
family 4.8 s / 4.2M | war 9.6 s / 3.1M | revenge 2.8 s / 1.4M
Big win on 200-row cases; slightly slower on the 5-row case because the CTE computes all 196 stores up front.

2) 570 Top-N first, store_avg only for stores in the top rows (RECOMMENDED)
family 2.9 s / 4.2M | war 4.0 s / 3.1M | revenge 1.0 s / 1.2M
Best or tied-best everywhere, no index dependency. ~65x faster on the worst case.

3) 571 Materialized CTE + GIST containment: rental_period <@ tstzrange(now()-interval, far_future) instead of lower(rental_period) >= ...
family 4.6 s / 4.2M | war 3.4 s / 3.1M | revenge 1.3 s / 1.4M
NO benefit: planner never used rental_rental_period_idx (GIST); it still goes through rental_inventory_id_idx and applies the predicate as a Filter (a 1.5-2y window is not selective).

What it did well, honestly



It found the rewrite

- Given the plan, it identified the SubPlan and proposed MATERIALIZED CTEs. That's the same answer I reached.

It was tireless

- It tested variants I'd have dismissed without checking.

It had no priors to be wrong about

- My speed comes from pattern recognition. Pattern recognition is exactly what fails on the unusual case.

It documented everything

- Every variant it tried is still in the workbook. My two experiments were in my head.

What it could have improved on



It stopped at the presenting problem

- It found the SubPlan and proposed the rewrite. It did not go on to the work_mem spill the fix exposed.

It couldn't tell "better" from "solved"

- A 6% improvement and a 40x improvement got the same enthusiasm.

It doesn't always know when to stop

- Left alone it may keep optimizing well past the point of any return.

Every iteration was real load

- On a database that was already struggling. The agent is a workload too.

An agent exploring your database is a workload.

- Multiple EXPLAIN ANALYZE runs against a struggling production system is not free diagnosis.
- This is the strongest practical argument for reading collected monitoring data instead of the live database
- Run the experiments that must touch the database somewhere safe, under parameters a human approved.
- **Budget the agent's curiosity like you'd budget anyone else's.**

Three things worth doing Monday

1. Turn on the four

- `pg_stat_statements`, `auto_explain`, `track_io_timing`, and a sane `log_min_duration(_statement)` threshold.
- Whether or not an AI ever touches your database, you can't answer "why" without them.

2. Write your questions

- The seven mentioned earlier for query analysis
- That list is what you'd hand a new engineer.
- It's also the context for a prompt you'd hand a model.

3. Make verification a habit

- Say what you expect the fix to change before you run it.
- Then test it against more than one set of parameters.

Questions?



Thank you!

Learn more at pganalyze.com

Contact us: sales@pganalyze.com