

TECHNICAL SESSION · INTERMEDIATE

Explaining the Unexplainable

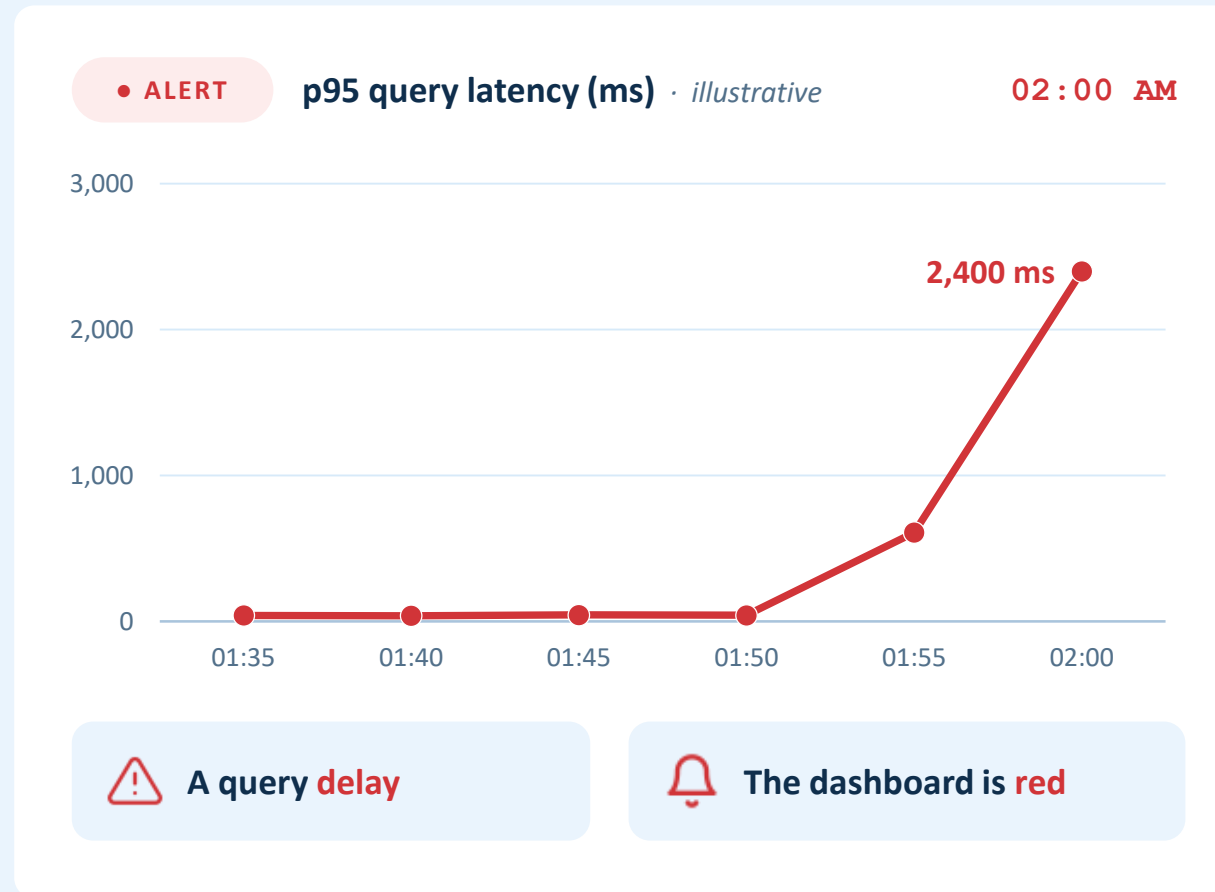
Mastering EXPLAIN ANALYZE for Real-World Query Tuning

Kavach Shah · Nikunj Doshi



Thursday, October 1, 2026 · 13:30–14:20

It's 2 AM. Production is melting.



THE 2 AM QUESTIONS

- 1 Which query?** SECTION 05
pg_stat_statements ranks the statements costing the most total time.
- 2 Why is it slow?** SECTIONS 01-04
EXPLAIN ANALYZE pinpoints the node where estimates and reality part ways.
- 3 What do I fix first?** WRAP-UP
A repeatable checklist that gets you to a fix the same day.

What we'll cover

- **Why it matters**
Adoption, expert voices and the research on bad plans
- **01 · Reading the Tree**
How Postgres executes a plan, inside-out
- **02 · Cost vs. Actual Time**
Spotting where estimates and reality part ways
- **03 · Node-by-Node Breakdown**
Seq Scan vs. Index Scan vs. Bitmap Heap Scan
- **04 · The Usual Suspects**
Bad estimates, missing indexes, DISTINCT misuse, nested loops
- **05 · Tools That Help**
EXPLAIN options, auto_explain and pg_stat_statements
- **Wrap-up**
The debugging checklist, trade-offs, takeaways and Q&A

WHAT YOU'LL LEAVE WITH



A mental model

Section 01 · Reading the Tree



A sharper eye

Sections 02–04



A toolkit

Section 05 · Tools That Help



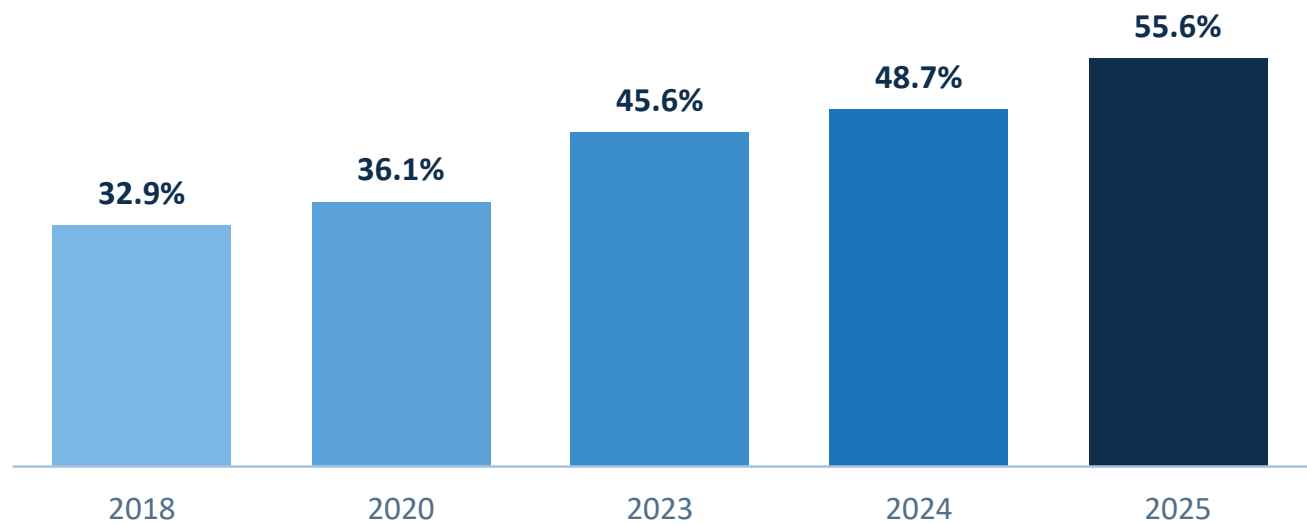
A checklist

Wrap-up · same-day debugging

PostgreSQL is now developers' most-used database

Developers using PostgreSQL

Share of all Stack Overflow Developer Survey respondents, selected years



Source: Stack Overflow Developer Survey 2018, 2020, 2023, 2024 and 2025 (all respondents).

55.6%



of all 2025 respondents use PostgreSQL, more than any other database.

#1



Most-used, most-admired and most-desired database, 2023–2025.

+22.7 pts



Up from 32.9% in 2018.

What the experts keep telling us

OFFICIAL DOCS

“ *Plan-reading is an art that requires some experience to master...* ”

PostgreSQL Documentation · §14.1 “Using EXPLAIN”

ENGINEERING PRINCIPLE

“ *Measure. Don’t tune for speed until you’ve measured...* ”

Rob Pike · Notes on Programming in C, Rule 2 (1989)

RESEARCH

“ *...cardinality estimation is much more crucial than the cost model.* ”

Leis et al. · “How Good Are Query Optimizers, Really?”, PVLDB 2015

PRACTITIONER TESTIMONIAL

“ *...it can easily show problems that might be otherwise unclear.* ”

Hubert “depesz” Lubaczewski · on PostgreSQL 18 enabling BUFFERS by default (2025)

What the research says about bad plans



3 · 9 · 128 · 310

Row estimates PostgreSQL gave for the same simple 2-join query, just by reordering tables or predicates. True count: 2,600.



1,000×

Every system tested, commercial engines included, routinely misestimated row counts by 1,000× or more.

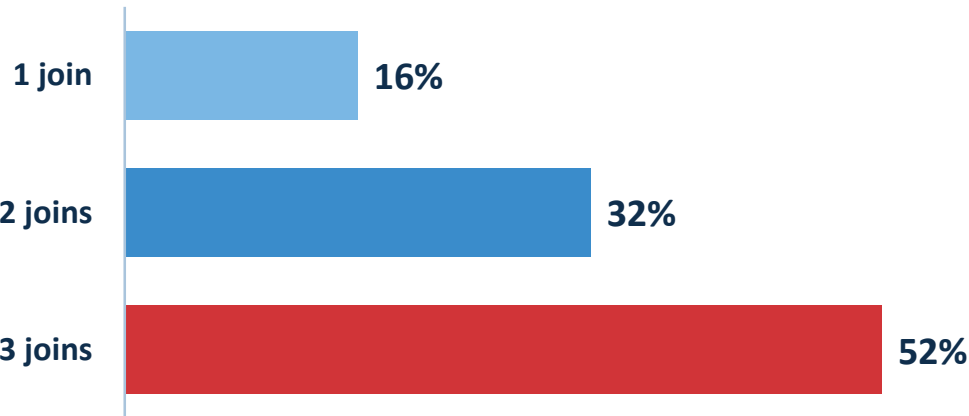


Nested loops

Most timeouts shared one trait: a far-too-low estimate led PostgreSQL to pick a nested-loop join with no index lookup.

PostgreSQL row estimates off by 10× or more

Share of estimates, by number of joins in the query



Errors compound with every join, and multi-join results are systematically underestimated.

Source: Leis et al., “How Good Are Query Optimizers, Really?”, PVLDB 9(3), 2015 · Join Order Benchmark (IMDB), PostgreSQL 9.4.

Five stops on the way to a fix

01



Reading the Tree

How Postgres executes a plan, inside-out



02



Cost vs. Actual Time

Where estimates and reality part ways



03



Node-by-Node Breakdown

Seq Scan vs. Index Scan vs. Bitmap Heap Scan



04



The Usual Suspects

Four patterns behind slow plans



05



Tools That Help

EXPLAIN options, auto_explain and pg_stat_statements



SECTION 01 OF 5

Reading the Tree

1

What you read vs. what actually runs

WHAT YOU SEE

Top to bottom, outermost node first: the final result sits at the very top of the plan.



Sort

- Hash Join
 - Seq Scan on orders
 - Hash
 - Index Scan on customers

WHAT ACTUALLY RUNS

Inside out: the most-indented nodes usually produce rows first. A Hash Join builds its hash table before it starts probing.



1. Index Scan on customers
2. Hash (builds hash table)
3. Seq Scan on orders (probes)
4. Hash Join (emits matches)
5. Sort (final output)

A sample plan, read in execution order



```
Sort (cost=1193.52..1193.77 rows=100 width=64)
  -> Hash Join (cost=88.00..1190.20 rows=100)
        Hash Cond: (o.customer_id = c.id)
        -> Seq Scan on orders o (cost=0.00..980.00)
              Filter: (status = 'pending'::text)
        -> Hash (cost=63.00..63.00 rows=2000)
              -> Index Scan using customers_pkey on customers c
```

Plain EXPLAIN output (estimates only), trimmed for space.



THE ONE RULE

Indentation is nesting. The most-indented nodes usually produce rows first.



SECTION 02 OF 5

Cost vs. Actual Time

2

Two very different numbers



cost

The planner's prediction in arbitrary units, estimated BEFORE the query runs from table statistics.

Only as good as the last ANALYZE.



actual time

Real milliseconds, measured DURING execution when you run EXPLAIN ANALYZE. Ground truth.

Reported per loop (more on that shortly).

Anatomy of one plan line



```
Seq Scan on orders  (cost=0.00..980.00 rows=50000 width=48)
                    (actual time=0.020..842.110 rows=48.00 loops=1)
```

- `cost=start..total` Planner's estimate in arbitrary units, not milliseconds.
- `rows=estimated` How many rows the planner expects this node to return.
- `actual time=start..total` Real milliseconds to the first row and to the last row (averaged per loop).
- `rows=48.00 loops=1` Estimated 50,000 rows, actual 48: a ~1,000× miss. PG 18+ prints actual rows with two decimals.

The loops trap

actual time is the average for ONE loop, not the node's total contribution.



```
Index Scan using idx_o1 on order_items
```

```
(actual time=0.050..0.850
```

```
rows=3.00 loops=48000)
```

```
Real time = 0.85 × 48000
```

```
≈ 40.8 seconds
```



The formula that matters

Total time contributed by a node =

actual time (per loop) × loops

A node with tiny per-loop time can still dominate the query if it runs inside a nested loop tens of thousands of times.

In parallel plans, loops also counts worker processes, so time × loops overstates wall-clock time.

When estimated and actual rows diverge

PLANNER ESTIMATED

rows=50,000

ACTUALLY RETURNED

rows=48.00

≈ 1,000×

off, at a single node



A miss this large is a red flag, not a rounding error.

- Table statistics are stale: run ANALYZE (or check autovacuum settings).
- Correlated columns the planner can't model: consider extended statistics (CREATE STATISTICS).
- A function or expression in the WHERE clause the planner can't estimate selectivity for.



SECTION 03 OF 5

Node-by-Node Breakdown

3

Three common ways Postgres reads a table



Seq Scan

Reads every page of the table in physical order.

WHEN IT APPEARS

Appears when a large fraction of rows qualify, the table is small, or there's no useful index.

Not inherently bad: sometimes it's the fastest option.



Index Scan

Walks an index, fetches matching rows one at a time.

WHEN IT APPEARS

Preferred when a small fraction of rows qualify and the index is selective.

Watch for random I/O cost on very large result sets.



Bitmap Heap Scan

Builds a bitmap of matching pages, then reads them in physical order.

WHEN IT APPEARS

The middle ground: medium selectivity, too many rows for an Index Scan, too few for a full Seq Scan.

A sign the planner is balancing random vs. sequential I/O.

Side by side

	Best For	I/O Pattern	Watch For
Seq Scan	Low selectivity / small tables	Sequential	High cost on large filtered tables
Index Scan	High selectivity, few matching rows	Random	Slow on large result sets
Bitmap Heap Scan	Medium selectivity	Bitmap-sorted, mixed	Lossy “Heap Blocks” (bitmap outgrew work_mem)

A quick gut-check

Roughly, what share of the table does this query need?

< 5%

Index Scan

5–25%

Bitmap Heap Scan

> 25%

Seq Scan

Rule of thumb only: the real crossover shifts with row width, caching, random_page_cost and physical correlation.

If the plan doesn't match your expectation (an Index Scan you thought would win, or a Seq Scan on a table you swore was indexed), that mismatch is where you start investigating.



SECTION 04 OF 5

The Usual Suspects

4

Suspect #1: Bad Row Estimates

SIGNATURE IN THE PLAN



```
Index Scan using idx_events_created_at on events
(cost=0.43..412.10 rows=1200)
(actual time=0.500..891.200
 rows=210000.00 loops=1)
```



THE TELL

Estimated 1,200 rows, returned 210,000: a 175× miss at a single node.



THE CAUSE

Table statistics are stale, sample size is too small, or columns are correlated in ways the planner can't model on its own.



THE FIX

Run ANALYZE (or check autovacuum thresholds); raise the statistics target if the sample is too small. For correlated columns, add CREATE STATISTICS.

Suspect #2: Missing Indexes

SIGNATURE IN THE PLAN



Seq Scan on payments

(cost=0.00..112275.05 rows=3)

(actual time=41.237..1893.556 rows=3.00 loops=1)

Filter: (customer_id = 88213)

Rows Removed by Filter: 4982001



THE TELL

Nearly 5 million rows read and discarded to return just 3. The live demo repeats this on a 10M-row table.



THE CAUSE

No usable index on `customer_id`, so Postgres has to read the whole table to find the 3 rows that match.



THE FIX

Add a B-tree index on the filtered column(s). Confirm with `EXPLAIN` that the planner switches to an Index or Bitmap Scan.

The demo database: one table, no indexes

SCHEMA · EXPLAINDEMO DATABASE

payments

TABLE

~10,000,000 rows · every column NOT NULL · no primary key

id	<code>bigint</code>	identity, not a primary key
customer_id	<code>integer</code>	~1.6M values · the demo filter
amount	<code>numeric(10,2)</code>	in no index, forces heap reads
status	<code>text</code>	paid / pending / refunded
created_at	<code>timestampz</code>	within the last 365 days

indexed by
→

idx_payments_customer_id

B-tree · created by 03-fix.sql

customer_id	<code>integer</code>	B-tree key, sorted
ctid	<code>tid</code>	points to the heap row

LIFECYCLE

Dropped by 99-reset.sql, so `\d payments` lists no indexes when the demo starts.

Run order: offstage, then on stage

OFFSTAGE · BEFORE YOU WALK ON

00-setup.sql

Once, days ahead · builds 10M rows, no indexes



99-reset.sql

Drops the index, re-runs VACUUM ANALYZE



01-arm.sql

Warms the cache, turns off parallelism and JIT

🔄 BETWEEN REHEARSALS · SAME PSQL SESSION

Run 99-reset.sql, then 01-arm.sql

ON STAGE · LIVE

02-slow.sql

Seq Scan · 80,295 buffers · ~546 ms



03-fix.sql

CREATE INDEX on customer_id · ~5 s



04-fast.sql

Index Scan · 5 buffers · ~0.04 ms



05-optional-selectivity.sql

Only if ahead · three plans, one index

Suspect #3: DISTINCT Misuse

SIGNATURE IN THE PLAN



```
Unique  (cost=91820.35..102544.50)
-> Sort  (cost=91820.35..93965.18)
    Sort Key: customer_id, order_date, ...
    Sort Method: external merge  Disk: 831488kB
```



THE TELL

“external merge” plus a Disk figure means the sort spilled out of work_mem.



THE CAUSE

DISTINCT across many wide columns forces Postgres to sort (or hash) the entire result set just to de-duplicate it, often spilling to disk.



THE FIX

Ask if you need DISTINCT at all. Consider EXISTS/semi-join, GROUP BY on fewer columns, or DISTINCT ON with an index-friendly ORDER BY.

Suspect #4: Over-Eager Nested Loops

SIGNATURE IN THE PLAN



```
Nested Loop (cost=0.43..812.10 rows=45)
-> Seq Scan on orders (rows=3)
    (actual rows=180000.00)
-> Index Scan using sales_pkey on sales
    (actual time=0.020..0.310 rows=1.00
     loops=180000)
```



THE TELL

The inner node re-executed 180,000 times: 0.31 ms each becomes ~56 seconds.



THE CAUSE

The planner underestimated the outer side's row count, so it chose a Nested Loop expecting a handful of iterations. Reality delivered 180,000.



THE FIX

Fix the upstream estimate (see Suspect #1) first. If the loop count is genuinely large, nudge the planner toward a Hash or Merge Join instead.

The four to check, in order

1



Bad Row Estimates

2



Missing Indexes

3



DISTINCT Misuse

4



**Over-Eager Nested
Loops**



SECTION 05 OF 5

Tools That Help

5

EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)

ANALYZE

Actually runs the query and reports real timings and row counts. Without it, you only see estimates.

Careful with writes: INSERT, UPDATE, DELETE, MERGE and CREATE TABLE AS really execute, so wrap them in BEGIN ... ROLLBACK.

BUFFERS

Adds shared/local buffer hit and read counts per node: the fastest way to spot I/O-heavy operations.

On by default with ANALYZE since PostgreSQL 18. High “read” counts mean pages came from outside shared buffers.

FORMAT JSON

Machine-readable output ideal for feeding into visualizers or diffing two plans programmatically.

Pair with tools like explain.dalibo.com for a visual tree.

auto_explain: catch it before Slack does



Automatically logs the plan for any query that crosses a duration threshold. No need to reproduce the slow query by hand.

Ideal for the “it was slow once, at 2 AM, and never again” tickets.



With `log_analyze` on, every statement is timed per node, even ones that never get logged. The docs warn the overhead can be severe; `log_timing = off` reduces it.



```
# postgresql.conf (restart required)
shared_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '1s'
auto_explain.log_analyze = on
auto_explain.log_buffers = on
auto_explain.log_timing = off

# Then just watch the log
```

pg_stat_statements: your triage layer

Before you EXPLAIN anything, ask pg_stat_statements which query deserves your attention.



```
SELECT query, calls, mean_exec_time, total_exec_time
FROM pg_stat_statements -- needs shared_preload_libraries + CREATE EXTENSION
ORDER BY total_exec_time DESC LIMIT 10;
```

High calls, low mean time

A hot-path query: small gains multiply across volume.

Low calls, high mean time

Likely a batch job or report, usually safe to tune in isolation.

High total_exec_time

The one consuming the most total execution time. Start here.

The repeatable debugging checklist



Check `pg_stat_statements` first if you don't know which query to chase.



Multiply actual time by loops before judging a node's real cost.



Run `EXPLAIN (ANALYZE, BUFFERS)`. Don't guess from `EXPLAIN` alone.



Match scan types to expected selectivity. Investigate any mismatch.



Find the most-indented nodes first: that's usually where rows start.



Run `ANALYZE` before you touch an index if row estimates look stale.



Compare estimated rows to actual rows at every node. Flag anything off by 10× or more.

Advantages and disadvantages

● ADVANTAGES

Real measurements, not guesses

Actual time and row counts per node, taken while the query runs.

Points at one node, not the whole query

You tune the node that costs you, instead of rewriting from scratch.

Available everywhere

Built into every Postgres install. No extension, agent, or vendor tool.

Pairs with BUFFERS (on by default in PG 18+)

Exposes the I/O that raw timings hide, so cache misses stop being invisible.

● DISADVANTAGES

It actually executes the query

INSERT, UPDATE, DELETE and MERGE run for real unless rolled back.

Timing adds overhead

Instrumentation can inflate totals, especially on many small, fast nodes.

Per-loop numbers mislead

A node's real cost stays hidden until you multiply actual time by loops.

One plan, one moment

New parameters or stats can change the plan; cache state, the timings.

Four things to carry back to your team

1

Read inside-out

The most-indented nodes usually produce rows first. Start your reading there, not at the top of the plan.

2

Trust actual, not cost

Cost is a prediction in arbitrary units. Only EXPLAIN ANALYZE reports real milliseconds.

3

Multiply by loops

A 0.31 ms node inside a 180,000-iteration nested loop is a 56-second node.

4

Fix the estimate first

Most bad plans start as a bad row estimate. Run ANALYZE before you reach for a new index.

The plan was telling you all along

EXPLAIN ANALYZE does not make a query faster. It tells you exactly where to look, and on a bad night, that is the whole job.



Measure, don't guess

Estimates start the conversation. Actual timings end it.



One node at a time

The plan names the culprit. Resist the urge to rewrite everything around it.



Make it repeatable

auto_explain and pg_stat_statements turn a 2 AM scramble into a routine.



Questions?

Thank you for coming to Postgres Summit US 2026.

Kavach Shah & Nikunj Doshi

STAY IN TOUCH

Let's Connect

Questions after the session, war stories, or a plan you still can't explain? We'd like to hear about it.



Kavach Shah

LINKEDIN



Scan to connect

linkedin.com/in/kavach22



Nikunj Doshi

LINKEDIN



Scan to connect

linkedin.com/in/nikunjdoshi23