Parallel Query is Silently Stealing Your CPU

Scott Mead

Principal Engineer · Postgres Engines · AWS

Postgres Conference San Jose 2026

The Premise

Parallel query is a great feature.

It is also a great way to set your database on fire.

Refresher: how Postgres goes parallel

- Planner sees a query that *might* benefit from parallelism
- Estimates cost with `parallel_setup_cost` + `parallel_tuple_cost`
- If parallel plan wins, executor spawns **N background workers***
- Workers feed tuples back to the leader via shared memory
- Leader merges, returns result

Sounds great. What could go wrong?

* leader is also a worker

Know your defaults

```
max_parallel_workers_per_gather  = 2
max_parallel_workers             = 8
max_worker_processes             = 8
parallel_setup_cost              = 1000
parallel_tuple_cost              = 0.1
min_parallel_table_scan_size     = 8MB
min_parallel_index_scan_size     = 512kB
```

These are active on your cluster right now — whether you knew it or not.

Failure Mode #1

The "Free Lunch" Myth

The win that isn't

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT event_type, count(*)
FROM events
WHERE created_at >= now() - interval '180 days'
GROUP BY event_type;
```

Metric	Serial	Parallel (2 workers)
Wall-clock time	3 min	**1 min** ✅
Total CPU time	3 min	**~3 min** ➡️
Buffer reads	6.4M	6.4M

Wall-clock dropped. **CPU per query is the same.**

But you're now running 3 processes instead of 1.

Why this matters in production

- Your dashboard latency: down 3×. Looks great.
- Now run **200 concurrent connections** doing the same thing.
- Serial: **200 processes** competing for CPU simultaneously.
- Parallel: **600 processes** competing for CPU simultaneously.
- Same total CPU. Triple the concurrency. Triple the context switching.
- You don't have 600 processes worth of headroom. Welcome to your context-switch nightmare.

> *Parallelism is a tradeoff between latency and throughput.*
> *The default behavior assumes you only care about latency.*

Failure Mode #2

The Cost of Getting Started

Every connection is a process

Postgres does not use threads. Every client connection is a `fork()`:

- New OS process spawned
- Memory allocated and initialized
- Shared memory attached
- Authentication handshake completed
- Session state established

All of this before a single row is touched.

On a well-tuned system with a connection pool, this cost is paid once and amortized. Without one, it's paid on every request.

Every parallel query pays again

On top of the client connection cost, every parallel query launches background workers — each one its own `fork()`:

- New OS process spawned
- Dynamic shared memory segment created
- IPC message queues initialized
- Worker registered with postmaster
- Worker connects to DSM and begins execution

`parallel_setup_cost = 1000` is the planner's *model* of this cost.
The actual CPU burn is real and paid at execution time regardless.

Your connection pooler can't fix this

PgBouncer, pgpool, RDS Proxy — they all operate at the **client connection** level.

```
Client → [PgBouncer] → Backend process
                              ↓
                       parallel worker 1  ← fork()
                       parallel worker 2  ← fork()
```

The pooler eliminated the client `fork()`.
It has no visibility into — and no control over —
the workers spawned inside the backend.

Two separate taxes. The pooler only waives one.

The compounding problem

A query with **5ms execution time** on a large table might spend:

```
client connection setup:     ~2ms
parallel worker launch (×2):  ~3ms
                             ------
actual query execution:        5ms
                             ------
total wall-clock:             10ms   ← not 5ms
```

Half the wall-clock time is setup, not work.

Now multiply by 500 concurrent requests.
Workers launching, forking, tearing down — constantly.
Your CPU is busy before any data moves.

Detection

```sql
-- Connection churn — new connections being created constantly?
SELECT count(*), state, backend_start::date
FROM pg_stat_activity
GROUP BY state, backend_start::date
ORDER BY backend_start::date DESC;

-- Worker overhead — how many workers active right now?
SELECT count(*) FILTER (WHERE backend_type = 'parallel worker') AS workers,
       count(*) FILTER (WHERE backend_type = 'client backend')  AS clients
FROM pg_stat_activity;
```

If `workers` is a significant fraction of `clients`, you're paying the parallel setup tax on most of your queries.

Failure Mode #3

Your Best Customer Gets the Worst Plan

The setup

```
SELECT * FROM events WHERE customer_id = 42;
```

100K customers. Events table grows over time.
Each customer accumulates events at roughly the same rate.

At **any normal event count per customer**, the planner correctly chooses an index scan. Fast. Predictable.

Until it isn't.

The crossover

On a **50GB events table** (250M rows), at **547,000 events** for a single customer:

```
Index Scan cost       = 547,000 × 4.0
                      = 2,188,000

Parallel Seq Scan cost = 1,000
                       + (6,400,000 / 3)
                       + (0.1 × 547,000)
                       = 1,000 + 2,133,333 + 54,700
                       = 2,189,033
```

Dead even. One more event tips the balance.

Below **~547K events**: index scan ✅

Above **~547K events**: parallel seq scan

* serial seq scan at this table size: 6,400,000 — never competitive

Index cost: the full picture

The crossover slide uses `matching_rows × random_page_cost` — a reasonable approximation for a low-correlation index. The planner's actual formula:

```
Index Scan cost
  = (index pages × random_page_cost)          -- B-tree traversal
  + (heap pages × effective_page_cost)         -- correlation-adjusted
  + (rows × cpu_index_tuple_cost)              -- 0.005 default
  + (rows × cpu_tuple_cost)                    -- 0.01  default
```

Correlation is the key variable. If `customer_id` rows are inserted in order (e.g. monotonically increasing IDs), heap pages may be nearly sequential — effective cost approaches `seq_page_cost`. If events are scattered across the heap, it approaches `random_page_cost`.

Index cost: correlation matters

At 547K rows, **full correlation** (best case for index scan):

```
heap pages fetched  ≈ 547,000 × 1.0   =   547,000   (sequential-like)
index pages         ≈ 1,500   × 4.0   =     6,000
cpu costs           ≈ 547,000 × 0.015 =     8,205
                                        ----------
Total (correlated)                      ≈   561,205   ← index wins easily
```

At 547K rows, **zero correlation** (worst case — events scattered across heap):

```
heap pages fetched  ≈ 547,000 × 4.0   = 2,188,000   (fully random)
index pages         ≈ 1,500   × 4.0   =     6,000
cpu costs           ≈ 547,000 × 0.015 =     8,205
                                        ----------
Total (uncorrelated)                    ≈ 2,202,205   ← dead even with parallel
```

For an events table with scattered inserts per customer, correlation is low.

The approximation holds. On a perfectly ordered table, the crossover

Why this is silent

- 99,999 customers get index scans — fast, cheap, invisible
- 1 customer crosses **547K events** — parallel seq scan kicks in
- The query still returns correct results
- Nothing in the slow query log
- `pg_stat_statements` average looks fine — diluted by 99,999 fast queries
- **Your most active customer gets the worst plan**

The plan before and after

Under 547K events — Index Scan:

```
Index Scan on events  (cost=0.00..10,000 rows=500,000)
  Index Cond: (customer_id = 42)
```

Over 547K events — silent flip:

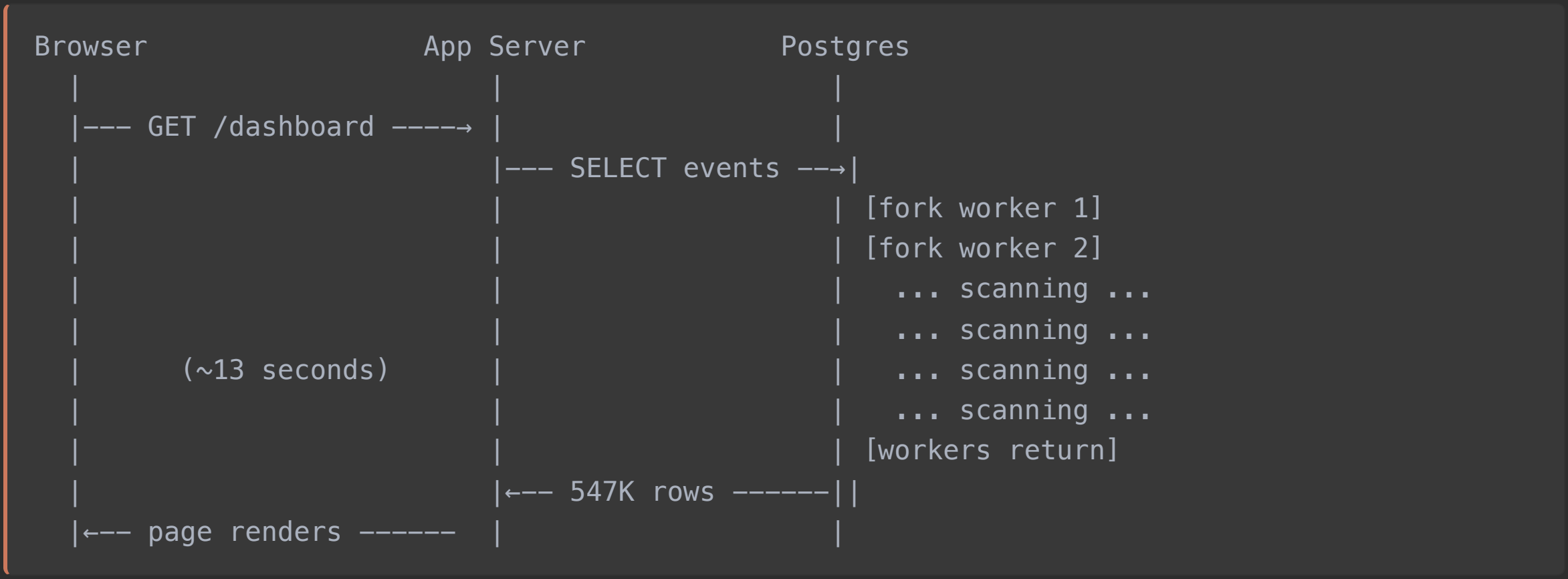
```
Gather  (cost=1000.00..2134583.00 rows=547000)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Seq Scan on events
        Filter: (customer_id = 42)
```

Same query. Same index. Different plan.

Latency goes from **milliseconds to seconds**.

Nobody changed anything.

What your customer experiences



Your most active customer hits their dashboard.

The page spinner runs for **13 seconds**.

Index scan was milliseconds. Nothing changed except event count.

This is the default behavior

You haven't misconfigured anything.

Nobody lowered any cost parameters.

This is a **stock Postgres install**:

```
parallel_setup_cost  = 1000   -- sounds expensive
parallel_tuple_cost  = 0.1    -- doesn't sound like much
min_parallel_table_scan_size = 8MB
```

The planner is doing exactly what it's designed to do.

Your data grew into a shape the defaults weren't built for.

Failure Mode #4

Worker Pool Exhaustion

The math nobody does

```
max_parallel_workers_per_gather = 2   -- default
max_parallel_workers            = 8   -- default
```

- 1 query → 2 workers ✅
- 2 queries → 4 workers ✅
- 3 queries → 6 workers ✅
- 4 queries → 8 workers ✅
- **5th query → 0 workers launched**

The 5th query already paid `parallel_setup_cost` in planning.

And now... it doesn't fall back to serial. It does something worse.

It does NOT fall back to serial

When workers can't launch, Postgres does **not** re-plan.

The leader executes **the same parallel plan** — alone, in one process.

> *"the leader will execute the portion of the plan below the Gather node entirely by itself, almost as if the Gather node were not present."*
> *— PostgreSQL docs, §15.2*

Why this is worse than "just serial"

The planner picked Parallel Seq Scan **because** it assumed 4 workers.

That math beat the Index Scan it would have picked otherwise.

```
Workers Planned: 4
Workers Launched: 0      ← the smoking gun
->  Parallel Seq Scan on events  (250M rows, single-threaded)
```

You're not running a serial plan.
You're running **a parallel plan, serially**, on a single core,
that was only chosen because parallelism was assumed.

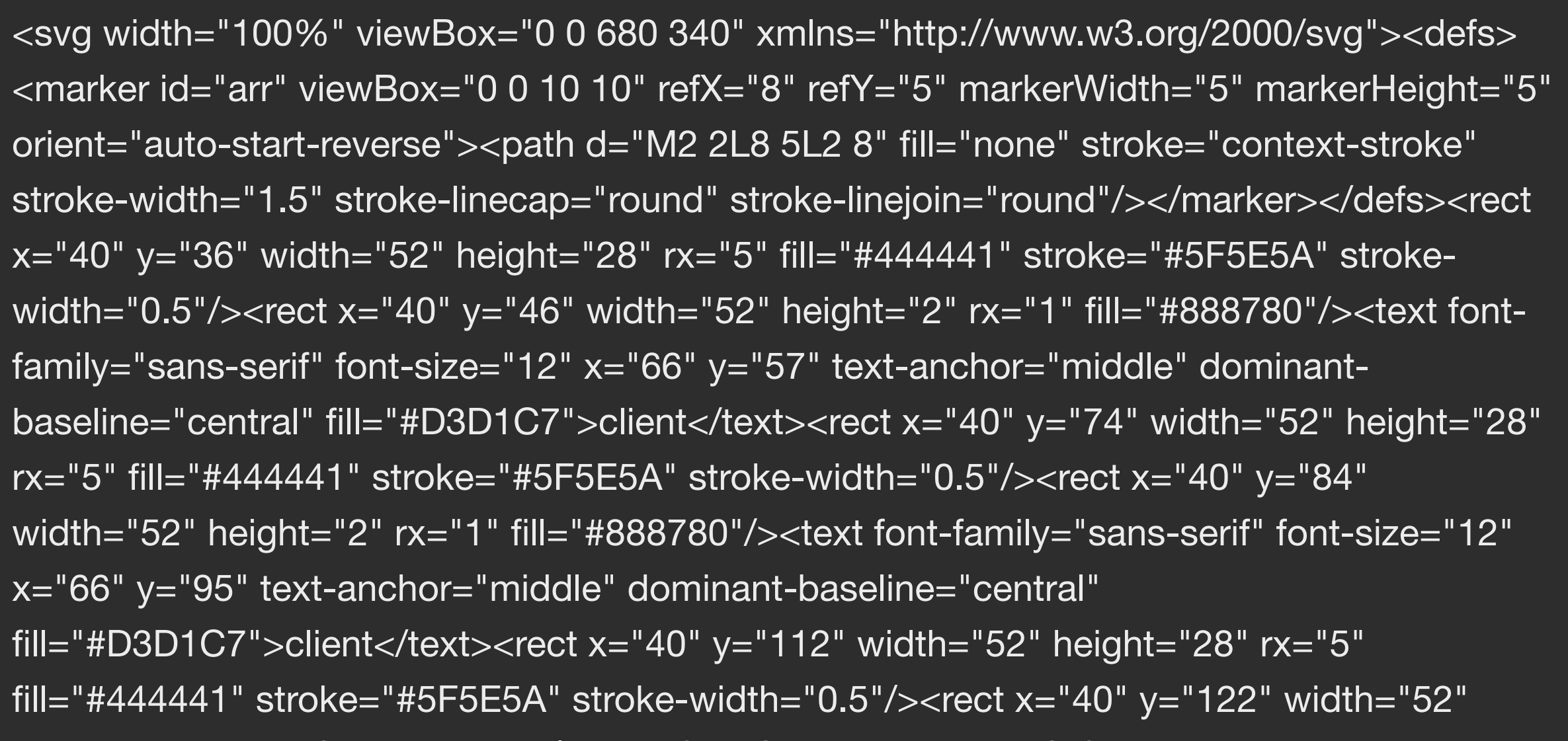
It's worse than disabling parallelism entirely.

What you actually see

```
SELECT pid, backend_type, wait_event_type, wait_event, query
FROM pg_stat_activity
WHERE backend_type IN ('client backend', 'parallel worker');
```

- 8 parallel workers pegged
- 12 client backends in `IPC: ExecuteGather`
- Nothing in the slow query log
- `pg_stat_statements.mean_exec_time` looks "normal-ish"
- Your customers are timing out

The pile-on


<svg width="100%" viewBox="0 0 680 340" xmlns="http://www.w3.org/2000/svg"><defs>
<marker id="arr" viewBox="0 0 10 10" refX="8" refY="5" markerWidth="5" markerHeight="5"
orient="auto-start-reverse"><path d="M2 2L8 5L2 8" fill="none" stroke="context-stroke"
stroke-width="1.5" stroke-linecap="round" stroke-linejoin="round"/></marker></defs><rect
x="40" y="36" width="52" height="28" rx="5" fill="#444441" stroke="#5F5E5A" stroke-
width="0.5"/><rect x="40" y="46" width="52" height="2" rx="1" fill="#888780"/><text font-
family="sans-serif" font-size="12" x="66" y="57" text-anchor="middle" dominant-
baseline="central" fill="#D3D1C7">client</text><rect x="40" y="74" width="52" height="28"
rx="5" fill="#444441" stroke="#5F5E5A" stroke-width="0.5"/><rect x="40" y="84"
width="52" height="2" rx="1" fill="#888780"/><text font-family="sans-serif" font-size="12"
x="66" y="95" text-anchor="middle" dominant-baseline="central"
fill="#D3D1C7">client</text><rect x="40" y="112" width="52" height="28" rx="5"
fill="#444441" stroke="#5F5E5A" stroke-width="0.5"/><rect x="40" y="122" width="52"
height="2" rx="1" fill="#888780"/><text font-family="sans-serif" font-size="12" x="66"

The problem with EXPLAIN

`EXPLAIN ANALYZE` shows `Workers Planned: 2, Workers Launched: 0` —
but only **if you run it while the system is under load**.

Run it on an idle box and workers launch just fine.
The problem disappears. You find nothing.

This is not a debugging tool for worker exhaustion.
You need **cumulative counters**, not point-in-time plans.

pg_stat_database + pg_stat_statements (PG 18+)

```sql
-- Cluster-level: is the pool being starved?
SELECT datname,
       parallel_workers_to_launch,
       parallel_workers_launched,
       parallel_workers_to_launch
         - parallel_workers_launched AS gap
FROM pg_stat_database
WHERE datname = current_database();

-- Query-level: which queries are losing workers?
SELECT query,
       parallel_workers_to_launch,
       parallel_workers_launched,
       parallel_workers_to_launch
         - parallel_workers_launched AS gap
FROM pg_stat_statements
WHERE parallel_workers_to_launch > 0
ORDER BY gap DESC
LIMIT 10;
```

Failure Mode #5

Parallel Append × Partitions

A multiplier you didn't ask for

```
CREATE TABLE events_part (...) PARTITION BY RANGE (created_at);
-- 12 monthly partitions, growing every month
```

```
SELECT event_type, count(*) FROM events_part GROUP BY event_type;
```

```
Finalize GroupAggregate
  -> Gather (workers planned: 2)
    -> Parallel Append
      -> Parallel Seq Scan on events_p202504
      -> Parallel Seq Scan on events_p202505
      ... 12 partitions ...
```

2 workers shared across 12 partitions — not 2 per partition.

But the number of workers *requested* scales with partition count.

And it grows over time

The number of workers requested scales roughly logarithmically with partition count.

Partitions	Workers requested
6	2
12	3
24	4
48	5

That number is capped by `max_parallel_workers_per_gather` — but as partitions grow, the planner keeps asking for more. Nobody noticed the partition count doubled. Nobody re-audited the plan. The query just got slower and hungrier.

> *Parallel features compound. Audit them when your data shape changes.*

* Postgres also factors in partition size via compute_parallel_worker()

Detection

How do you find this in your fleet today?

Find the suspects

```sql
-- Live: are workers being starved right now?
SELECT count(*) FILTER (WHERE backend_type = 'parallel worker') AS workers,
       count(*) FILTER (WHERE state = 'active')                 AS active_clients
FROM pg_stat_activity;

-- Live: who is waiting on parallel infrastructure?
SELECT pid, wait_event_type, wait_event, query
FROM pg_stat_activity
WHERE wait_event IN ('BgWorkerShutdown', 'BgWorkerStartup', 'ExecuteGather', 'ParallelFinish');
```

If `workers` is regularly at `max_parallel_workers`, you have a problem.

Sessions on `ExecuteGather` are blocked waiting for parallel results.

Sessions on `ParallelFinish` are waiting for workers to finish and clean up.

Sessions on `BgWorkerStartup` / `BgWorkerShutdown` are in worker lifecycle overhead.

Find the suspects (PG 18+)

```sql
-- Cumulative: which queries are losing workers?
SELECT query,
       parallel_workers_to_launch,
       parallel_workers_launched,
       parallel_workers_to_launch
         - parallel_workers_launched AS gap
FROM pg_stat_statements
WHERE parallel_workers_to_launch > 0
ORDER BY gap DESC
LIMIT 10;

-- Cumulative: is the pool being starved cluster-wide?
SELECT datname,
       parallel_workers_to_launch,
       parallel_workers_launched,
       parallel_workers_to_launch
         - parallel_workers_launched AS gap
FROM pg_stat_database
WHERE datname = current_database();
```


A persistent gap means queries are silently running parallel plans on a single core.

Taking Back Control

The philosophy

Parallel query has its place.

If a query, data model, and data distribution are designed correctly,
it can provide tangible benefits.

The trick is ensuring **you** are in control of the parallelism
in use at any given time — **globally, across the cluster**.

The problem with every partial fix is the same:
you can't bound total worker count across concurrent sessions.

Step 1: Disable globally

```
-- postgresql.conf / ALTER SYSTEM
ALTER SYSTEM SET max_parallel_workers_per_gather = 0;
SELECT pg_reload_conf();
```

This is not the nuclear option. **This is the baseline.**

Parallel query is now off for all workloads, all users, all tables.

Nothing happens without your explicit permission.

Step 2: Enable intentionally

Per-query, using session variables:

```
SET max_parallel_workers_per_gather = 4;
SELECT count(*) FROM events WHERE created_at > now() - interval '1 year';
RESET max_parallel_workers_per_gather;
```

Per-query, using pg_hint_plan:

```
/*+ Parallel(events 4 hard) */
SELECT count(*) FROM events WHERE created_at > now() - interval '1 year';
```

`hard` means use exactly 4 workers — not "up to 4".

You asked for it. You got it. Nothing surprises you.

pg_hint_plan docs: https://pg-hint-plan.readthedocs.io/en/latest/hint_details.html

Step 3: Enable per-user or per-database

Only if you can **guarantee a max concurrency count**.

```
-- Analytical user gets parallelism, OLTP users don't
ALTER ROLE analyst SET max_parallel_workers_per_gather = 4;

-- Or scope to a specific database
ALTER DATABASE analytics SET max_parallel_workers_per_gather = 4;
```

If 10 analyst sessions can run concurrently and each gets 4 workers, you need `max_parallel_workers ≥ 40`. Know your math before you enable.

Why per-table is not enough

```
ALTER TABLE events SET (parallel_workers = 4);
```

This caps workers *per query on this table*.
It does not cap *total workers across all concurrent sessions*.

10 sessions hit `events` simultaneously → up to 40 workers.
You have no cluster-level guarantee.

And when that 11th session arrives and the worker pool is exhausted?
Your analyst gets no error. No warning. No slow query log entry.
They get a parallel plan running on a single CPU —
untraceable, silent, and slower than if parallelism had never been enabled.

In a mixed workload system, per-table settings trade
one instability for another.

Takeaways

What to do Monday morning

1. **Disable parallel query globally.** Make it your baseline, not your last resort.
2. **Re-enable intentionally** — per-query via session GUCs or `pg_hint_plan`.
3. **Per-user or per-database** only when you can bound concurrency.
4. **Add a worker-count panel** to your monitoring — `pg_stat_activity`.
5. **Watch for** `Workers Launched: 0` — the silent parallel plan with no workers.

> *Parallelism is not opt-in by default.*
> *Make it opt-in by design.*

Questions?

Scott Mead · meads@amazon.com