dbtune

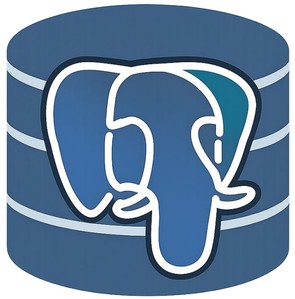
Why your PostgreSQL tuning guide might be wrong (and what to do about it)

Mohsin Ejaz
Senior DevOps Engineer

About me

- Nearly two decades in DevOps, QA, and Release Management with PostgreSQL
- 17 years at EnterpriseDB
- Specializing in CI/CD, automation, and database benchmarking
- Currently: DevOps,Pre Sales and Customer success – focusing on reliability and performance
- Focus: Ensuring database performance isn't just a "best case" scenario, but a reliable, automated reality for global enterprises

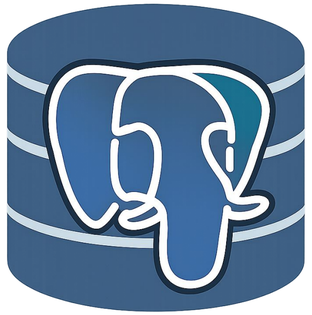
Same PostgreSQL version. Same workload.



PostgreSQL

~ 6ms

?

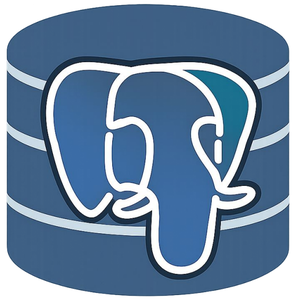


PostgreSQL

~ 0.06 ms

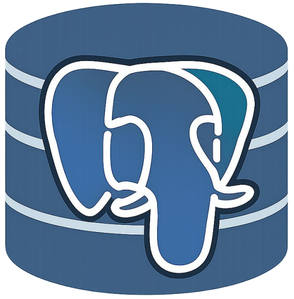
Same PostgreSQL.

Different defaults



Upstream PostgreSQL

JIT : ON



Managed PostgreSQL

JIT : OFF

What you will learn today

Three patterns

1. Different starting points
2. Infrastructure changes the math
3. Interactions & amplifications

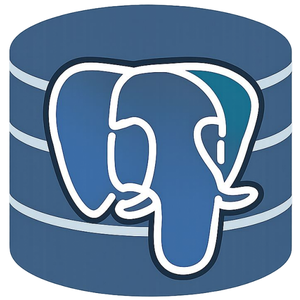
Practical takeaways

- How to test on YOUR System
- Infrastructure checklist
- Data-driven tuning workflow
- Tools you can use today

Pattern 1

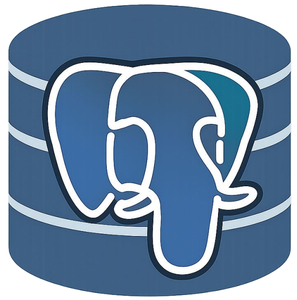
Different starting points

Example: JIT defaults in managed PostgreSQL



Upstream PostgreSQL

JIT : ON



Managed PostgreSQL

(RDS)

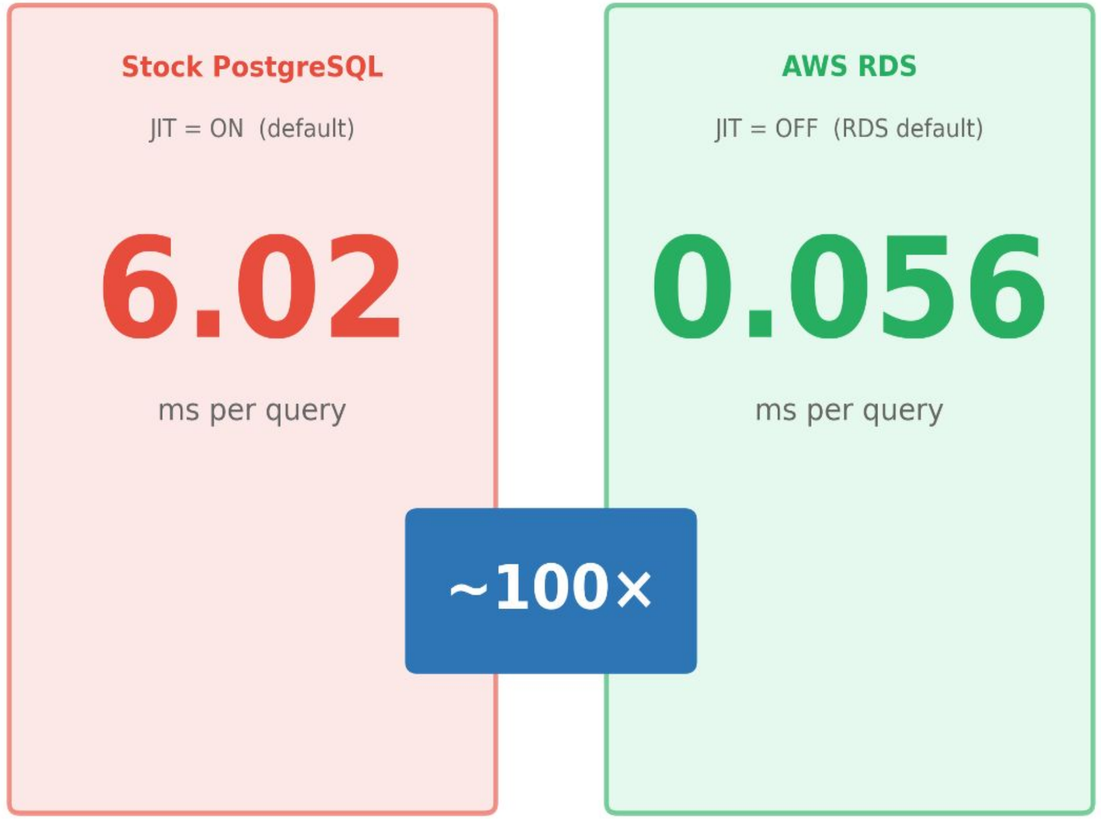
JIT : OFF

The query — nothing exotic

Same workload. Same PostgreSQL 16. Same data.

```
SELECT u.u_id, u.name,
       COUNT(r.rating) AS num_ratings
FROM   useracct u
JOIN   item_rating r ON r.u_id = u.u_id
WHERE  u.u_id = $1
GROUP  BY u.u_id, u.name;
```

Epinions benchmark — cached OLTP query



Epinions benchmark · PostgreSQL 16 · cached OLTP

Real Production Query

Django app · Adam Johnson (Django Steering Council)

Before		After
3,217 ms	jit = off →	10 ms

What EXPLAIN ANALYZE showed:

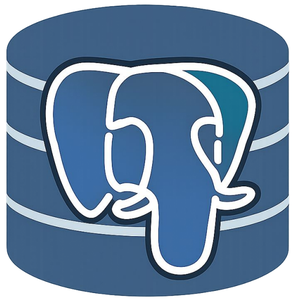
99.7% of 3,217 ms was JIT compilation

The query itself ran in milliseconds.
The compiler took 3 seconds.

99.7% faster — one config line

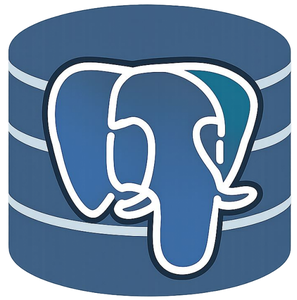
Source : https://adamj.eu/tech/2023/11/09/django-disable-postgresql-jit/

Example: Aurora specific behavior



Upstream PostgreSQL

Standard optimizer



Managed PostgreSQL

Custom optimizer

Aurora has a custom optimizer based on benchmarking & telemetry

Aurora's custom optimizer rewrites the query

Same SQL — very different execution plans · (Source: AWS Database Blog)

The query — standard SQL

```
SELECT ot.a, ot.b
FROM   ot
WHERE  ot.b < (
    SELECT AVG(it.b)
    FROM   it
    WHERE  it.a = ot.a
);
```

Correlated subquery — runs once per outer row without optimisation

Stock PostgreSQL

```
Seq Scan on ot
  Filter: (b < (SubPlan 1))
  SubPlan 1
    → Aggregate
      → Seq Scan on it
            Filter: (a = ot.a)
```

Subquery re-executes for every row in ot

Aurora PostgreSQL

apg_enable_correlated_scalar_transform = ON (Aurora default)

```
Hash Join
  Hash Cond: (ot.a = apg_scalar_subquery.scalar_output)
  Join Filter: (ot.b < apg_scalar_subquery.avg)
  →  Seq Scan on ot
  →  Hash
       →  Subquery Scan on apg_scalar_subquery
            →  HashAggregate
                 Group Key: it.a
                 →  Seq Scan on it
```

Subquery runs ONCE

Result is hash-joined, not looped

Same SQL, different plan

Aurora silently rewrites it

Source : https://aws.amazon.com/blogs/database/optimizing-correlated-subqueries-in-amazon-aurora-postgresql/

Pattern 1 – What to do

Know your starting point

1. **SHOW ALL;**

   Know your real baseline

2. **Compare with upstream defaults**

   Not assumptions - actual defaults

3. **Read provider documentation**

   Defaults reflect intent

Pattern 2

Infrastructure changes the math

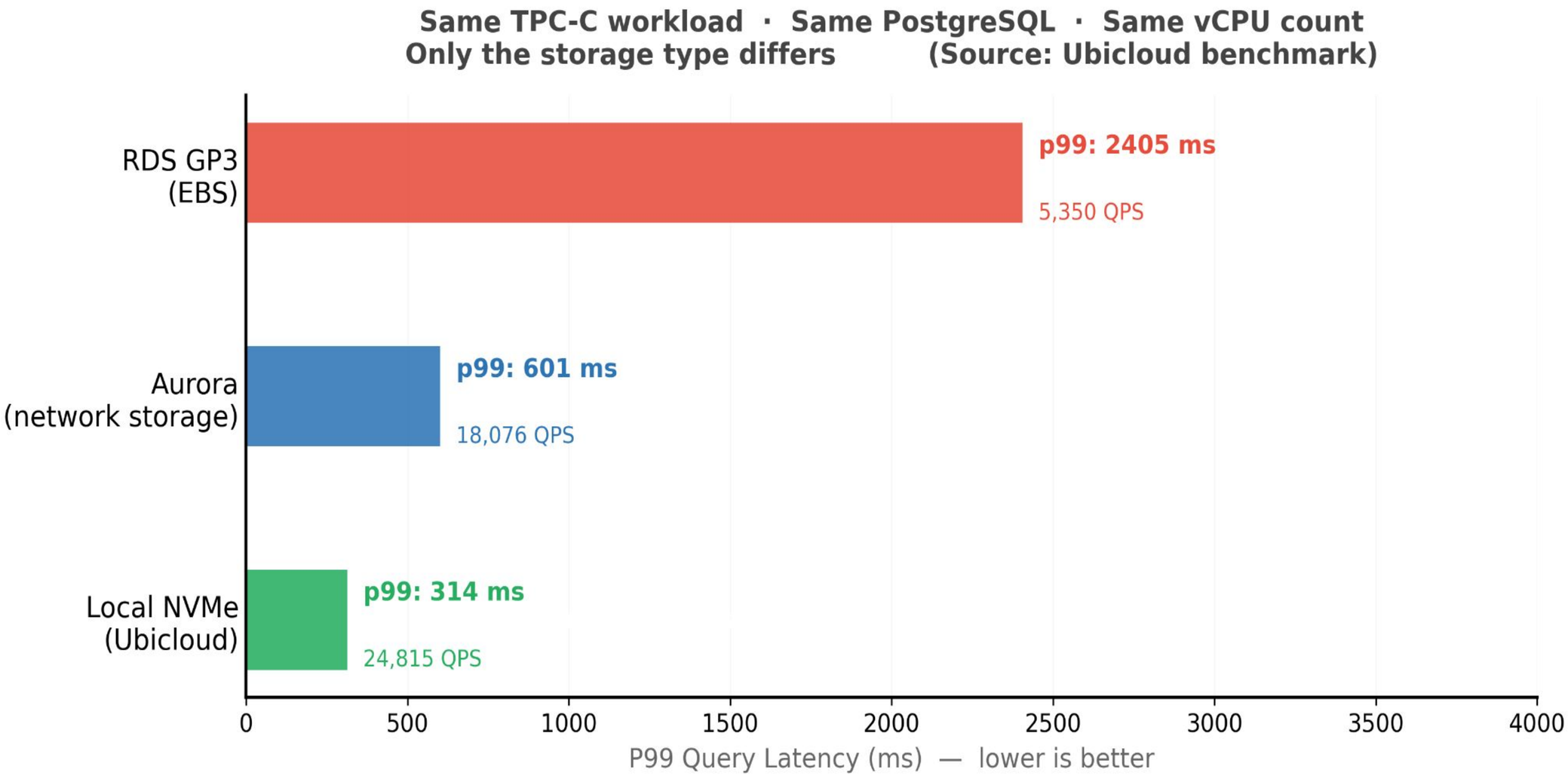
Storage changes I/O cost

Local storage

Predictable Latency

Network - attached storage

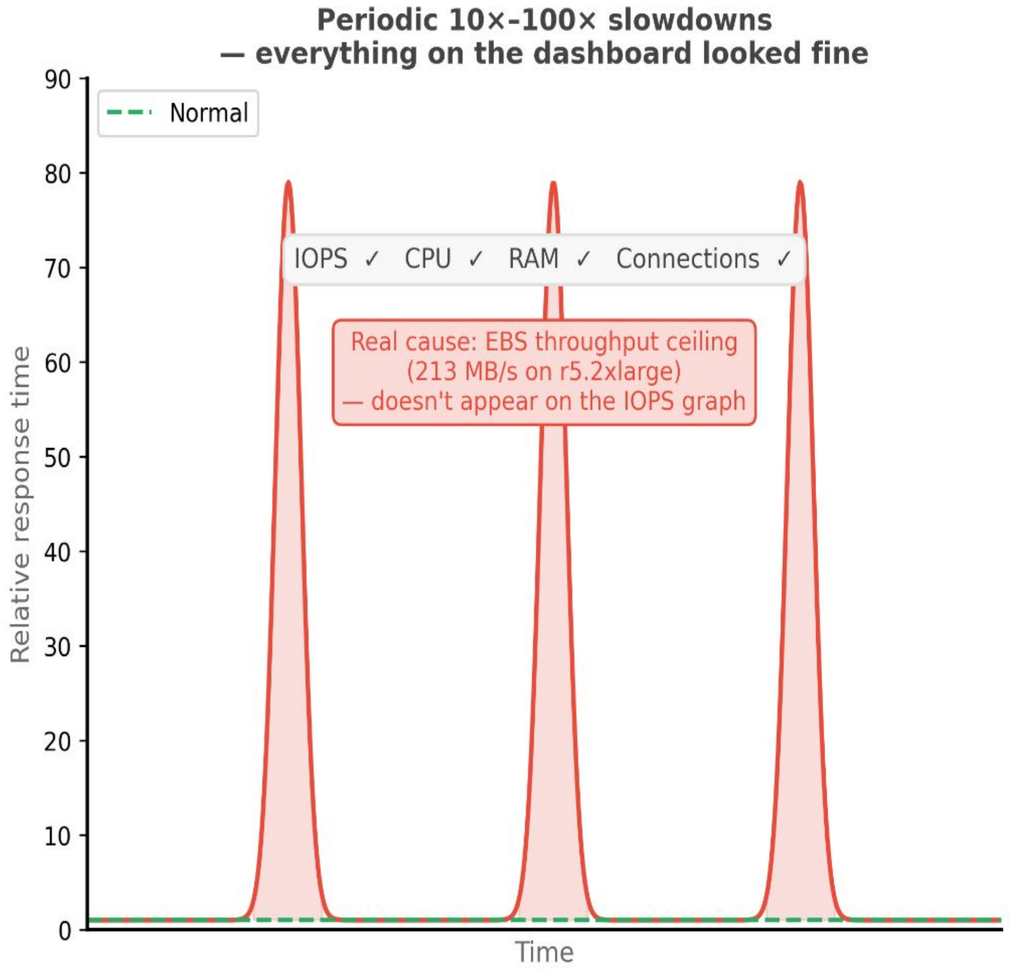
Added latency & variability

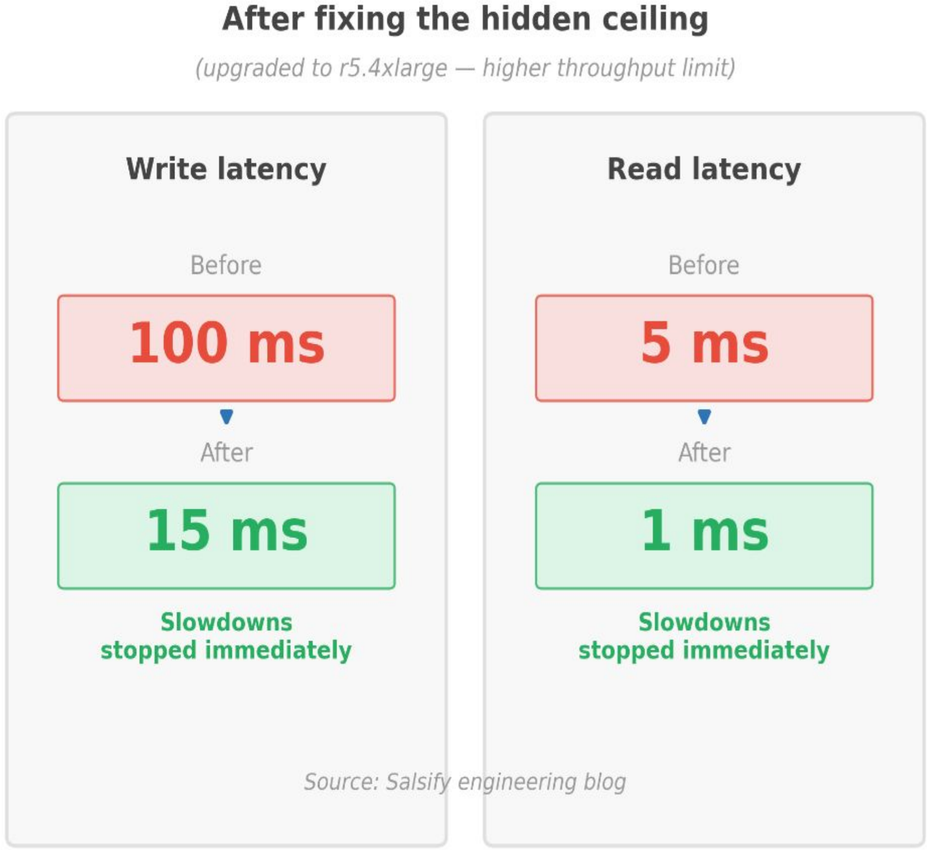


Source : https://www.ubicloud.com/blog/postgresql-performance-local-vs-network-attached-storage

Storage variability breaks cost assumptions

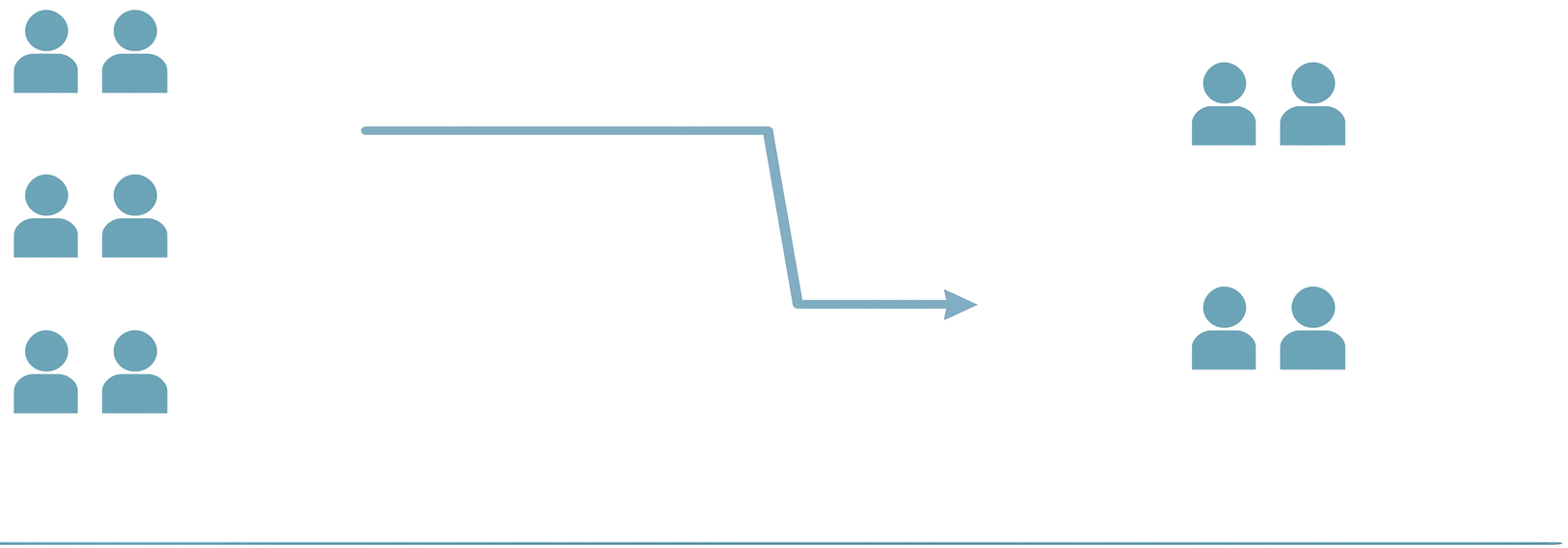
- Same parameter
- Same PostgreSQL
- Different infrastructure
- Different math





Source : https://www.salsify.com/blog/engineering/discovering-rds-throughput-limits

Compute changes parallelism math



The query — SEATS benchmark

Frequent-flyer lookup · DBtune benchmark

```
SELECT c.C_ID,
       ff.FF_AL_ID
FROM   customer c
JOIN   frequent_flyer ff
       ON ff.FF_C_ID = c.C_ID
WHERE  ff.FF_C_ID_STR = $1;
```

FF_C_ID_STR is effectively unique.

The planner estimated 1,241 rows.

It returned 1.

Parallelism ON

`max_parallel_workers_per_gather = 1`

Execution time:

92

ms

What the planner chose:

```
Gather  (Workers: 1)
→ Nested Loop  ×2
  Est. rows:  1,241
  Actual rows:    1
  Coordination: wasted
```

Worker overhead > query work

Parallelism OFF

`max_parallel_workers_per_gather = 0`

Execution time:

0.107

ms

What the planner chose:

```
Nested Loop
→ Index Scan (ff)
  FF_C_ID_STR = $1
→ Index Only Scan (c)
  No coordination needed
```

No coordination tax — straight to result

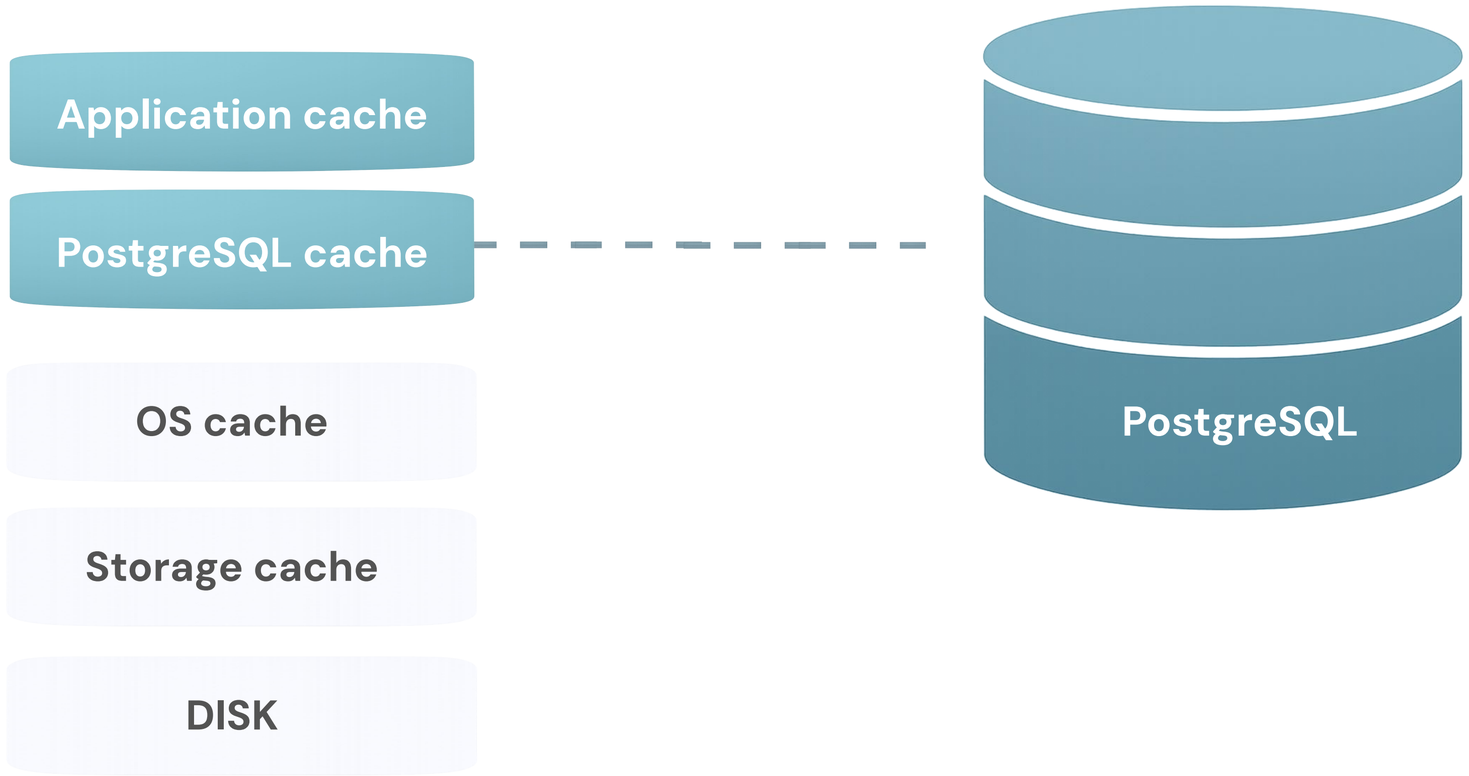
SEATS benchmark · DBtune internal data · Same query, same PostgreSQL, same data · Only parallelism setting differs

Compute - What to do

Tune for CPU stability

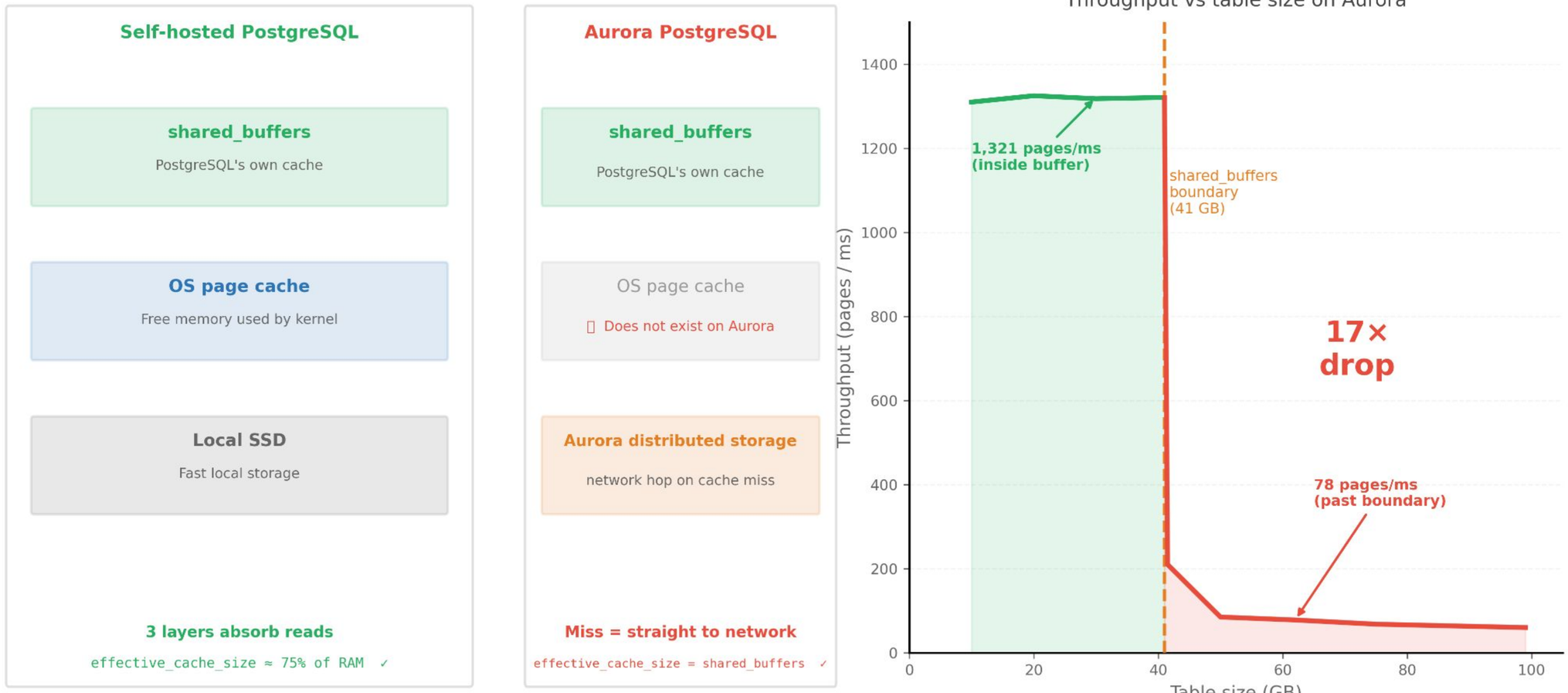
- Is CPU availability consistent or variable ?
- Observe saturation during real workload spikes
- Validate parallel plan under stress
- Tune for worst-case, not best-case conditions

Caching breaks planner assumptions



Aurora has no OS cache layer — cross the buffer boundary and throughput falls off a cliff

db.r6g.2xlarge · 64 GB RAM · shared_buffers = 41 GB · Source: AWS Heroes / dev.to



Source: dev.to/aws-heroes/amazon-aurora-and-postgresql-buffer-cache-n67 · Data: db.r6g.2xlarge, PostgreSQL 14.9, shared_buffers = 41 GB

Source : https://dev.to/aws-heroes/amazon-aurora-and-postgresql-buffer-cache-n67

Caching - What to do

Tune for what PostgreSQL actually sees

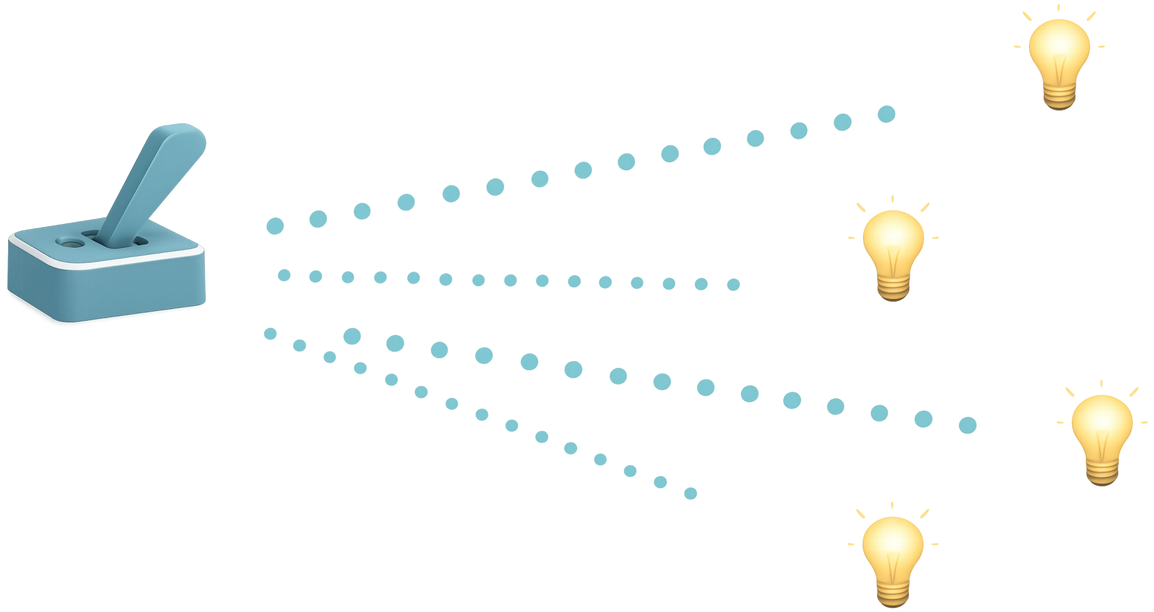
- Don’t tune blindly for read latency if reads are absorbed elsewhere.
- Focus tuning on what actually hits PostgreSQL: writes, cache misses, maintenance operations.
- Validate assumptions with metrics, not intuition.

Pattern 2 summary

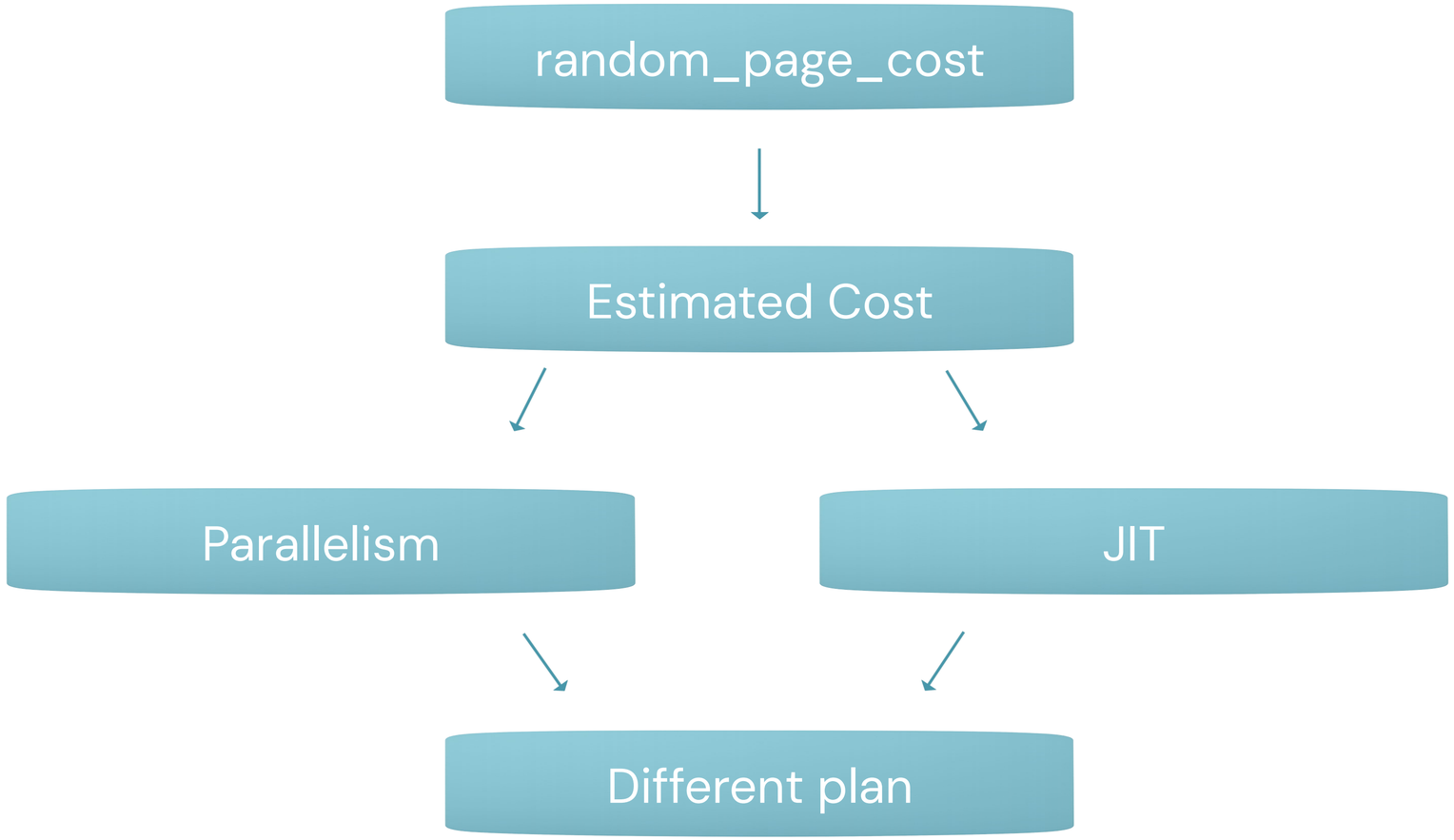
- Storage change I/O cost
- Compute changes CPI cost
- Caching hides reality

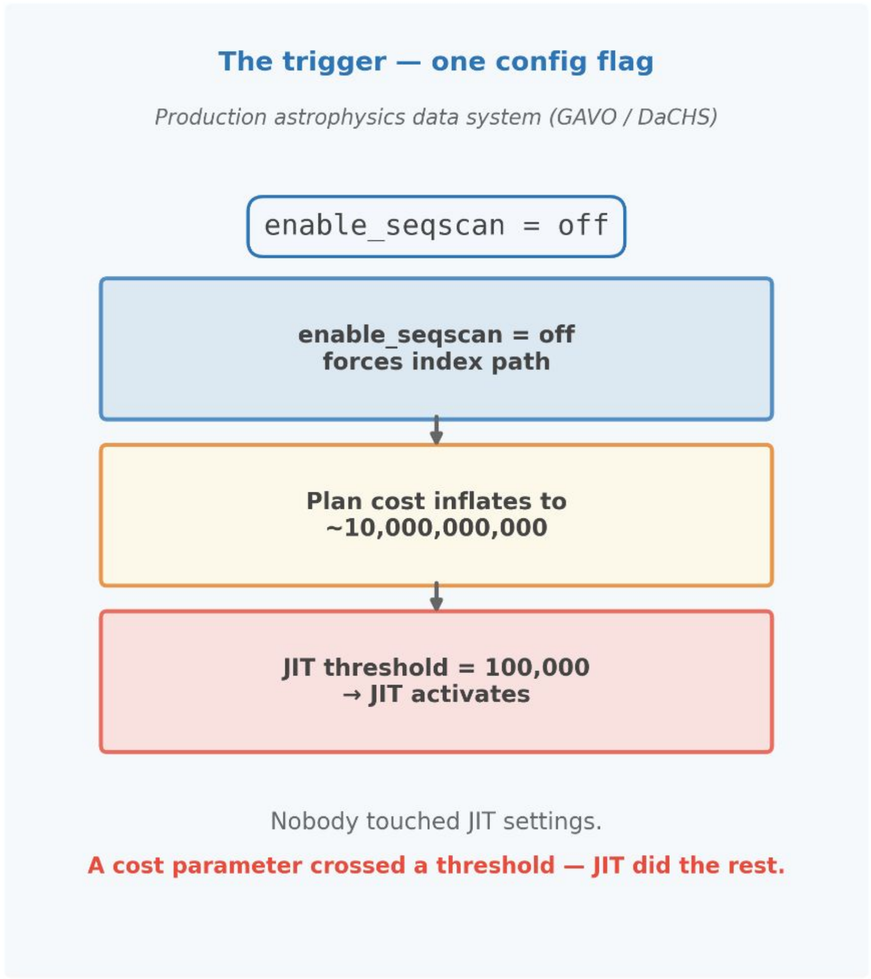
Pattern 3

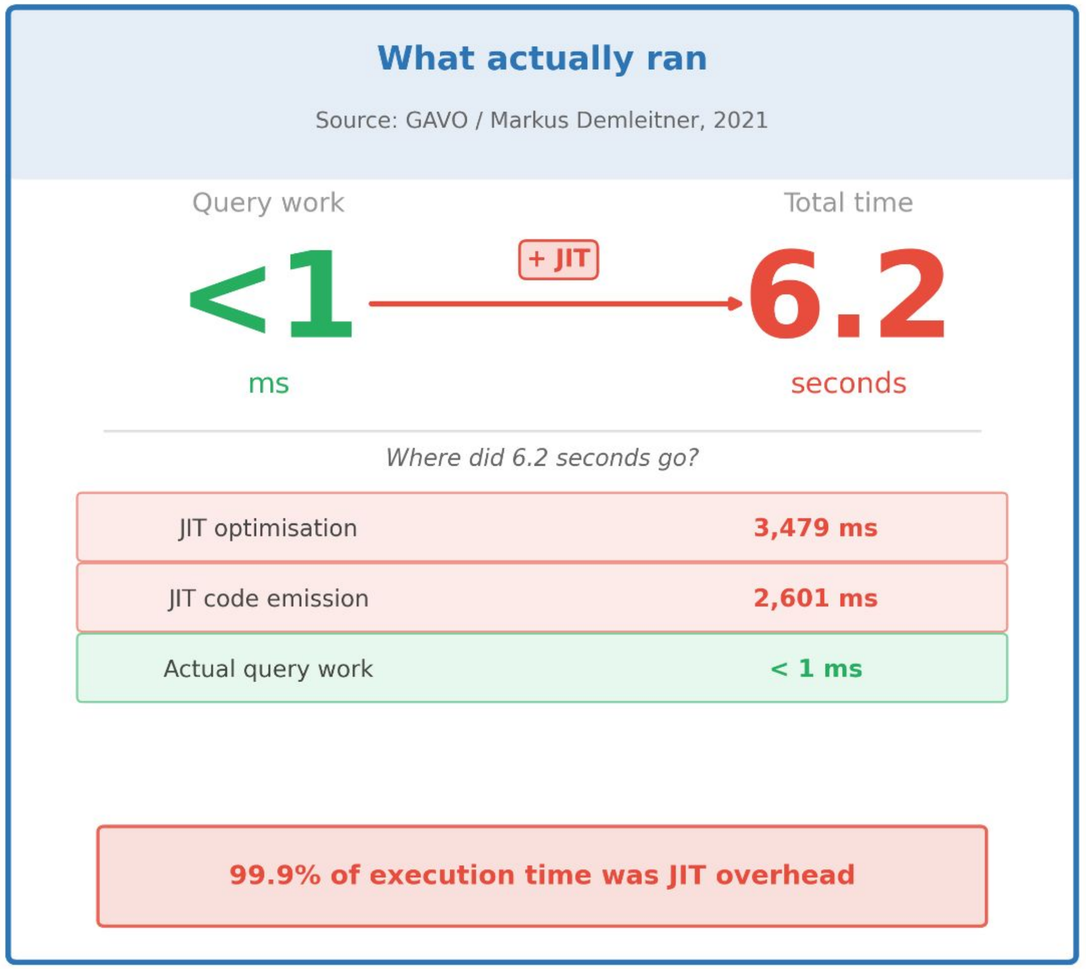
Interactions and amplifications



Interactions: The cascade effect







Source : https://blog.g-vo.org/taming-the-postgres-jit.html

Handling cascades safely

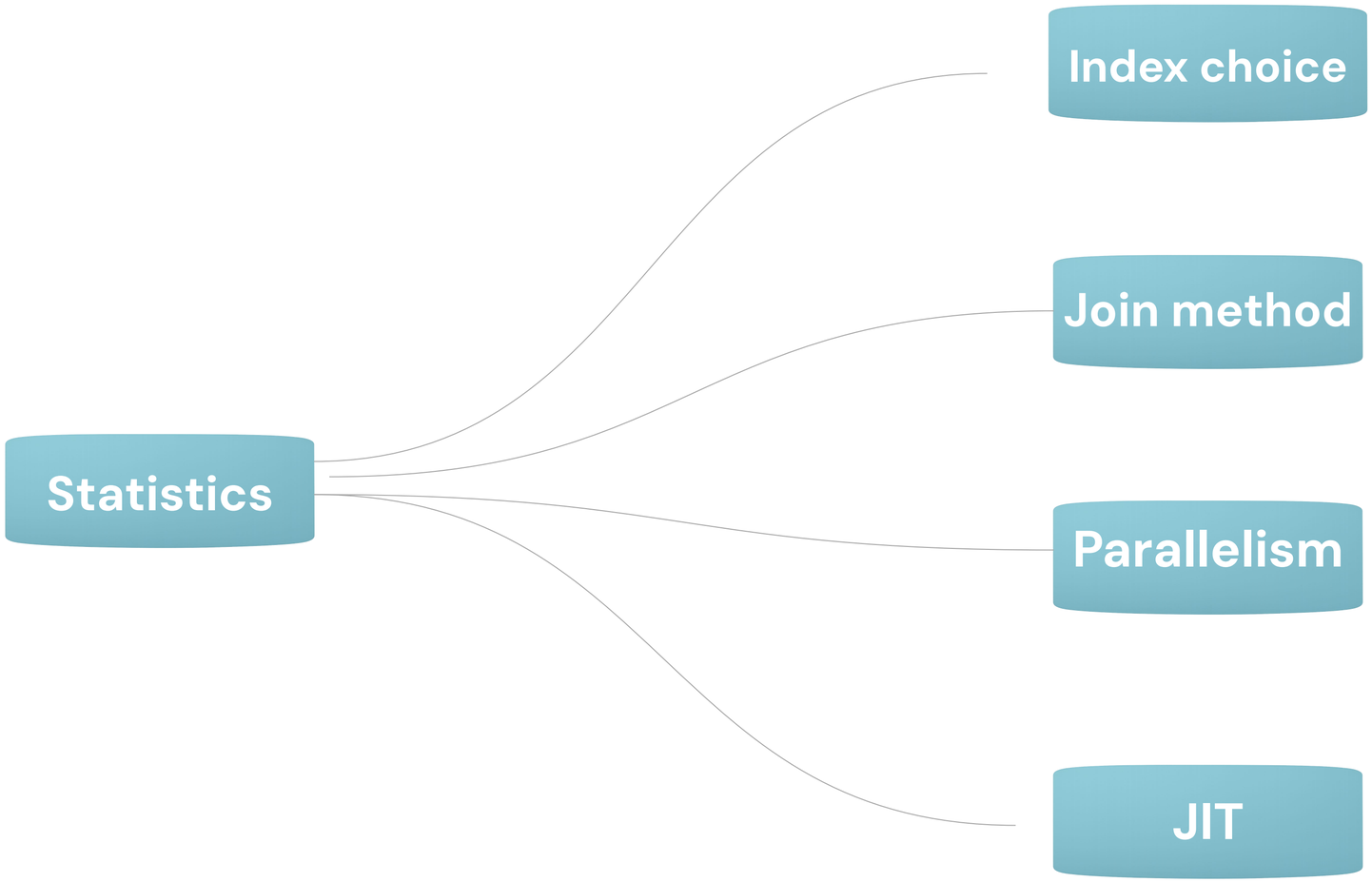
Don’t only ask: “Is this parameter good or bad?”

Ask: “What else does this unlock?”

Before - After → Compare

- Plan Shape
- Parallelism
- JIT

Statistics amplify everything



The query — SEATS benchmark

Lookup by frequent-flyer ID

```
SELECT c.c_id,
       c.c_first_name,
       c.c_last_name,
       c.c_balance
FROM   customer c
WHERE  c.c_id = $1;
```

c_id is effectively unique.

Statistics said otherwise.

What the planner saw

Statistics were stale

Estimated rows:

1,241

rows

Actually returned:

1

row

1,000× estimation error

Performance impact

Parallel plan chosen for 1 row

Workers launched — 0-row result

Result:

~600×

slower

Fix:

ANALYZE

\+ increase statistics target

The query — real production

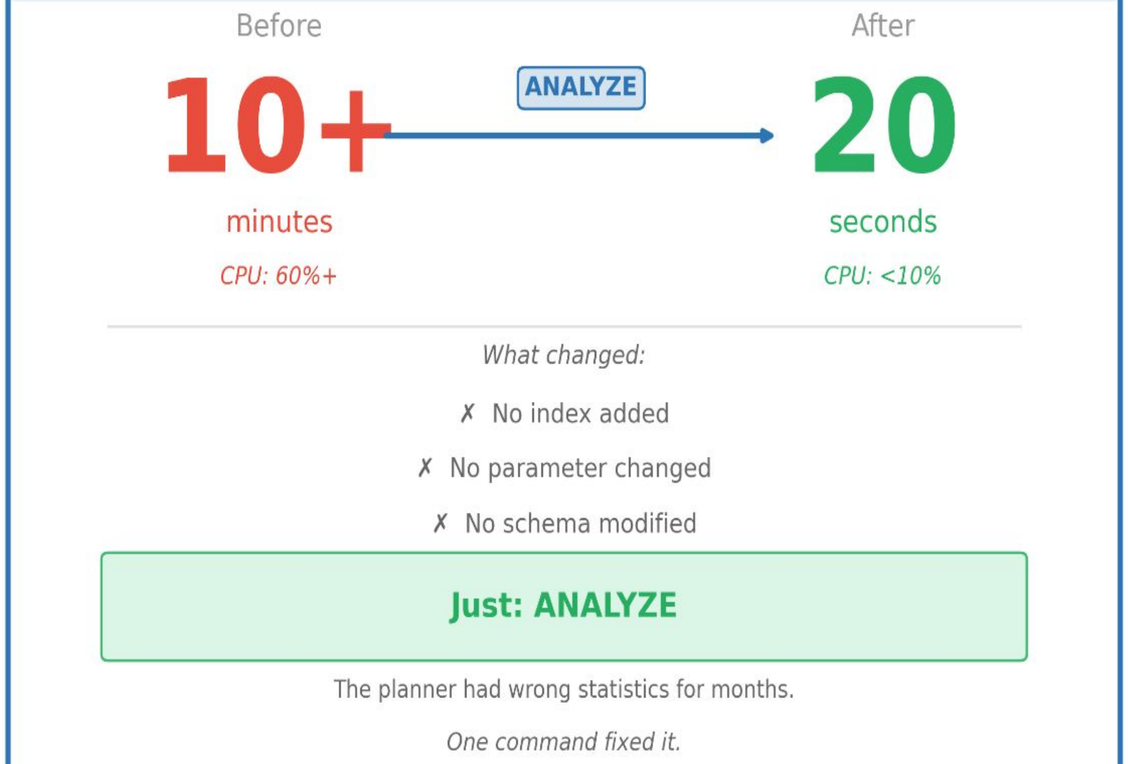
6-table JOIN · never analyzed since initial load

```
SELECT o.order_id,
       c.name,
       p.product_name,
       w.warehouse_city,
       s.shipped_at,
       i.qty_on_hand
FROM   orders o
JOIN   customers  c ON c.id      = o.customer_id
JOIN   products   p ON p.id      = o.product_id
JOIN   shipments  s ON s.order_id = o.order_id
JOIN   inventory  i ON i.prod_id  = o.product_id
JOIN   warehouses w ON w.id      = i.warehouse_id
WHERE  o.status = 'pending';
```

Planner had no statistics. Chose worst join order.

Real Production Database

Source: Stormatics



What changed:

✗ No index added

✗ No parameter changed

✗ No schema modified

Just: ANALYZE

The planner had wrong statistics for months.

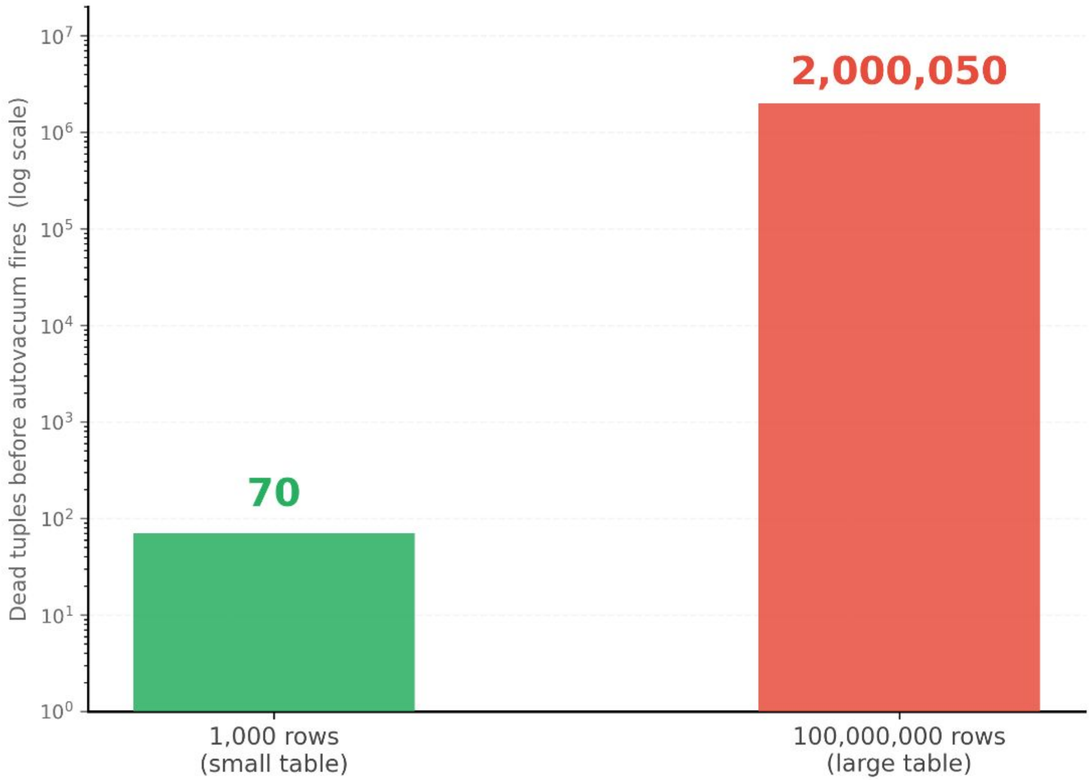
One command fixed it.

Source : https://stormatics.tech/blogs/dont-skip-analyze-a-real-world-postgresql-story

Same autovacuum formula — very different outcome at scale

threshold = 50 + (table_rows × 0.02)

`autovacuum_vacuum_threshold + ( rows × autovacuum_vacuum_scale_factor )`



1,000 rows → 70 dead tuples

`50 + (1,000 × 0.02) = 70`

- ✓ **Fires quickly**
- ✓ Stats stay fresh
- ✓ Default works fine here

100M rows → 2,000,050 dead tuples

`50 + (100,000,000 × 0.02) = 2,000,050`

- ✗ **Waits for 2M dead tuples**
- ✗ Stats drift during bulk ops
- ✗ Bloat builds → CPU spikes

`Fix: autovacuum_vacuum_scale_factor = 0.001`
`→ fires after ~100K dead tuples instead`

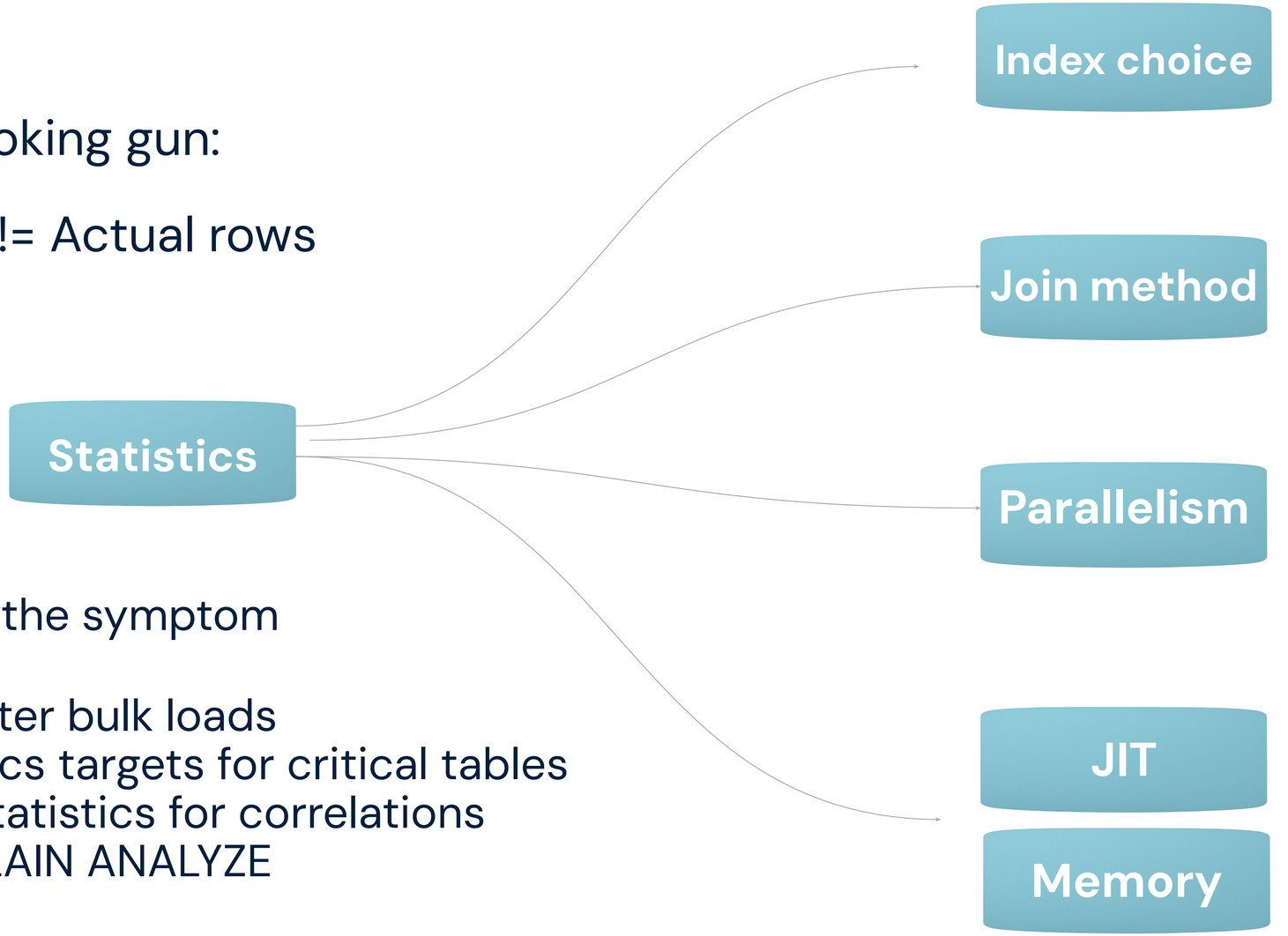
Source: stormatics.tech · Default scale_factor = 0.02 · Log scale used — actual difference is ~28,000×

Source : https://stormatics.tech/blogs/scenarios-that-trigger-autovacuum-in-postgresql

Handling statistical amplification

Look for the smoking gun:

Estimated rows != Actual rows



Fix the cause, not the symptom

- Run ANALYZE after bulk loads
- Increase statistics targets for critical tables
- Use extended statistics for correlations
- Verify with EXPLAIN ANALYZE

Pattern 3 – Emergent behaviour

Small changes

Large systems

Delayed effects

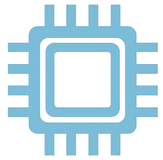
That's why tuning sometimes looks random.

It isn’t.

How to test on YOUR system-CHECKLIST

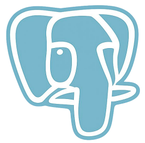
Storage

- Local or network - attached ?
- Stable or variable under load?



Compute

- Dedicated or burstable CPU?
- CPU limits (VMs, containers)



PostgreSQL

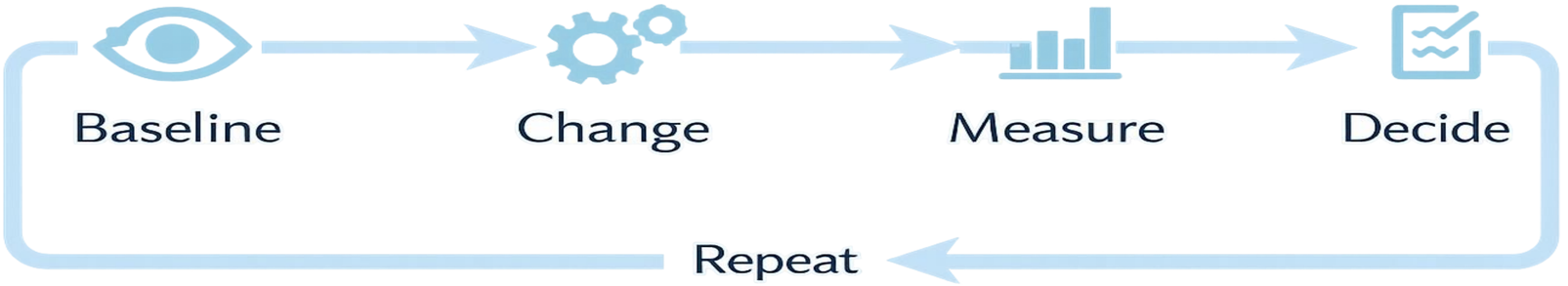
- Self hosted or managed?
- SHOW ALL; vs upstream defaults
- Provider docs = changed assumptions

Workload

- OLTP or analytics?
- Read heavy or write heavy?
- Caching above PostgreSQL?

Step 2 — Measure on YOUR system

Treat tuning like science, not folklore



What to measure

Workload

- pg_stat_statements
- Top queries by total time
- Focus on top 5 (80/20 rule)

Queries

- EXPLAIN ANALYZE
- Estimated vs Actual rows
- Plan changes,parallelism,JIT

Infrastructure

- IOPS,CPU saturation
- Credit exhaustion
- PostgreSQL doesn’t see these– you must

Before —› Change —› After —› Decide

Tools (to support good thinking)

Query analysis

- pg_stat_statements
- EXPLAIN ANALYZE
- explain.dalibo.com
- pg_stat_plans

Infrastructure metrics

- CloudWatch / Azure monitor / Cloud monitoring
- Prometheus + Grafana
- pganalyze

Starting point

- PGTune
- SHOW ALL;
- Provider docs

Automation

- DBtune

Tools don’t replace thinking. They reduce blind spots.

Why the puzzle wasn’t a bug

Same PostgreSQL

Same workload

Very different performance

Three patterns

- ✓ Different starting points
- ✓ Infrastructure changes the math
- ✓ Interaction & Amplification

What good PostgreSQL tuning looks like

Not memorizing parameters

But building a feedback loop

Checklist

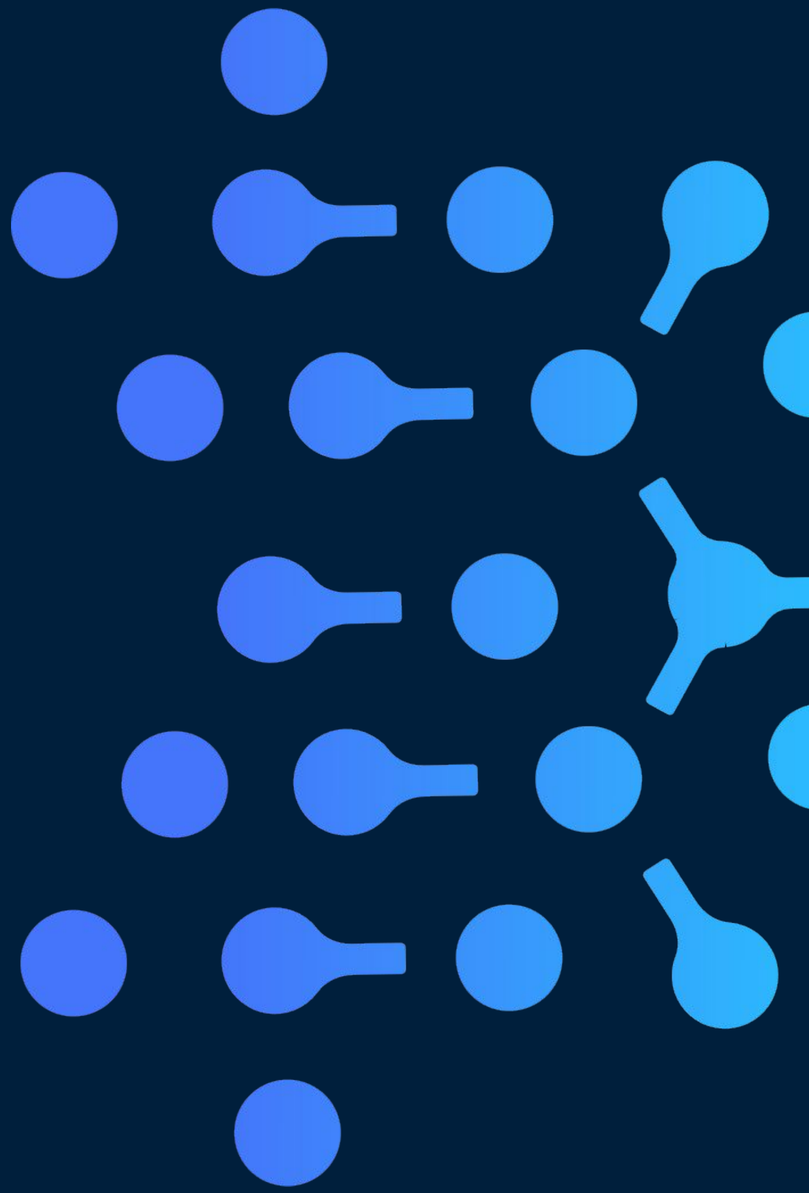
- ✓ Understand your system
- ✓ Form a hypothesis
- ✓ Change one thing
- ✓ Measure after
- ✓ Decide with data

Tuning guides are written for a system.

You are running a different one. Prove it on YOUR system

dbtune

Q&A



Thank you

Mohsin Ejaz

mohsin@dbtune.com