STOP GUESSING:

HOW TO ACTUALLY FIX POSTGRESQL PERFORMANCE

Anita Singh
Principal Database Specialist SA, AWS

Ranjan Burman
Senior Database Specialist SA, AWS

Postgres Conference San Jose 2026

Misconceptions or Misunderstood ?

1. Vacuum is hogger – Shut that thing down…
2. PostgreSQL planner ignores the indexes
3. EXPLAIN plan estimates are so off
4. Those Mysterious LW:Locks appearing out of nowhere
5. Your connections are eating up memory

By the end of this session, you'll know how to diagnose and fix each one.

Session Overview

💡 Resources

🔍 Query Execution Plans, Statistics , Indexes

📁 Field Case Studies on :

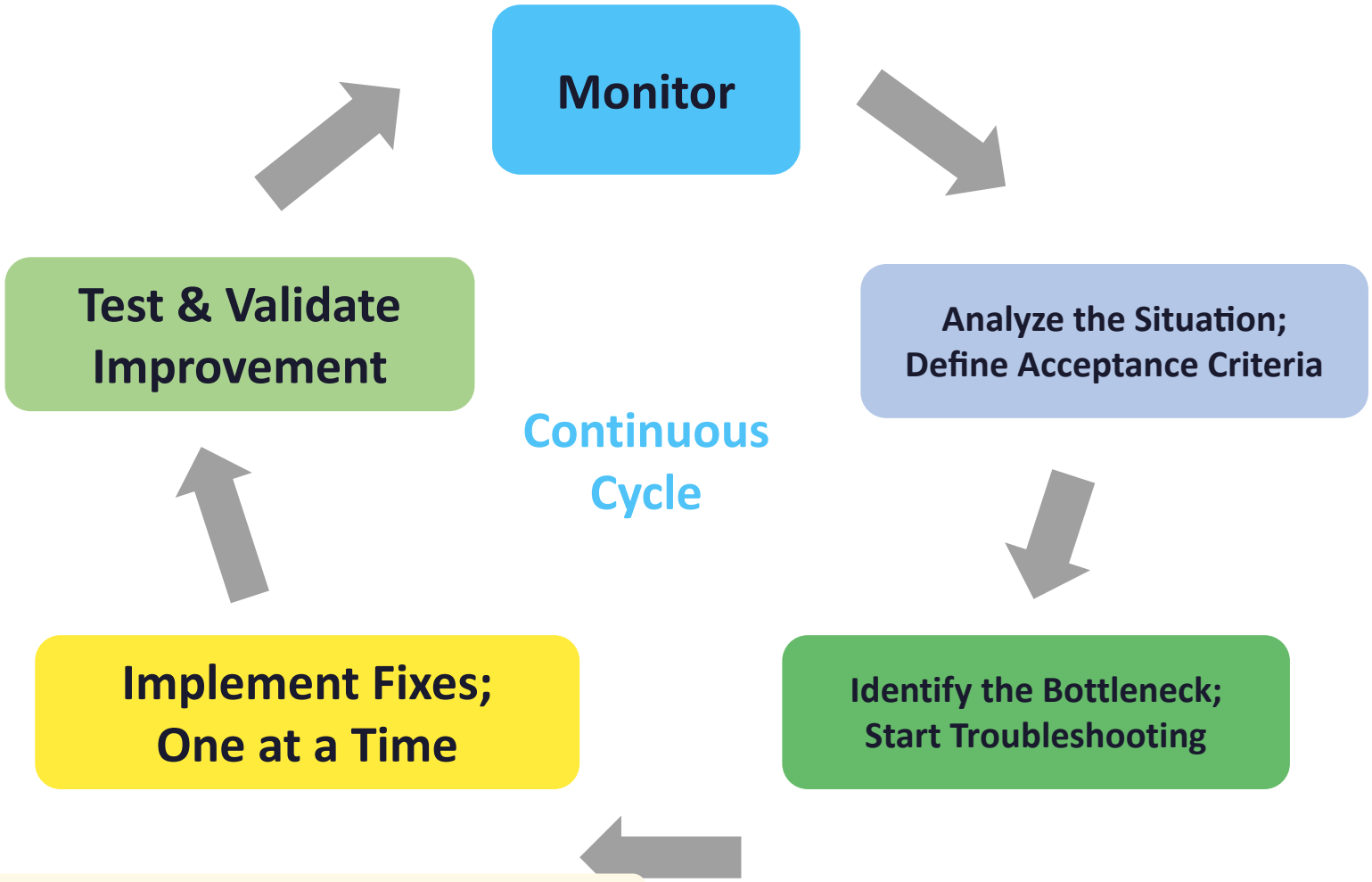
⚠️ Prepared Statements , Vacuum, TOAST, Sub transactions, Wait Events, Connection Management at Scale

Performance Tuning

Where do we start?

It depends... Proactive or Reactive Mode

Performance Tuning



“If you skip 'analyze', you're just guessing faster.”

01

Infrastructure & Resource Analysis

CPU, Memory, Storage & Access Patterns

Key Performance Factors — What to Measure First

Before diving into fixes, determine WHICH resource is the bottleneck

CPU

- Sub-optimal queries
- Undersized instance
- High connection:vCPU ratio
- Large parallel queries

Memory

- High # connections
- Memory Parameters
- Memory-intensive queries
- Logical replication
- Excessive vacuum memory

IOPS

- Bloated Table/Index
- Ineffective vacuum
- Inefficient checkpoints
- Insufficient throughput

Application

- Blocking/locking
- Long-running txns
- Idle-in-transaction
- Subtransaction overflow

CPU — Causes & Diagnosis

Common Causes

- Sub-optimal queries (missing indexes, seq scans)
- High connection-to-vCPU ratio
- Large parallel queries
- Configuration Parameters(random_page_cost, seq_page_cost)
- Undersized CPU for workload

Troubleshoot

1. pg_stat_statements(*exec_time), pg_stat_activity and top CPU Consuming OS Processes
2. Identify top CPU-consuming queries
3. EXPLAIN ANALYZE suspect queries
4. Check connection count vs vCPUs
5. Review parallel query settings

Must do : Connection pooler (PgBouncer, RDS Proxy) to reduce context switches

Monitoring: % Total CPU | % System CPU | % Wait CPU

Memory — Causes, Diagnosis & Fixes

Common Causes

- High # of connections consuming memory
- Sub-optimal memory-intensive queries
- Unmanaged logical replication slots
- Excessive vacuum memory (maintenance_work_mem)

Key Configuration

Parameter	Recommendation
shared_buffers	25% of OS memory
work_mem	Per operation (not per session!)
maintenance_work_mem	~5% of OS memory
effective_cache_size	50-70% of total mem

Monitoring Tools

```
-- Check memory contexts (v14+)
System view: pg_backend_memory_contexts;
System Function:
pg_log_backend_memory_contexts
-- Monitor: pg_proctab for OS-level
   memory per backend
```

Metrics: % Buffer Cache Hit Ratio | Freeable Memory

Memory Deep Dive — work_mem Is Per Operation!

⚠️ work_mem is allocated PER SORT/HASH OPERATION, not per session!
A single complex query with 5 hash joins could use 5x work_mem. With 100 connections, total memory = 100 × 5 × work_mem

Memory Budget Calculation

```
Total Memory = shared_buffers
             + (max_connections * work_mem * avg_ops_per_query)
             + maintenance_work_mem * autovacuum_max_workers
             + temp_buffers * max_connections
             + OS reserved (~1-2GB)+logical_Decoding_work_mem*replication_slots
```

Quick Wins

- Reduce idle connections (connection pooling)
- Tune work_mem conservatively, increase per-query

Storage & IOPS — Causes, Diagnosis & Fixes

Common Causes

- Sub-optimal queries causing excessive I/O
- Insufficient storage sizing (IOPs, throughput)
- Table and index bloat (dead tuples)
- Ineffective vacuum not reclaiming space
- Inefficient checkpoint configuration

Key Configuration

Parameter	Recommendation
max_wal_size	4 GB (Reduce ckpt frequency)
checkpoint_timeout	15 min (Increase ckpt intervals)
random_page_cost	1.0 (SSD Storage)

Quick Wins

Enable track_io_timing, Analyze High IOPS Query

- Monitor Sequential scans ⇒ Create required Indexes
- Monitor Bloated objects proactively, pg_repack or vacuum online
- Tune vacuum parameters
- Better Storage System – such as local NVME

Metrics: Read/Write Latency | IOPS | Disk Queue Depth |Throughput

Storage & IOPS — Causes, Diagnosis & Fixes

IOPS

Operations/sec

Key for OLTP

MB/s

Throughput

Key for DW/ETL

Latency

Response time

Key for all

Key Parameters

```
max_wal_size = 4GB    -- Reduce checkpoint
frequency
checkpoint_timeout = 15min  -- Longer
intervals
checkpoint_completion_target = 0.9
random_page_cost = 1.0      -- For SSD
storage
```

BLOAT: The Silent Performance Killer

MVCC → Dead Tuples → Increased Physical I/O → Reduced Buffer Efficiency

Fix: VACUUM (online) or pg_repack (no lock) to reclaim space

Key Metrics: Read/Write Latency | IOPS | Throughput | QueueDepth

02

Query Execution & EXPLAIN Plans

Understanding What PostgreSQL is doing

Query Patterns & EXPLAIN Plans — Reading the Evidence

EXPLAIN ANALYZE is your #1 diagnostic tool

Rows: Estimated vs Actual

Large mismatch ⇒ stale statistics

Planning vs Exec Time

Planning >> Execution = statistics issue

Seq Scan vs Index Scan, Nested Loop vs Hash Join

Seq Scan on large tables => Missing Foreign Key Index?

Sort Method

disk sort ⇒ increase work_mem

```
postgres=> EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM orders WHERE amount > 100;

 Seq Scan on orders
   (cost=0.00..18584.00 rows=52340 width=64)
   (actual time=1.205..892.410 rows=52340 loops=1)
   Filter: (amount > 100)
   Rows Removed by Filter: 947660
   Buffers: shared hit=1204 read=7380
 Planning Time: 0.078 ms
 Execution Time: 895.230 ms
```

Query Patterns & EXPLAIN Plans — Reading the Evidence

EXPLAIN ANALYZE is your #1 diagnostic tool

Rows: Estimated vs Actual

Large mismatch ⇒ stale statistics

Planning vs Exec Time

Planning >> Execution = statistics issue

Seq Scan vs Index Scan, Nested Loop vs Hash Join

Seq Scan on large tables => Missing Foreign Key Index?

Sort Method

disk sort ⇒ increase work_mem

```
EXPLAIN ANALYZE SELECT * FROM cars WHERE make = 'Honda' AND model = 'Civic';
Seq Scan on cars
  (cost=0.00..20406.00 rows=28 width=13)
  (actual time=1.068..86.353 rows=5000 loops=1)
  Filter: ((make)::text = 'Honda'::text)
    AND ((model)::text = 'Civic'::text))
  Rows Removed by Filter: 995000
Planning Time: 5.067 ms
Execution Time: 87.483 ms
```

03

Data-Distribution Statistics & Indexes

Helping the Planner Make Better Decisions

Statistics — The Optimizer's Blind Spot

PostgreSQL planner relies on statistics to derive execution plans. Without accurate stats, it guesses wrong.

Statistics Sample

- default_statistics_target controls data sample size
- Sample size = 300 × default_statistics_target (100) = 30,000
- default_statistics_target : 100(default), 10,000(max)

Key Metrics Collected: in pg_statistics catalog

- # of distinct values (n_distinct)
- Most common values (MCV)
- Histogram bounds
- NULL percentage
- Correlation (Multivariate Statistics)
- Univariate Statistics - Expression

Change Defaults only when Data is skewed , go granular….

Case Study: Planning Time 598ms → 0.2ms

The Problem

- CPU Spiked up / Queries regressed 90% after a configuration change
- default_statistics_target changed from 300 to 8000 at DB level
- Planning time became 400x greater than execution time!

BEFORE (DB-level default_statistics_target=8000)

```
Planning Time:  598.000 ms  ❌
Execution Time:   1.500 ms

Planning = 400x Execution!
```

AFTER (column-level statistics)

```
Planning Time:    0.203 ms  ✅
Execution Time:   1.500 ms

2,946x faster planning!
```

Fix & Best Practices

```
Set statistics at COLUMN level, not database level

ALTER TABLE my_table ALTER COLUMN my_col SET STATISTICS 256;
ANALYZE my_table;  -- Recommended default_statistics_target = 256
```

Indexes — Missing, Redundant & Ineffective

Indexes enhance reads but cost writes — always test the net impact!

PostgreSQL Index Types

Type	Best For
B-tree	Default. Equality & range queries. Most common.
Hash	Equality only. Faster for = comparisons.
BRIN	Block Range Index. Large tables with natural ordering.
GIN	Generalized Inverted. Full-text, JSONB, arrays.
GiST	Generalized Search Tree. Geometric, range types.
SP-GiST	Space-Partitioned GiST. Non-balanced data.
Bloom	Multi-column equality. Signature file index.

Find Missing Indexes

Tables with high sequential scans

```
SELECT relname, seq_scan,
idx_scan,
  seq_scan - idx_scan AS
too_many_seqs
FROM pg_stat_user_tables
WHERE seq_scan > idx_scan
ORDER BY too_many_seqs DESC
LIMIT 10;
```

Find Unused/Redundant Indexes

```
SELECT indexrelname, idx_scan
FROM pg_stat_user_indexes
WHERE idx_scan = 0
ORDER BY
pg_relation_size(indexrelid)
DESC;
```

Also consider partial indexes, expression-based indexes, compound indexes

LAST RESORT : FORCE AN INDEX =>>> SET enable_seqscan = OFF;

Multicolumn Index Limitation — The "Leading Column" Problem

Why your index exists but PostgreSQL won't use it

```
-- Index: CREATE INDEX idx_loc ON users (country, city);

-- ✅ Uses index (leading column present)

SELECT * FROM users WHERE country = 'USA' AND city = 'NYC';
SELECT * FROM users WHERE country = 'USA';

-- ❌ Seq Scan! (leading column missing, index useless)
SELECT * FROM users WHERE city = 'NYC';
```

The Problem

- Index on (A, B, C) only works when query filters on A first
- Query by B or C alone → full table scan, index ignored
- Teams create redundant single-column indexes to compensate
- Each extra index slows writes, consumes storage, adds vacuum overhead

PG ≤ 17: Create separate indexes or use partial indexes

Impact

- Write throughput drops 20-30%
- Storage doubles from overlapping indexes
- Vacuum takes 2-3x longer
- Example: 8 indexes on one table

```
PG 18 FIX: SKIP SCAN
-- PG 17: Seq Scan: 10M rows, Time:
2,300ms
-- PG 18: Skip Scan: Time: 100ms →
23x faster!
```

Application Access Patterns — The Hidden Bottleneck

Long-Running Transactions

Block vacuum, hold locks, consume resources. Set statement_timeout, idle_in_transaction_session_timeout.

Idle-in-Transaction

Connections holding open transactions without doing work. Block vacuum and replication.

Idle Connections

Each connection uses ~5-10MB. 1000 idle connections = 5-10GB wasted memory.

Excessive Foreign Keys

Each FK check requires index lookup on referenced table. Batch operations amplify cost.

Missing Retries/Backoff

Application hammers DB on failure instead of exponential backoff. Cascading failures.

Best Practice: explicit BEGIN/COMMIT, timeouts, retries with exponential backoff

04

From the Field: PostgreSQL Performance Case Studies

Impact of Idle Connections

THE PROBLEM

- PostgreSQL production database is low on memory
- Low volume, read-only workload running on the system
- Key question: What is consuming the memory?

SYMPTOMS

< 5%

Available Memory

Review Metrics for Connection Memory Usage

1

MemAvailable

Tracks available memory on the DB instance

2

Database Connections

Number of active connections to the database

3

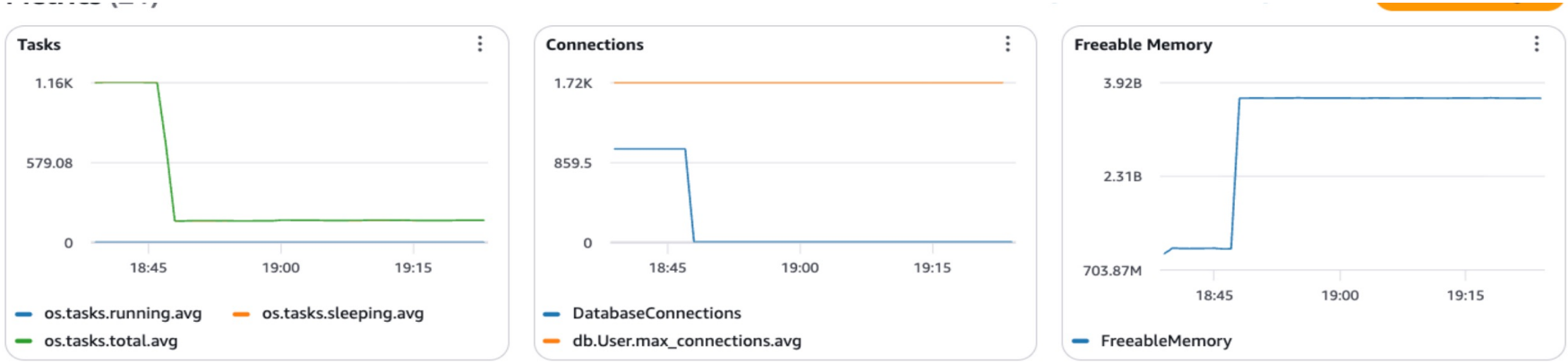
OS tasks – Running, Sleeping, Total

Monitor for correlation with connection spikes

4

CloudWatch Alarms

Detect when Freeable Memory drops below threshold



💡 RDS/Aurora PostgreSQL - Set CloudWatch alarms on FreeableMemory and DatabaseConnections metrics to proactively detect memory pressure before it impacts performance.

Root Cause & Solution

ROOT CAUSE

⚠️

Idle Connections Consuming Memory

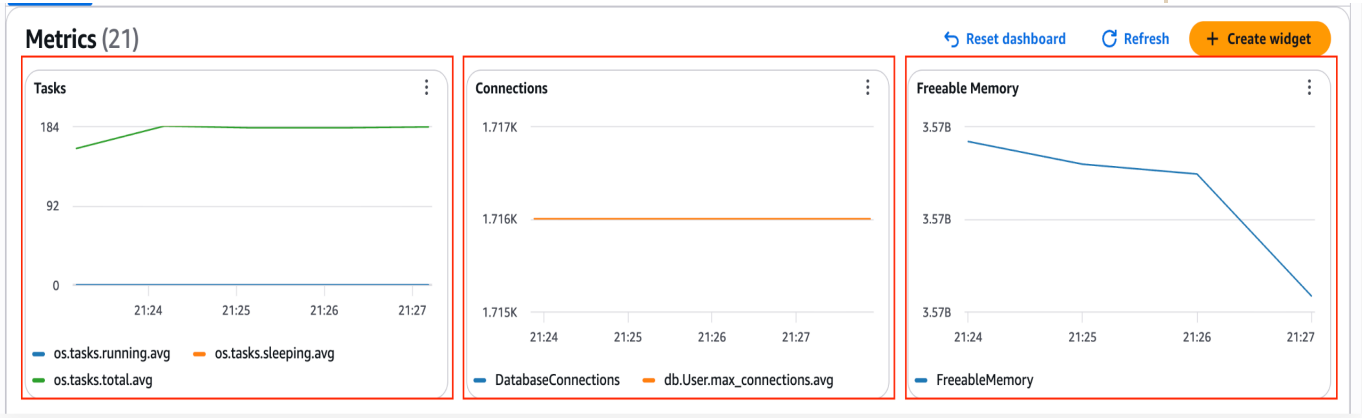
Each idle connection holds certain memory on the database instance, even without active queries.

SOLUTION

Use server side connection pooler

1. **PgBouncer**
   Lightweight connection pooler
2. **PgPool II**
   Advanced pooling + load balancing
3. **RDS Proxy**
   Fully managed AWS proxy service

After Using RDS Proxy



💡 Client-side pooler can complement server-side pooler to reduce network roundtrips

Prepared Statements — When Planning Time >> Execution Time

Understanding the Parse → Plan → Execute Pipeline

UNPREPARED (Every Execution)

Repeated every time → expensive for simple queries

PREPARED (Cached Plan)

Parse once, execute many → significant savings

plan_cache_mode options

Mode	Behavior
`auto`	Default. PG decides custom for first 5 executions, then generic if cost is similar.
`force_generic_plan`	Always use cached generic plan. Best when all queries have similar selectivity.
`force_custom_plan`	Always re-plan with actual parameter values. Best for highly skewed data.

⚠ **The auto mode can cause sudden performance changes after the 5th execution when PostgreSQL switches from custom to generic plans.**

Partition Lock Contention with Prepared Statements

LWLock:lock_manager — The Problem & Mechanism

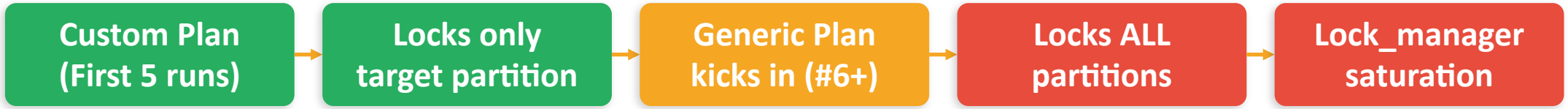
THE PROBLEM

Generic plans lock **ALL partitions** at plan time, even though runtime pruning scans only **1 partition**

SYMPTOMS

- pg_stat_activity shows LWLock:lock_manager waits
- CPU spikes but execution time is low
- Latency increases non-linearly with connections
- Problem appears suddenly after 5th execution

THE MECHANISM



WHY THIS CAUSES CONTENTION

16
LWLock partitions

200
Connections

1,000
Partitions

200K
Lock Acquisitions

SAMPLE QUERY

```
PREPARE q1 AS SELECT * FROM events WHERE event_date = $1;
EXECUTE q1('2025-01-15');  -- Custom plan: locks 1 partition
EXECUTE q1('2025-01-15');  -- Runs 5 times...
EXECUTE q1('2025-01-15');  -- 6th run: Generic plan → locks ALL 1000 partitions!
```

Partition Lock Contention: Solutions

SOLUTIONS

1	**Force custom plans**	`SET plan_cache_mode = 'force_custom_plan';`
2	**Reduce partition count**	Monthly instead of daily partitions → 12x fewer locks
3	**Aggressive partition pruning**	DROP old partitions regularly to reduce lock surface
4	**Increase max_locks_per_transaction**	Symptom relief only — does not fix root cause
5	**Partition-aware query design**	Query child partitions directly — bypasses parent table and avoids locking all partitions
6	**Aurora PostgreSQL shared plan cache**	Reduces memory consumption by keeping one copy of generic plan across all sessions

KEY INSIGHT

Plan-time pruning reduces I/O AND locks.

Runtime pruning reduces I/O but NOT locks.

This is why generic plans are dangerous with many partitions — they bypass plan-time pruning, so every partition gets locked even when only one is read.

Alert !! Most basic yet most common culprit for Performance

…

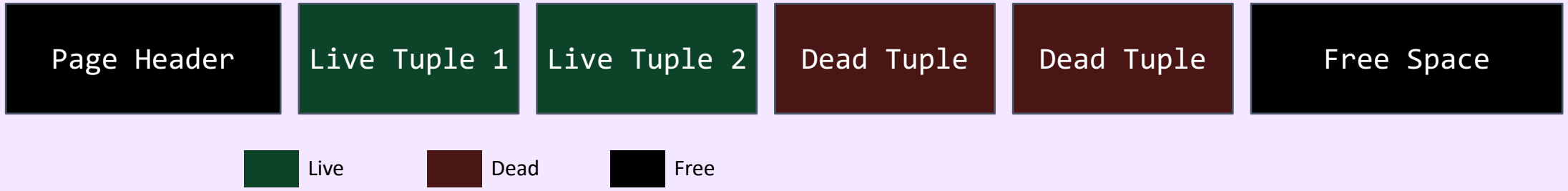
VACCUM

….only if not tamed well.

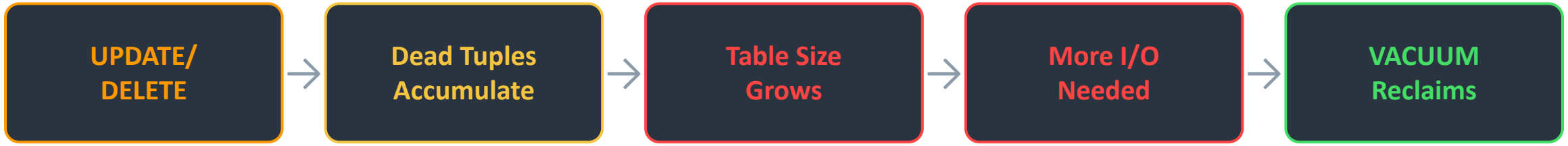
MVCC, Dead Tuples & The Bloat Problem

How UPDATE/DELETE operations create hidden performance debt

PostgreSQL Page Structure (8KB)



Impact Chain:




```
-- Check table bloat
SELECT schemaname, relname, n_dead_tup, n_live_tup,
       round(n_dead_tup::numeric / GREATEST(n_live_tup, 1) * 100, 1) AS dead_pct
FROM pg_stat_user_tables ORDER BY n_dead_tup DESC LIMIT 10;
```

Vacuum & Autovacuum — Keeping Your Database Healthy

Reclaim Dead Tuples | **Update Statistics** | **Update Visibility Map** | **Prevent TX ID Wraparound**

BEST PRACTICES

- Increase autovacuum_max_workers (default 3)
- Increase autovacuum_vacuum_cost_limit
- Set table-level params for large tables
- Larger maintenance_work_mem for big tables
- NEVER disable autovacuum!

REMOVE BLOCKERS

- Idle in transactions
- Long-running transactions on replicas
- hot_standby_feedback on replicas
- Abandoned replication slots

AUTOVACUUM THRESHOLD FORMULA

```
threshold = autovacuum_vacuum_threshold
+ autovacuum_vacuum_scale_factor ×
pg_class.reltuples
```

Defaults: threshold=50, scale_factor=20%

For 10M row table: 50 + 0.20 × 10,000,000 = 2,000,050 dead tuples!

PG 18: AUTOVACUUM ENHANCEMENTS

Hard cap on dead tuples

autovacuum_vacuum_max_threshold — Default: 100M. Fires immediately regardless of scale factor.

Spread freeze work

vacuum_max_eager_freeze_failure_rate — Smooth, incremental freezing during normal ops.

Dynamic scaling

autovacuum_max_workers reloadable — pg_reload_conf() to add workers on the fly.

High ReadIOPS from Bloated Tables

Root Cause Chain + Detection & Solutions

SCENARIO

ReadIOPS spiked 10x with no workload change. Long-running query on reader instance with hot_standby_feedback=on prevented vacuum from removing dead tuples on the writer.

ROOT CAUSE CHAIN

Long query on reader ▶ **hot_standby _feedback=on** ▶ **Vacuum blocked** ▶ **Dead tuples accumulate** ▶ **ReadIOPS spikes 10x**

SOLUTIONS

1	**Table-level autovacuum**	autovacuum_vacuum_scale_factor = 0.01 (1% instead of 20%)
2	**Avoid long running transactions**	Set statement_timeout and idle_in_transaction_session_timeout on reader instances
3	**VACUUM FULL / pg_repack**	Reclaim bloat once accumulated (requires downtime or pg_repack)
4	**Manage hot_standby_feedback**	Set max_standby_streaming_delay to a bounded value (e.g., 30s)

```
-- Table-level autovacuum for large/hot tables
ALTER TABLE orders SET (
  autovacuum_vacuum_scale_factor = 0.01,    -- 1% instead of 20%
  autovacuum_vacuum_threshold = 1000,
  autovacuum_analyze_scale_factor = 0.005
);
```

TOAST Table Bloat — The Invisible Performance Killer

The Problem & Why Standard Monitoring Misses It

WHAT IS TOAST?

TOAST stands for The Oversized-Attribute Storage Technique. PostgreSQL pages are 8KB. When a row has columns that are too large to fit in a single page — think jsonb documents, large text fields, bytea blobs — PostgreSQL automatically moves those values out of the main heap table and into a separate, hidden TOAST table.

THE PROBLEM

- Columns >2KB (jsonb, text, bytea) are stored in a hidden TOAST table
- Updates create dead tuples that are invisible to pg_stat_user_tables
- n_dead_tup only shows main heap dead tuples

WHY IT'S INVISIBLE

- pg_stat_user_tables.n_dead_tup = 0 (looks healthy!)
- Standard bloat monitoring queries miss it entirely
- Storage metrics don't separate heap vs TOAST
- VACUUM targets main table, ignores TOAST bloat

"50 GB table"
might actually be
50 GB heap + 200 GB TOAST

SYMPTOMS

⚠ Storage growing unexpectedly

⚠ Read/write latency increasing

⚠ Backup times ballooning

⚠ VACUUM running longer

✓ **But pg_stat_user_tables looks FINE**

Standard bloat monitoring queries check pg_stat_user_tables — they have no visibility into the TOAST relation.

TOAST Table Bloat — The Invisible Performance Killer

How to Find and Fix Hidden TOAST Bloat

SOLUTIONS

1	**Monitor TOAST separately**	Query pg_class for TOAST relation sizes (standard monitoring misses it)
2	**VACUUM FULL / pg_repack**	Only way to reclaim TOAST bloat (regular VACUUM won't shrink it)
3	**Reduce TOAST column updates**	Update only changed columns, avoid full-row rewrites in ORMs
4	**Column storage strategy**	Use EXTERNAL storage for columns that don't compress well, or split large columns into separate tables

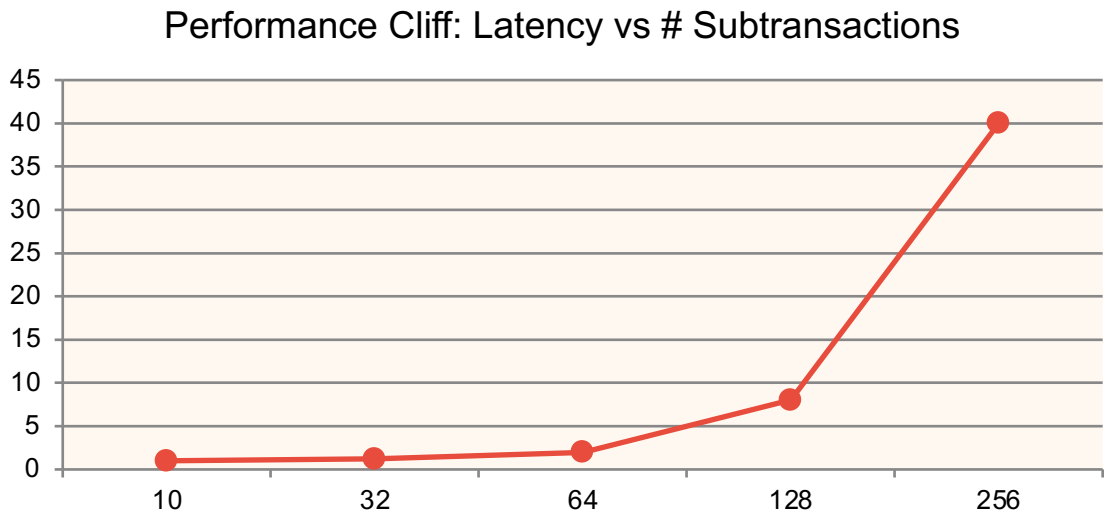
DETECTION QUERY

```
-- Reveal hidden TOAST sizes
SELECT c.relname AS table_name,
       pg_size_pretty(pg_relation_size(c.oid)) AS heap_size,
       pg_size_pretty(pg_relation_size(c.reltoastrelid)) AS toast_size
FROM pg_class c
WHERE c.reltoastrelid > 0
AND c.relkind = 'r'
ORDER BY pg_relation_size(c.reltoastrelid) DESC
LIMIT 10;
```

LWLock:SubstransSLRU Subtransactions

Symptoms

- Database is experiencing sudden performance degradation
- High CPU but low actual query execution time.
- Increased application timeouts
- pg_stat_activity shows **LWLock:SubtransSLRU** as the dominant wait event.



What is SLRU and Why It Matters

Simple Least Recently Used — PostgreSQL buffers for transaction related metadata

SLRU Caches

`pg_subtrans`
Subtransaction tracking

`pg_multixact`
Multixact member lookups

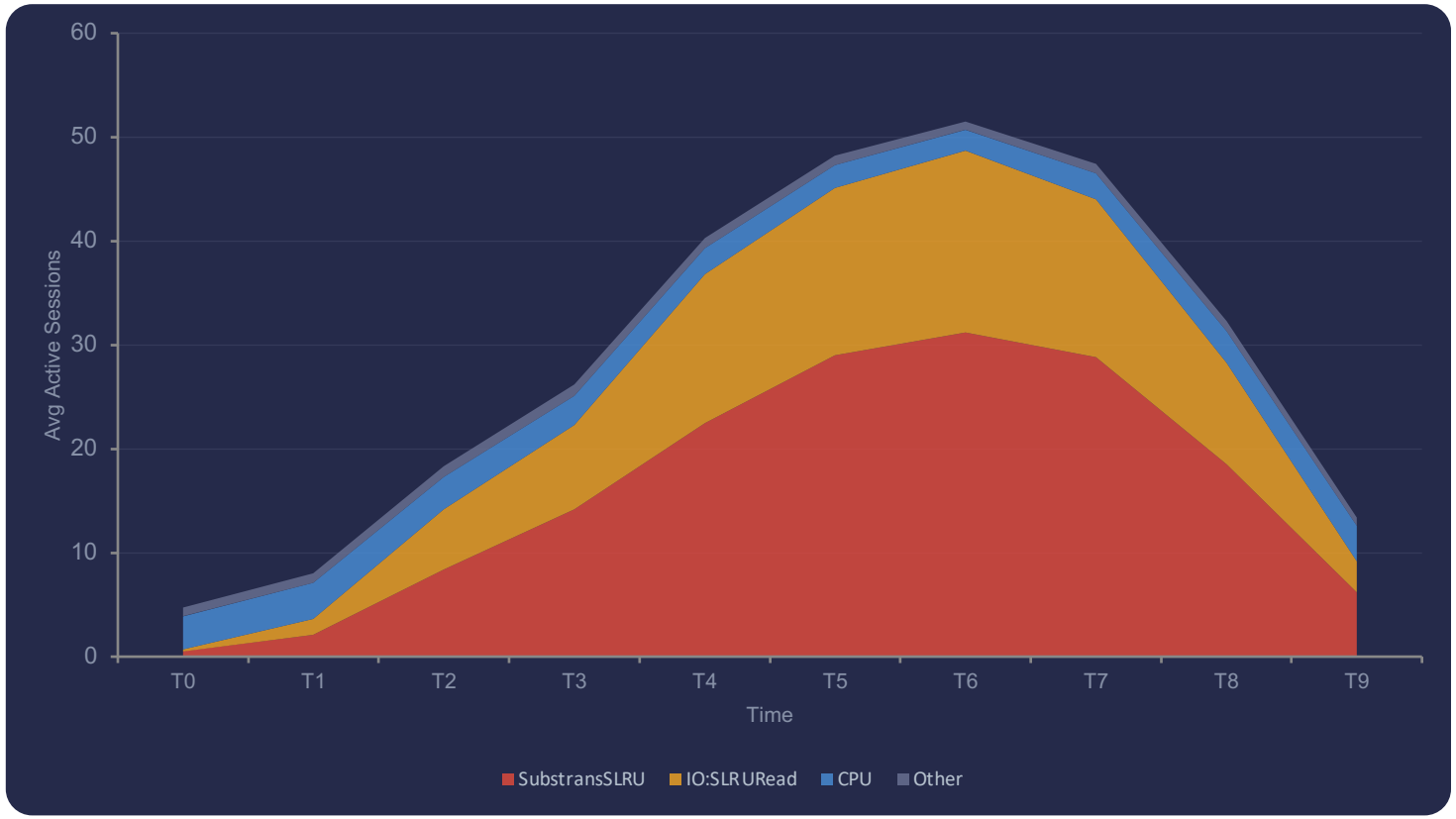
`pg_commit_ts`
Commit timestamps

Subtransactions > 64

- Overflow → pg_subtrans on disk via SLRU
- Only 32 shared SLRU buffers (pre-PG 17)
- LWLock contention under concurrency

Real-World Impact — Wait Events & DB Load

LWLock:SubtransSLRU dominates Average Active Sessions under subtransaction overflow



58%
LWLock:SubtransSLRU

32%
IO:SLRURead

Case Study: JDBC autosave=always

- Creates a SAVEPOINT before every statement
- Each SAVEPOINT = 1 subtransaction
- → Rapid overflow past 64
- Amplified on hot standby replicas
- Rapid XID growth → wraparound risk

Before PG 17: No knob to turn — you could only avoid subtransactions entirely.

GitLab famously had to ban subtransactions across their entire codebase.

PG 17 — The Fix Is Finally Here

Two landmark improvements that soften the subtransaction performance cliff

Configurable SLRU Cache Sizes

New server variables:

- `subtransaction_buffers`
- `multixact_member_buffers`
- `multixact_offset_buffers`
- `commit_timestamp_buffers`

Per-Bank SLRU Locking

Before: Single lock per entire SLRU cache

After: Cache divided into banks, each with independent locks

Dramatically reduces `LWLock:SubtransSLRU` and `LWLock:MultiXactSLRU` contention under high concurrency

Best Practices — All Versions

DETECT

Monitor wait events in `pg_stat_activity`

Track CloudWatch metric `MaximumUsedTransactionIDs`

FIX

JDBC: `autosave=conservative`
Review ORM subtxn settings
Remove unnecessary SAVEPOINTs
PG 17+: Tune `subtransaction_buffers`

AVOID

Subtransactions if possible

EXCEPTION blocks in PL/pgSQL

Long-running txns + subtxns *(especially on replicas)*

💡 **PG 17 softens the cliff, but avoiding unnecessary subtransactions remains the best strategy.**

PostgreSQL Tuning Options — The Big Picture

Object Tuning

- Table/index bloat reduction
- Partitioning strategy
- Index optimization (add/remove)
- FillFactor adjustment
- Data type selection

Query Tuning

- Query rewrites
- Analyze Data : Skewed ? Statistics (column-level)
- Plan hints (pg_hint_plan)
- Predicate pushdown
- Prepared statements

Maintenance Tuning

- Autovacuum optimization
- Bloat monitoring
- REINDEX/pg_repack
- Checkpoint tuning
- WAL management

Parameter Tuning

- Memory (shared_buffers, work_mem)
- Parallelism settings
- Optimizer planner costs
- Connection management
- I/O configuration

Thank you!

Speakers

Anita Singh

Principal Database Specialist SA, AWS

Ranjan Burman

Senior Database Specialist SA, AWS

Q & A

We'd love to hear your PostgreSQL war stories!

Your feedback helps us improve, please complete the quick survey.

TroubleShoot Amazon Aurora PostgreSQL Performance by Use Case