Postgres at Scale:

Lessons from Running Multi-Terabyte Clusters

PGConf India 2026

Roneel Kumar

Senior Database Specialist Solutions Architect

Sameer Kumar

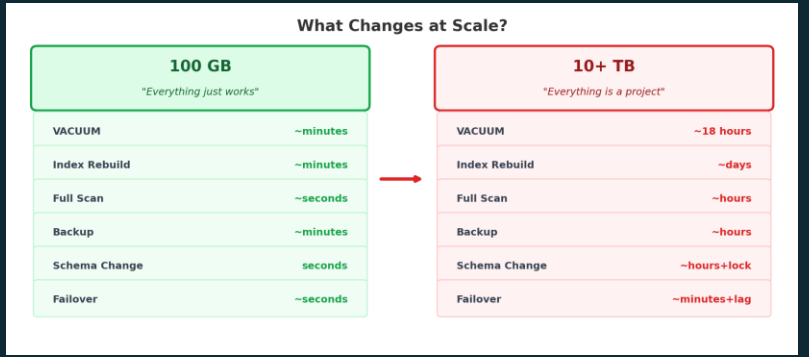
Principal Database Specialist

What We'll Cover Today

- Why scaling Postgres is different at TB scale
- Storage architecture and partitioning strategies
- Autovacuum and bloat management
- Replication and high availability challenges
- Schema evolution on massive tables
- Monitoring and observability at scale

From GB to TB: What Changes?

- Maintenance windows disappear (hours → days)
- Full table scans become prohibitively expensive
- Index rebuilds can take days, not minutes
- Backup and restore strategies need rethinking
- Single-node limitations hit hard
- OLTP vs Analytics: different scaling patterns



Storage, Bloat, and VACUUM

What if Postgres Had NO VACUUM? 🔥

VACUUM is not optional. It's survival.

Day 1: Everything looks great! 🚀

Day 7: Hmm, queries getting slower... 🤔

Day 30: Table size doubled, performance halved 📈

Day 90: Dead tuples everywhere, bloat at 80% 💀

Day 180: Transaction ID wraparound warning! ⚠️

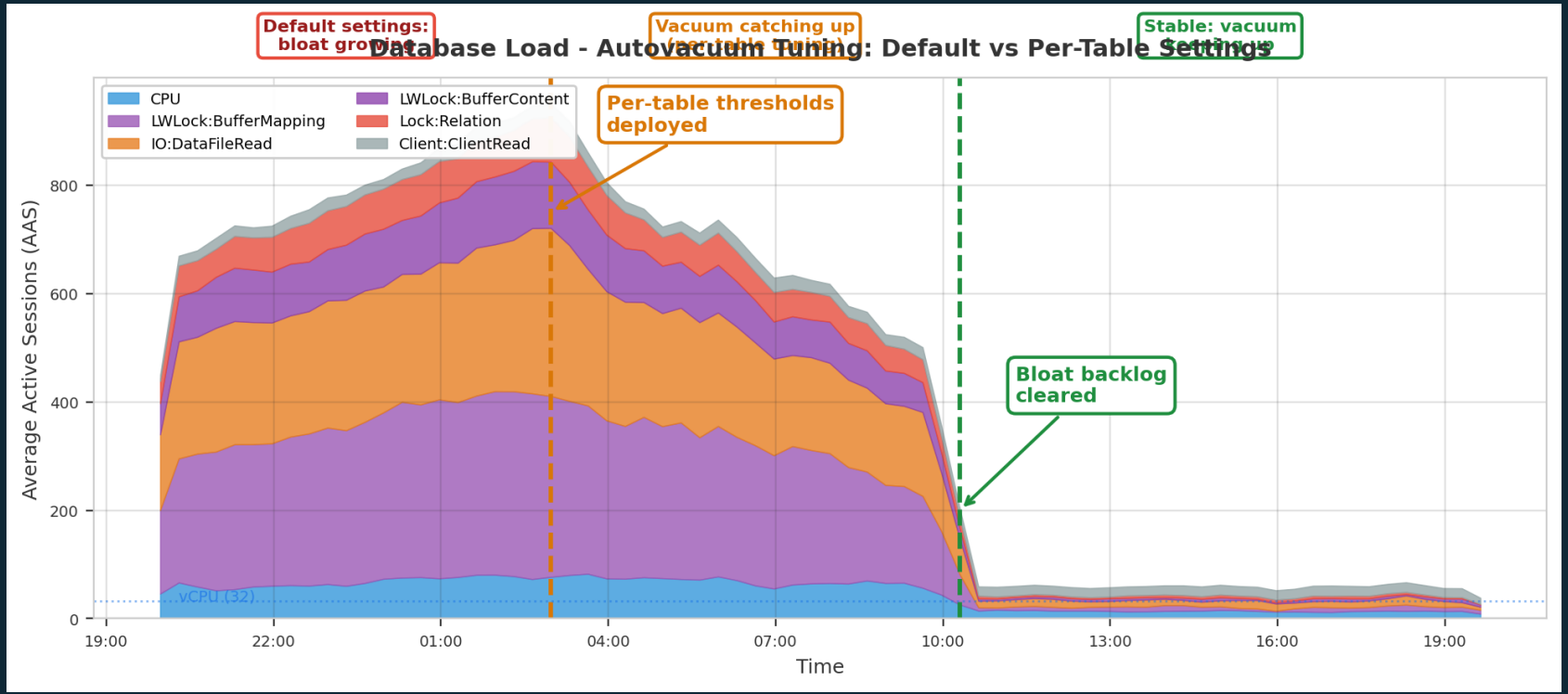
Day 365: Database size = 10x actual data 🎈

The End: Database is officially a snail 🐌

Autovacuum at Scale

- Default settings fail at multi-TB scale
- Vacuum can't keep up with bloat
- Tuning strategy:
  - Increase autovacuum_max_workers (4-8)
  - Lower autovacuum_vacuum_scale_factor (0.01-0.05)
  - Raise autovacuum_vacuum_cost_limit (2000-10000)
- Per-table settings for hot tables
- Monitor vacuum progress continuously

Autovacuum at Scale



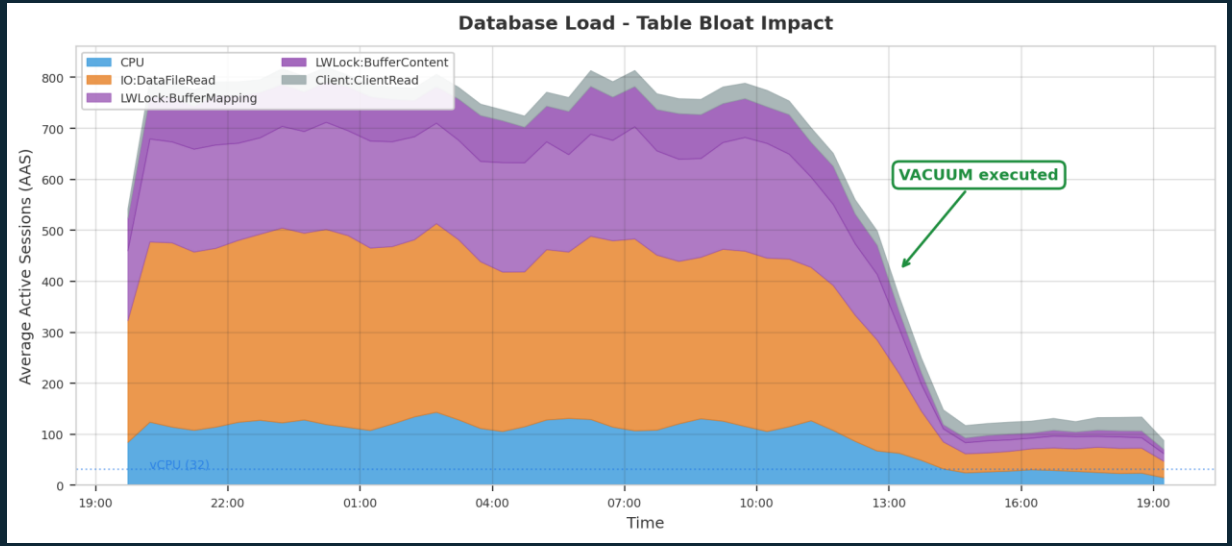
Vacuum Strategies for Multi-TB Tables

📞 3 AM: "Database is slow, queries timing out"

🔍 Investigation: 8TB table, 40% bloat, autovacuum running for hours

```
postgres=> SELECT pid, now() - pg_stat_activity.query_start AS duration, query, state
FROM pg_stat_activity
WHERE state != 'idle'
ORDER BY duration DESC;
-[ RECORD 1 ]-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
pid      | 19996
duration | 03:19:36.692611
query    | autovacuum: VACUUM public.orders
state    | active
-[ RECORD 2 ]-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
pid      | 14195
duration | 02:43:16.119127
query    | autovacuum: VACUUM public.item_inventory (to prevent wraparound)
state    | active
-[ RECORD 3 ]-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
pid      | 10802
duration | 00:39:24.604812
query    | select count(*) from item_inventory;
state    | active
-[ RECORD 4 ]-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

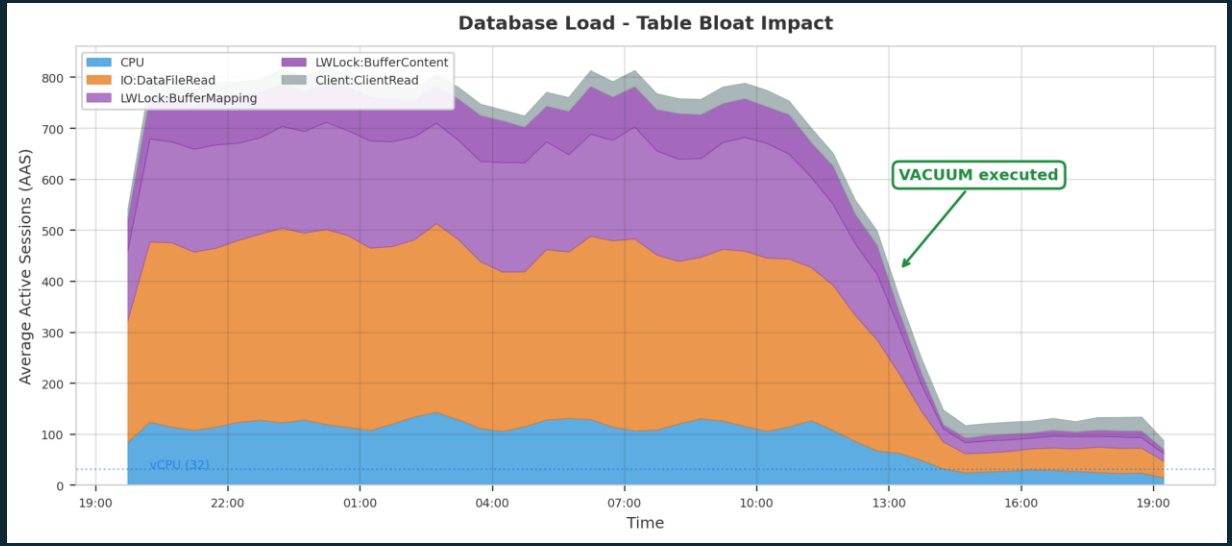
Vacuum Strategies for Multi-TB Tables



💡 **What We Learned:**

Partition the monster → vacuum 100GB chunks, not 8TB
Throttle during business hours → vacuum_cost_delay = 10ms
Schedule aggressive vacuum at 2 AM → cost_delay = 0
Watch freeze age → alert at a low level 200M and raise to critical at 1.5B transactions
Emergency bloat? → pg_repack (but test first!)

Vacuum Strategies for Multi-TB Tables

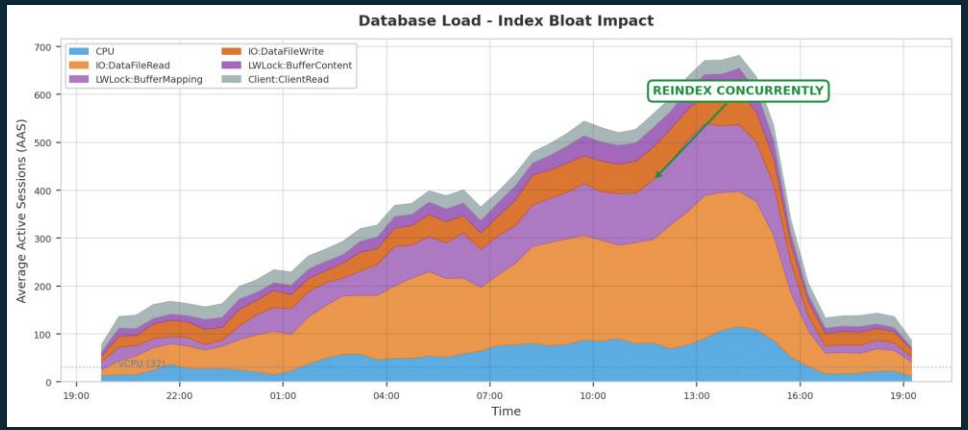


💡 **What We Learned:**

Partition the monster → vacuum 100GB chunks, not 8TB
Throttle during business hours → vacuum_cost_delay = 10ms
Schedule aggressive vacuum at 2 AM → cost_delay = 0
Watch freeze age → alert at a low level 200M and raise to critical at 1.5B transactions
Emergency bloat? → pg_repack (but test first!)

Detecting & Preventing Bloat

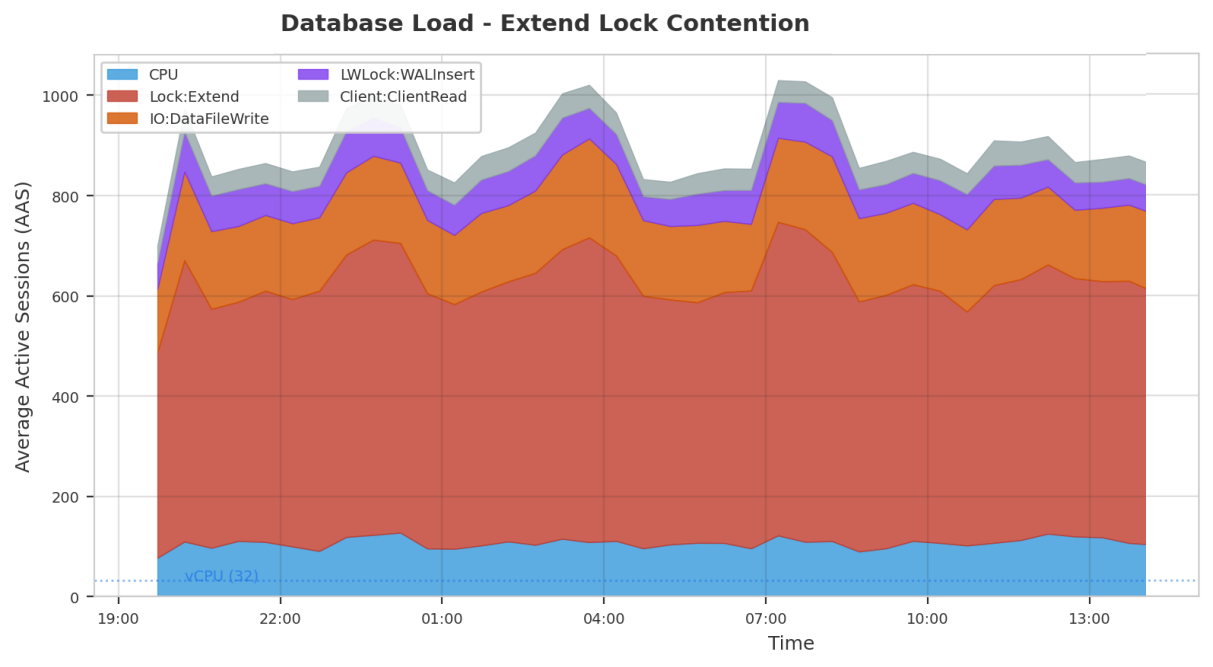
- Index bloat grows faster than table bloat
- Use pg_stat_user_tables to track dead tuples
- Query bloat with pgstattuple extension
- Prevention strategies:
    - Aggressive autovacuum on hot tables
    - HOT updates (avoid indexed column updates)
    - Fillfactor tuning (90 for frequent updates)
- REINDEX CONCURRENTLY for bloated indexes



Case Study: The Extend Lock 🔒

Problem: High-volume orders table, 50K inserts/sec

Symptom: Queries hanging on 'extend' wait events



Case Study: The Extend Lock 🔒

Investigation:

```
SELECT wait_event, count(*) FROM pg_stat_activity
WHERE state = 'active'
GROUP BY wait_event;
```

```
wait_event | count
extend     | 847    ← 🔥 Bottleneck!
```

Root Cause: fillfactor = 65 (for HOT updates)

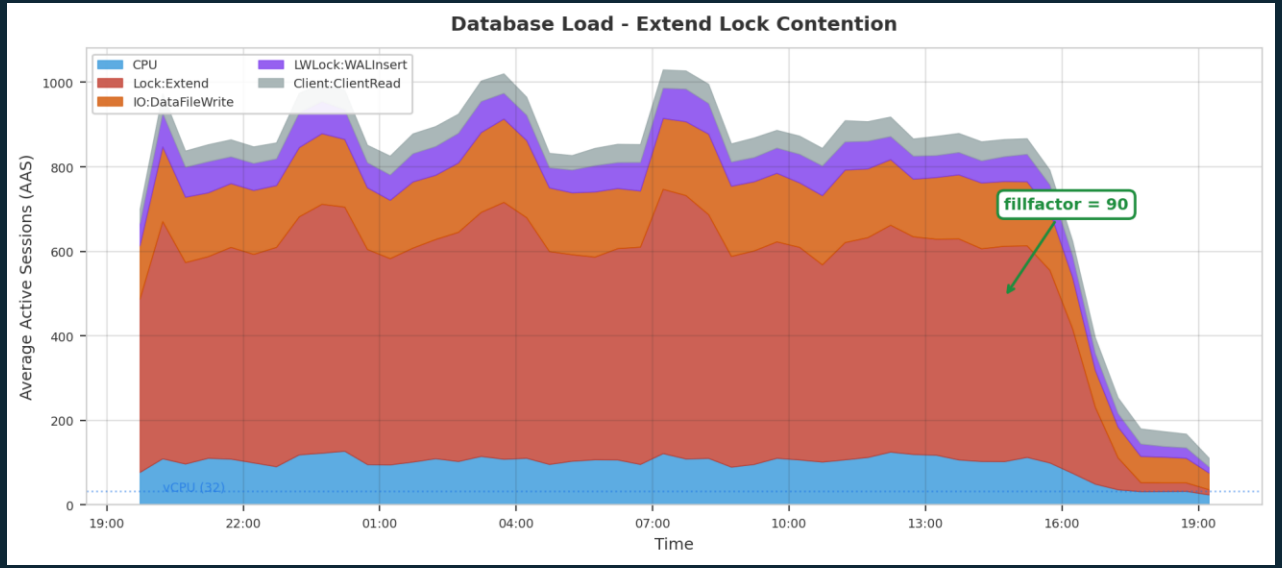
Pages only 65% full → Postgres constantly extending table → Serialized on extend lock

Case Study: The Extend Lock 🔒

Fix:

```
ALTER TABLE orders
SET (fillfactor =
90);

-- New pages use
fillfactor=90, no
rewrite needed!
(pg_repack if old
pages should be
updated)
```



Result: Extend waits dropped 95%, throughput increased 3x

Partitioning

Storage & Architecture Patterns

Partitioning - Is it optional ??

- Choose the Partitioning strategy based on access patterns:
  - Range: Time-series, chronological data
  - Hash: Even distribution, no hot partitions
  - List: Categorical data, tenant isolation
- Partition pruning saves query time
- Maintenance operations become partition-scoped

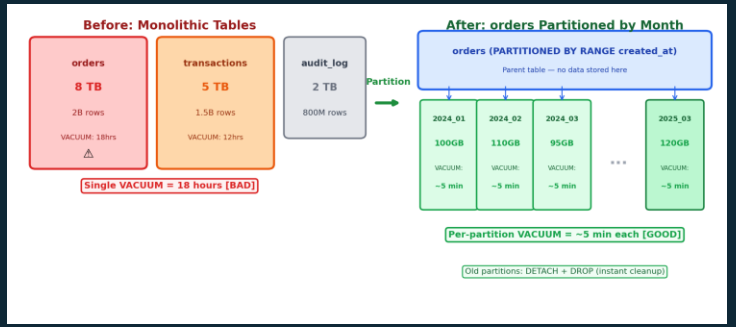
Keep Partition Count Reasonable

Too few partitions: Defeats the purpose

- Each partition still too large for maintenance
- Limited parallelism benefits

Too many partitions: Planning overhead kills performance

- Query planning time grows linearly with partition count
- 10,000+ partitions = seconds just to plan queries



Sweet spot: 100-1000 partitions for most workloads

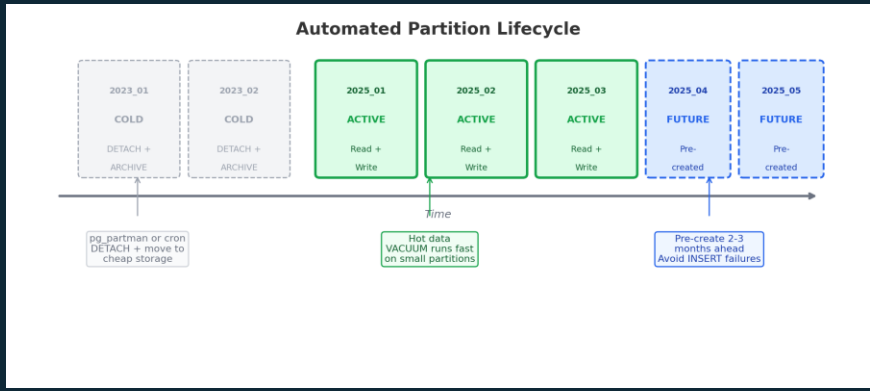
Example: 10TB table → 500 partitions = 20GB each

Automate Partition Creation/Archival

Manual partition management does it scale?

Creation automation:

- pg_partman extension (recommended)
- Cron job with CREATE TABLE ... PARTITION OF
- Create 3-6 months ahead of current date



Archival strategies:

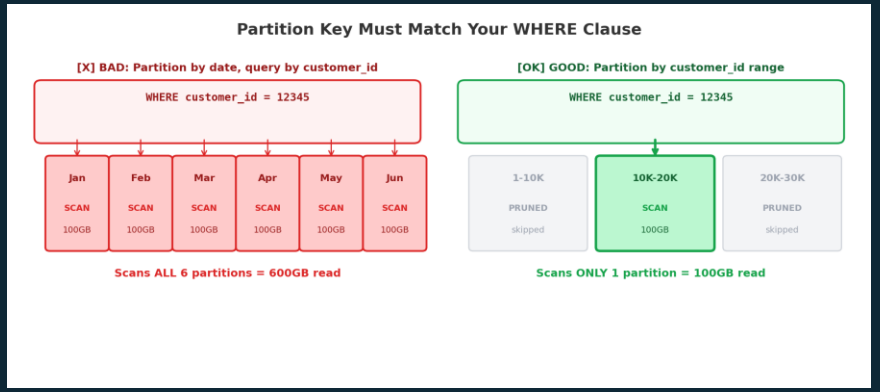
- DETACH PARTITION (PG 14+, non-blocking)
- Move to archive tablespace or external storage
- DROP old partitions after backup verification

Monitor Partition Sizes & Growth

- Uneven partition sizes indicate problems
- Query: pg_relation_size() per partition
- Watch for:
- Hot partitions (current month in time-series)
    - Skewed hash distribution
    - Unexpected growth rates
- Set alerts: partition > 150% of expected size
- Track partition count growth over time
- Rebalance if hash partitions become skewed

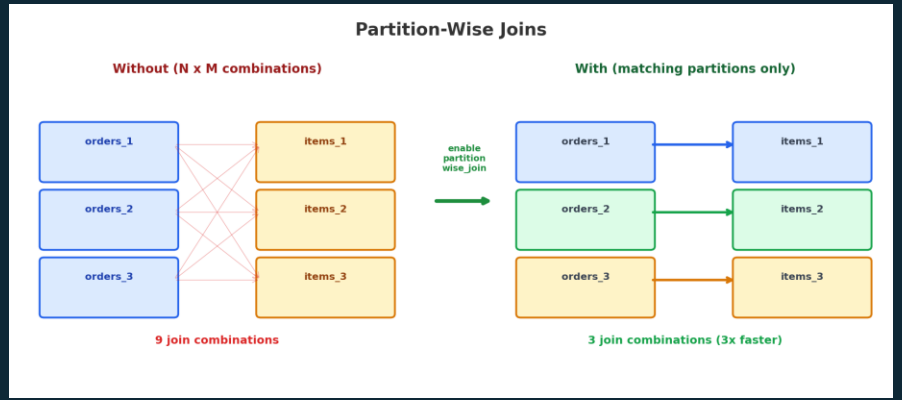
Align Partitions with Query Patterns

- Partition key MUST match your WHERE clauses
- Bad: Partition by date, query by customer_id
  - Result: All partitions scanned every query
- Good: Partition by date, query by date ranges
  - Result: Only relevant partitions touched
- Analyze your top 10 queries first
  - Consider composite partitioning for complex patterns
  - Example: Range by month + Hash by tenant_id



Partition-Wise Joins

- Joins between partitioned tables on partition key
- Without: N×M partition combinations checked
- With: Only matching partitions joined
- Enable: enable_partitionwise_join = on
- Requirements:
    - Both tables partitioned identically
    - Join condition includes partition key
- Can reduce join time from hours to minutes
- Monitor with EXPLAIN (Parallel Append nodes)



Workload and Query Tuning

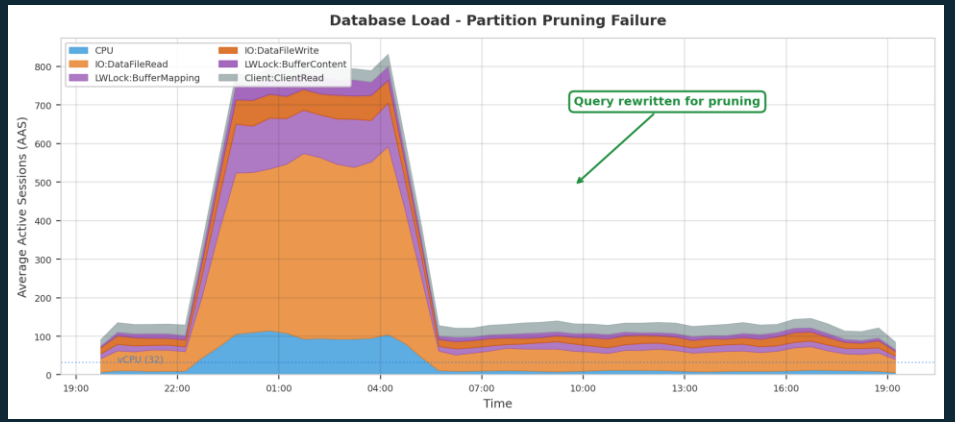
Case Study: Partition Pruning Failure

- E-commerce platform: 15TB orders table
- Partitioned by order_date (monthly, 180 partitions)
- Problem: Queries scanning all 180 partitions
  - Query time: 45 seconds
  - Expected: <1 second (single partition)
- Impact: Dashboard timeouts, customer complaints
- Root cause: Partition pruning not working
- Investigation revealed query pattern issues

Why Pruning Failed

Functions on partition key prevent pruning:

- EXTRACT(year FROM order_date)
- DATE_TRUNC('month', order_date)
- order_date::date (type cast)
- COALESCE(order_date, NOW())



Postgres can't map function results to partitions

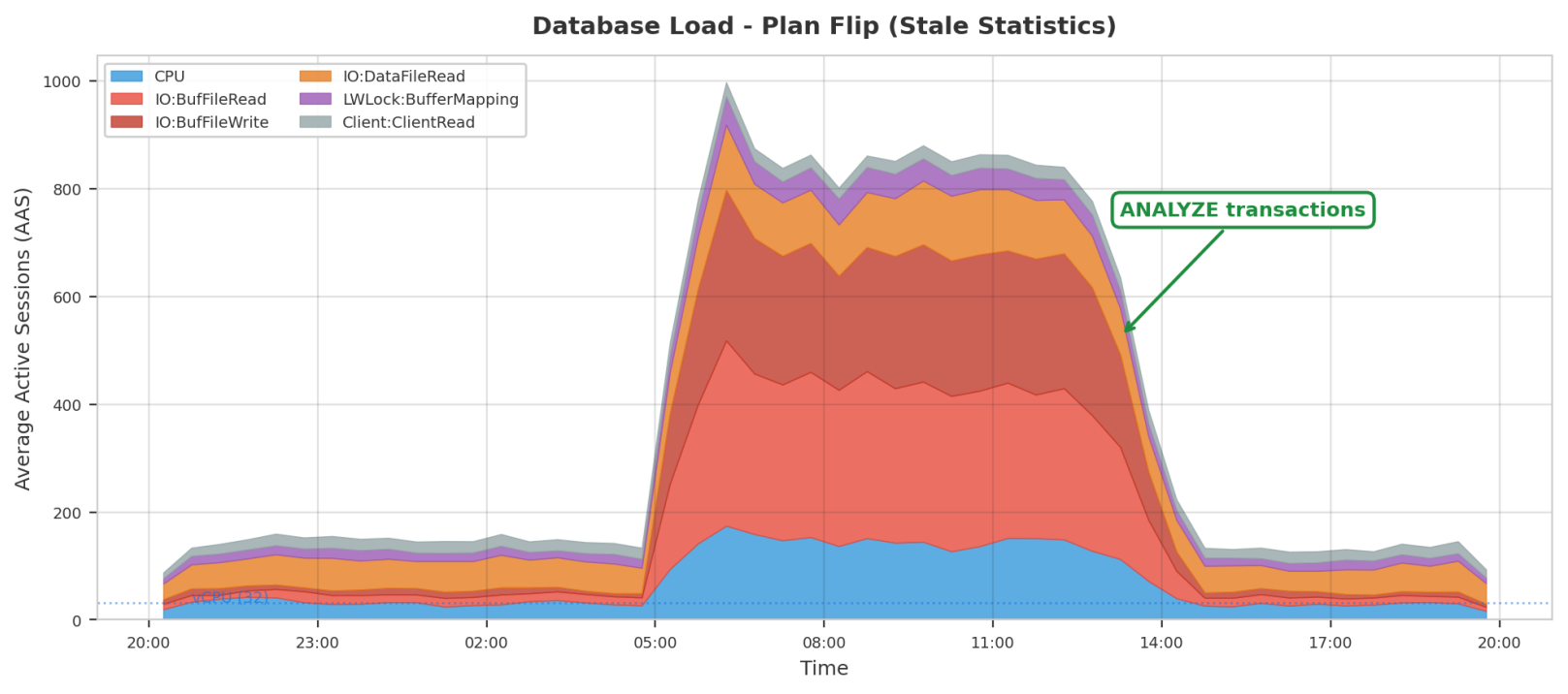
Planner must check all partitions

Result: Full table scan across all partitions

Planning time also increases significantly

Case Study: Dramatic Drop in Performance

Banking Customer: Transaction lookup query went from 50ms → 45 seconds



Case Study: Plan Flip

Investigating performance drop

GOOD PLAN (Nested Loop) - 50ms, 0 temp files:

```
Nested Loop  (cost=0.85..12.34 rows=1)
  ->  Index Scan on accounts (account_id = 12345)
  ->  Index Scan on transactions (account_id = 12345)
Temp Files: 0 bytes
```

BAD PLAN (Hash Join) - 45 seconds, 45GB temp files:

```
Hash Join  (cost=45678.23..89234.56 rows=500000)
  ->  Seq Scan on transactions (2B rows scanned!)
  ->  Hash on accounts (Filter: account_id = 12345)
Temp Files: 45GB written to disk
```

Case Study: Plan Flip 📊

Root cause analysis and long-term fix

🔍 Root Cause: Stale statistics

Fix: (Temporary)

```
ANALYZE Table;
```

🔧 **Long Term Fix: Set threshold at table level for very large tables**

```
ALTER TABLE transactions SET
(autovacuum_analyze_threshold = 50000);
```

OLTP vs Analytics: The Identity Crisis 🎭

Product Manager: "Can we run analytics on production?" 🥺

DBA: "Sure! Just once though..." 💀

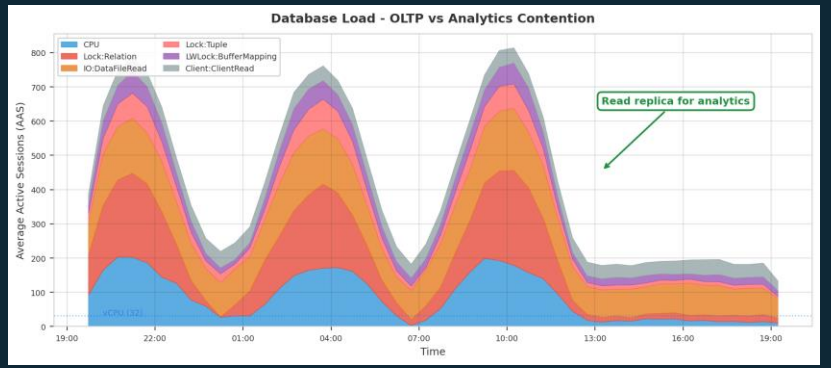
What Happens:

Analytics query scans 500M rows → locks acquired 🔒

OLTP writes waiting... waiting... timeout! ⏰

Connection pool exhausted, app servers panic 🚨

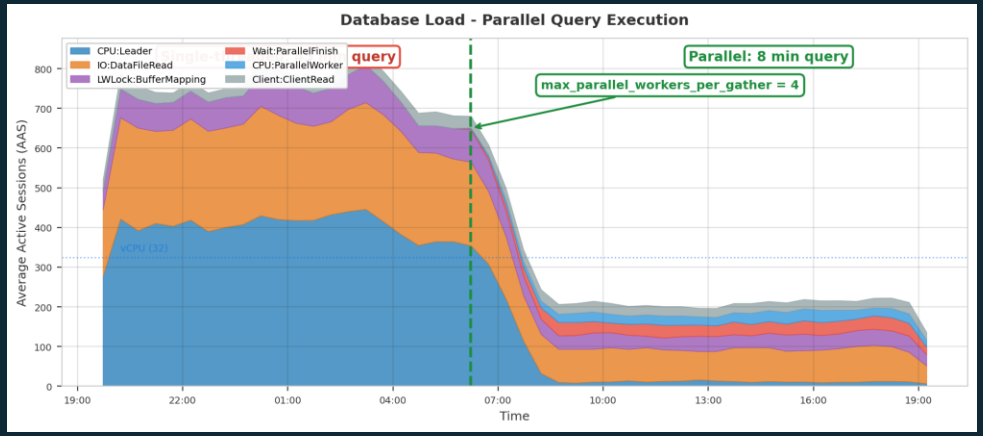
CEO's dashboard killed the checkout page 💸



The Truth: You can't serve two beasts. but if you must, tune for the one that bites.

Parallel Query Execution

- Essential for analytics workloads
- Key parameters:
  - max_parallel_workers_per_gather
  - max_parallel_workers
  - parallel_setup_cost, parallel_tuple_cost
- Not all queries benefit from parallelism
- Watch for parallel query bottlenecks
- Balance with concurrent workload



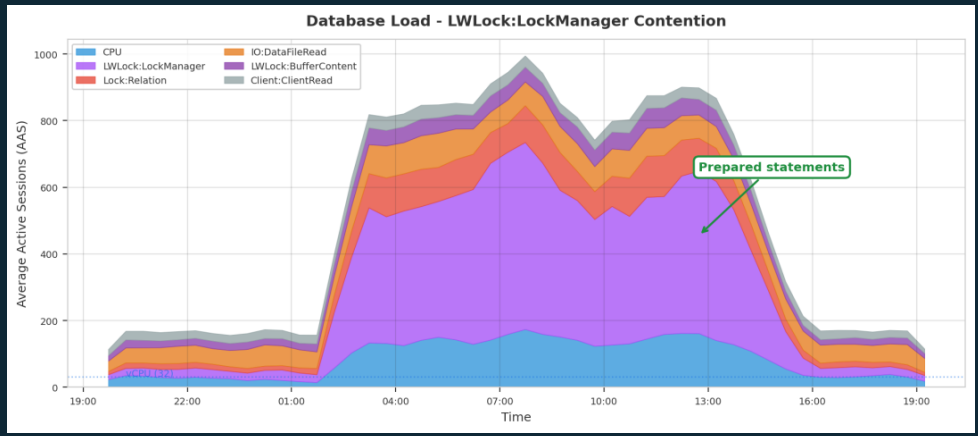

```
SELECT datname, parallel_workers_to_launch, parallel_workers_launched
FROM pg_stat_database
WHERE datname = current_database();
```

LWLock:LockManager Contention

High-concurrency problem at scale

Symptoms:

- Queries waiting on LWLock:LockManager
- High CPU usage but low throughput
- Wait events spike during peak load



Root cause: Lock table contention

- Every query acquires AccessShareLock
- Table + ALL indexes locked during planning

At scale: Thousands of locks = bottleneck

https://gitlab.com/gitlab-com/gl-infra/observability/team/-/work_items/2301

The Hidden Lock Explosion

Simple SELECT locks more than you think:

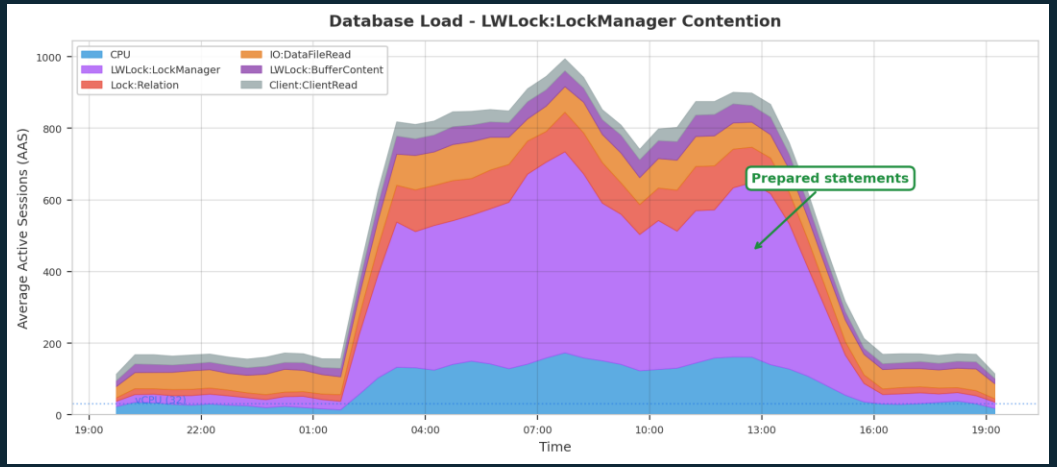
- Table: 1 lock
- Every index: 1 lock each

Example: Table with 20 indexes = 21 locks

High concurrency scenario:

- 1000 concurrent queries
- 20 indexes per table
- = 21,000 locks in lock manager

Result: LWLock contention kills performance

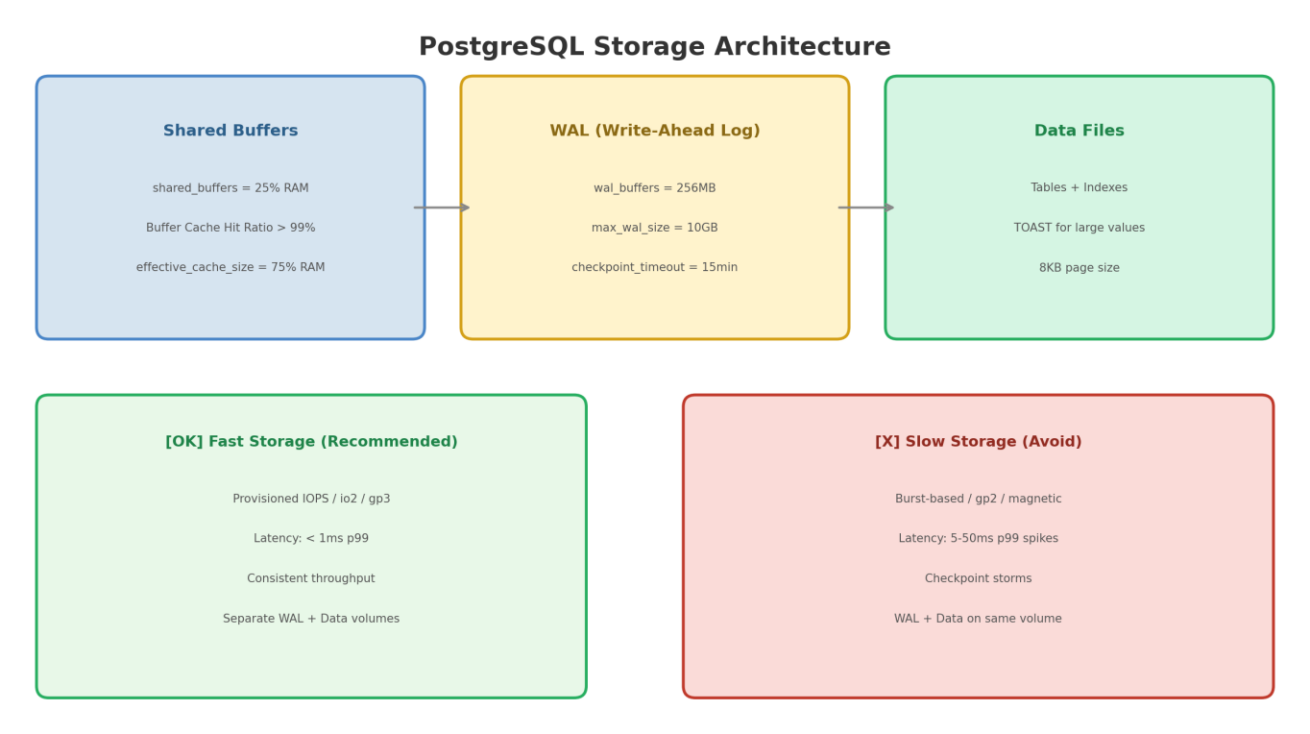


https://gitlab.com/gitlab-com/gl-infra/observability/team/-/work_items/2301
NUM_LOCK_PARTITIONS = 16

Transaction, Durability, and IO

PostgreSQL Storage Architecture:

Dos and Don’ts

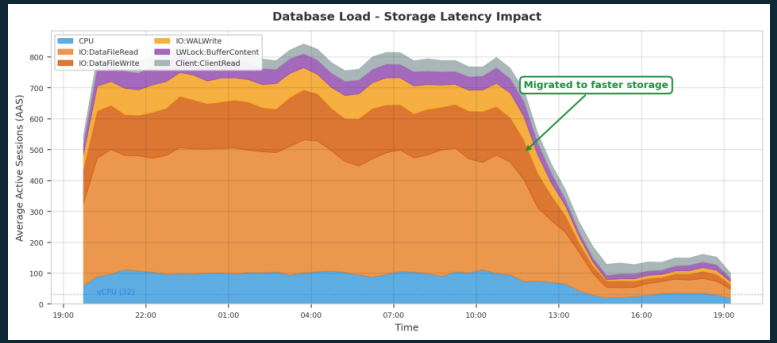


Storage Choices: The $100K Mistake 💰

Month 1: "Let's save money with under-provisioned storage!" 💡
Month 2: "Why are checkpoints taking 10 minutes?" 🤔
Month 3: "Latency spikes every 5 minutes..." 📊
Month 4: "Right-sized storage + HA. Fast AND reliable." ✅
Month 5: "Should have done this from day one." 💪

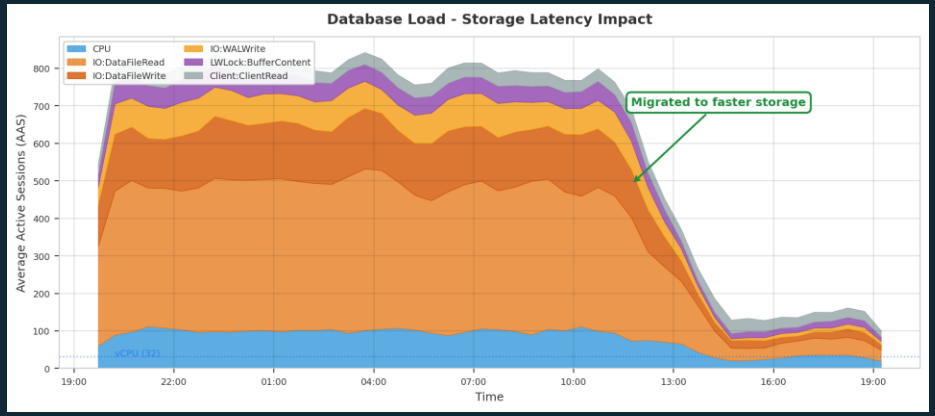
The Lesson: Size storage for your I/O profile, not just capacity

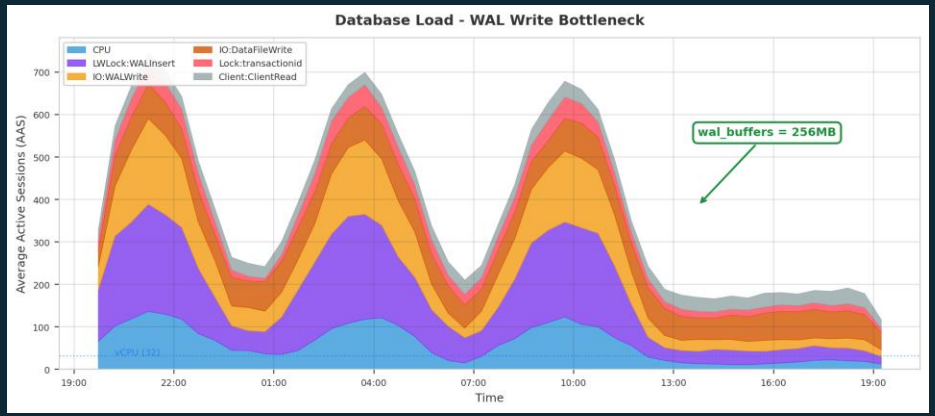
- Understand sustained write throughput needs (WAL)
- Random read IOPS for query workloads
- Checkpoint I/O bursts need headroom
- Test under realistic load before committing



WAL & I/O Considerations

- WAL generation scales with write volume
- wal_compression can reduce I/O pressure
- Separate WAL on fast storage (NVMe)
- checkpoint_timeout and max_wal_size tuning
- Monitor checkpoint frequency and duration
- Avoid checkpoint storms during peak hours





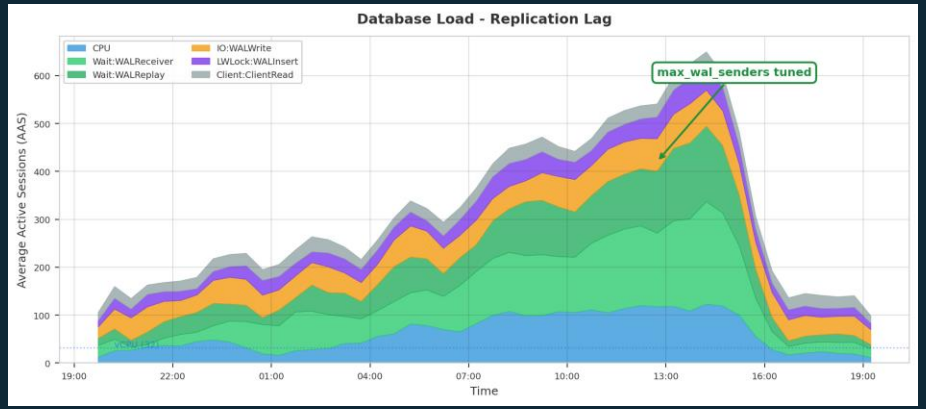
Replication

Replication at Scale: When Replicas Rebel 🔥

2 AM Alert: "Replica lag: 6 hours" 🚨

The Investigation:

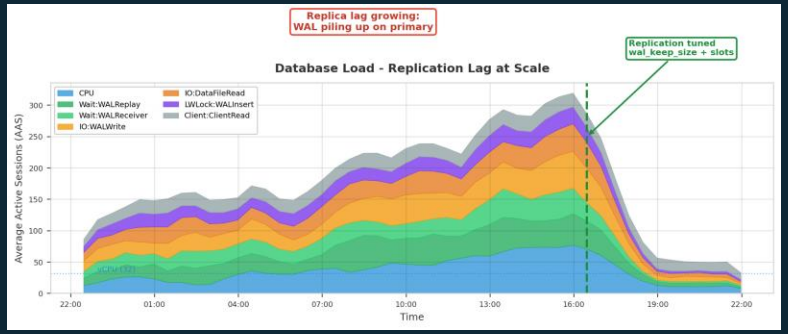
- Primary: Happily writing 50K TPS 🚀
- Replica: Single-threaded WAL replay at 5K TPS 🐌
- Replication slot: 500GB of WAL piling up 📦
- Disk: 95% full, autovacuum can't run 💀
- Failover plan: "Don't even think about it" ⚠️



Reality Check: Your HA setup is only as good as your fastest replica.

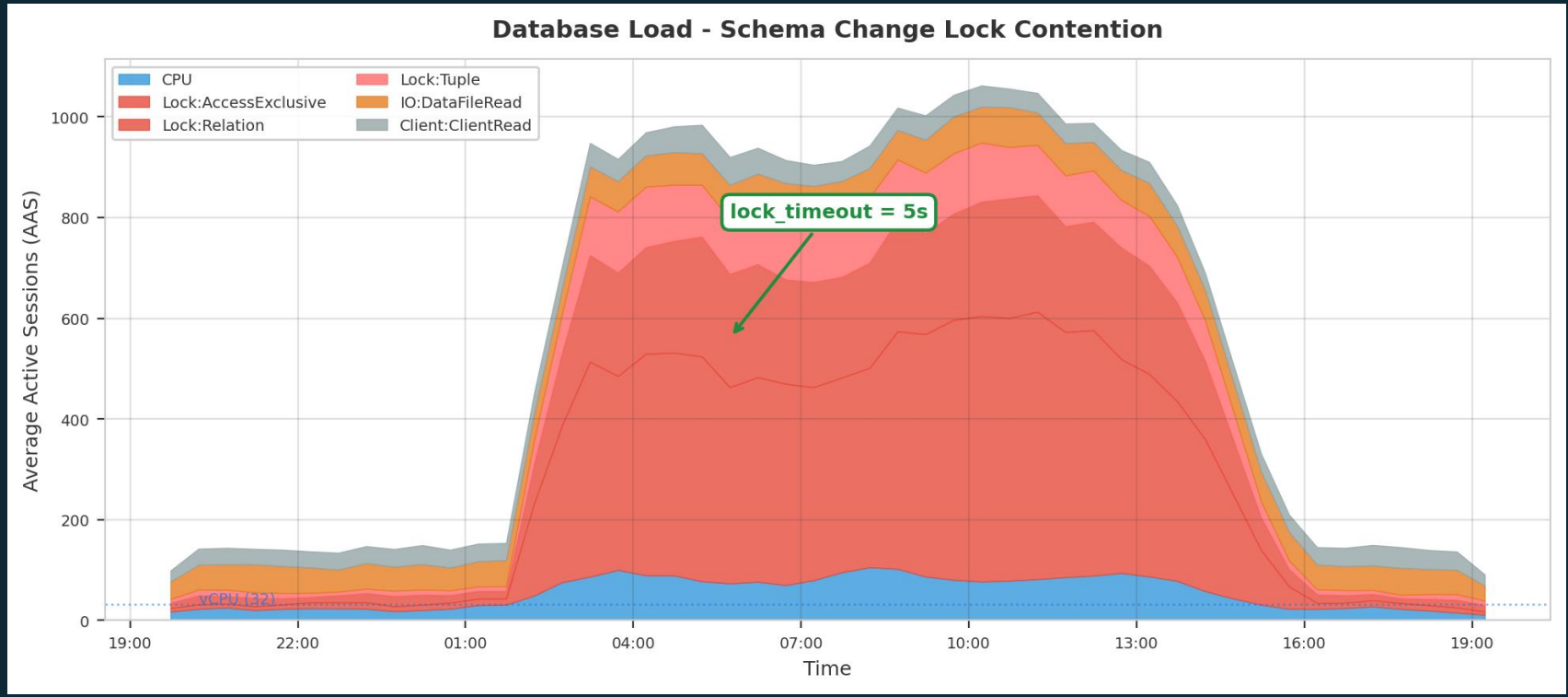
Replication at Scale

- Streaming replication: proven and reliable
- Challenges at scale:
  - Replication lag during high write load
  - Network bandwidth becomes bottleneck
  - Replica query conflicts and cancellations
- Tuning: wal_sender_timeout, max_wal_senders
- Monitor replication slots for disk bloat



Administrative Tasks

Schema Changes on Massive Tables



ALTER TABLE can lock for hours

Schema Changes on Massive Tables

Safe approaches:

- ADD COLUMN with DEFAULT (PG 11+: fast)
- CREATE INDEX CONCURRENTLY
- DROP INDEX CONCURRENTLY

Avoid:

ALTER COLUMN TYPE, ADD CONSTRAINT

Use shadow tables/ Blue Green for complex changes

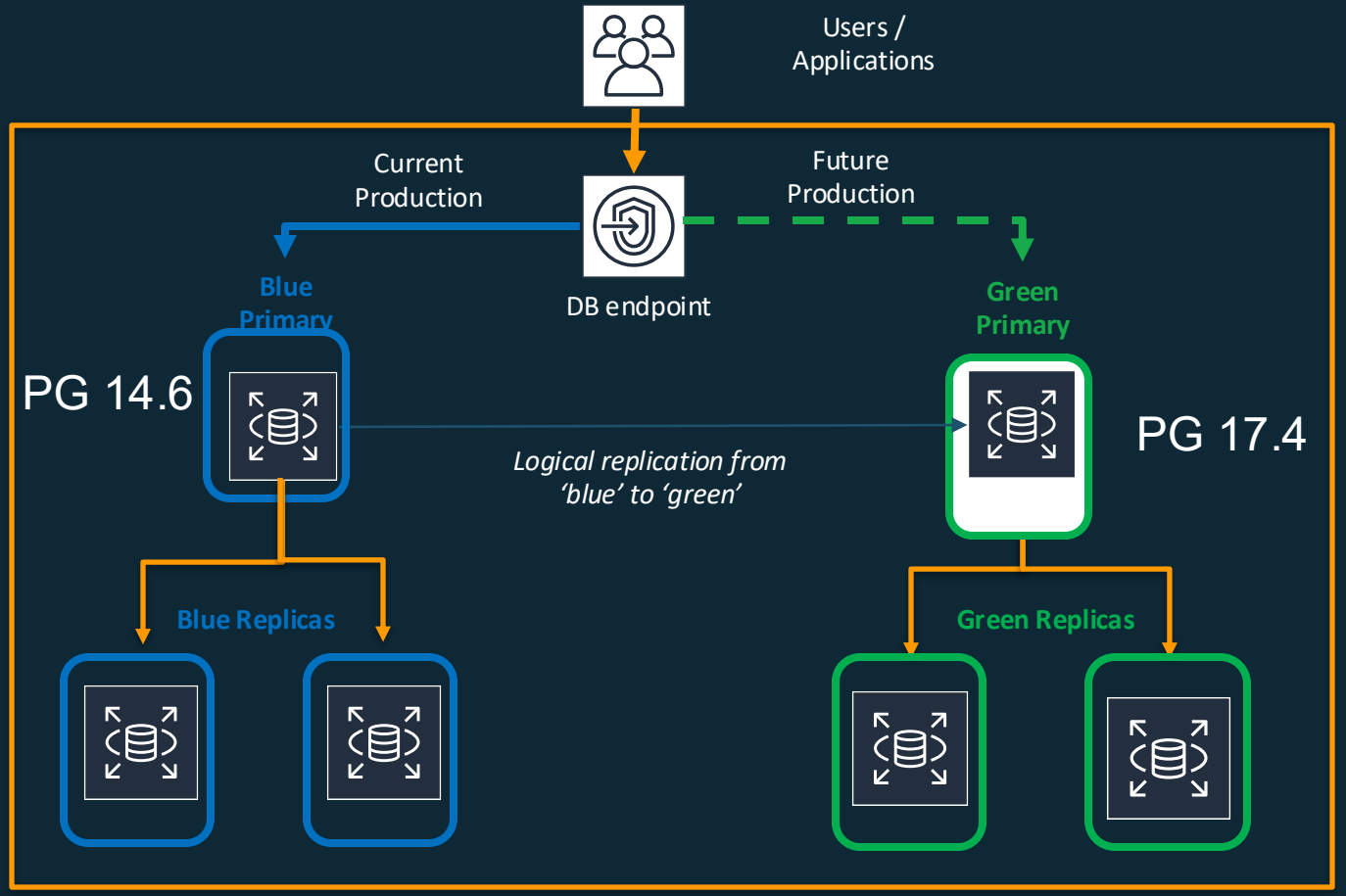
Instead of:

```
ALTER TABLE "accounts" ADD CONSTRAINT
"positive_balance" CHECK ("balance" >=
0);
```

Use:

```
ALTER TABLE "accounts" ADD CONSTRAINT
"positive_balance" CHECK ("balance" >= 0)
NOT VALID;
ALTER TABLE accounts VALIDATE CONSTRAINT
positive_balance;
```

Use Blue-Green Strategy for Version Upgrade



Use Blue-Green Strategy for Version Upgrade

Benefits at scale:

- Zero downtime for major version upgrades
- Full testing on production-like data
- Instant rollback capability
- Validate performance before cutover

Monitoring at Scale

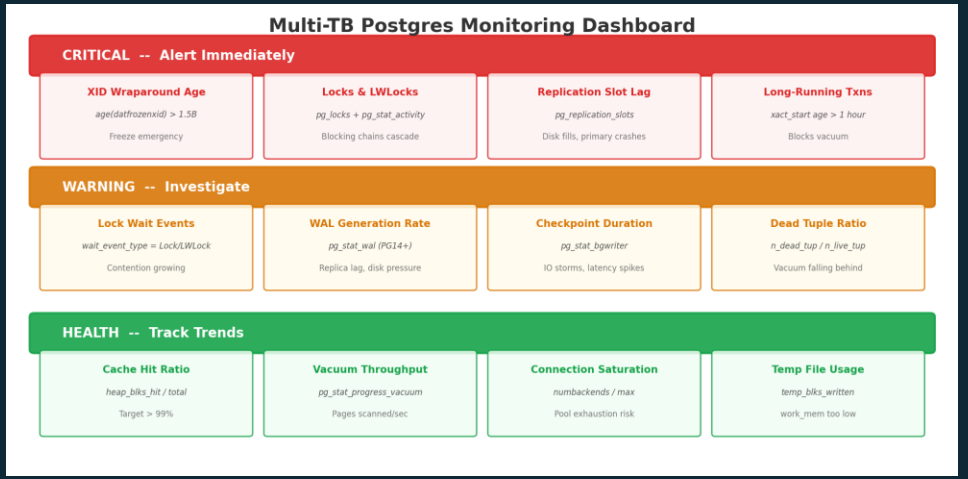
Essential metrics:

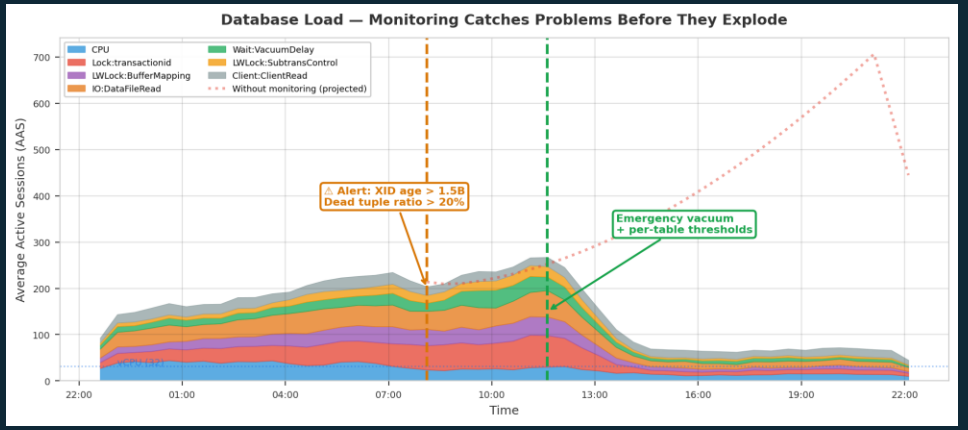
- Locks and LWLocks (pg_locks, pg_stat_activity)
- Replication lag (pg_stat_replication)
- Vacuum progress (pg_stat_progress_vacuum)
- Replication slots (pg_replication_slots)
- Bloat (pgstattuple, dead tuple ratio)
- Connection count and wait events

Scale-specific alerts:

- XID wraparound age - age(datfrozenxid) > 200M
- Long-running transactions — blocks vacuum
- WAL generation rate (pg_stat_wal, PG14+)
- Checkpoint duration & frequency
- Cache hit ratio (target > 99% at TB scale)
- Temp file usage (work_mem too low)

Real-time dashboards for operations





Designing Actionable Alerts

Alert on trends, not absolute values

Key alerts:

- Replication lag > threshold
- Autovacuum not running on hot tables
- Replication slot disk usage
- Long-running transactions
- Checkpoint frequency spikes

Avoid alert fatigue with proper thresholds

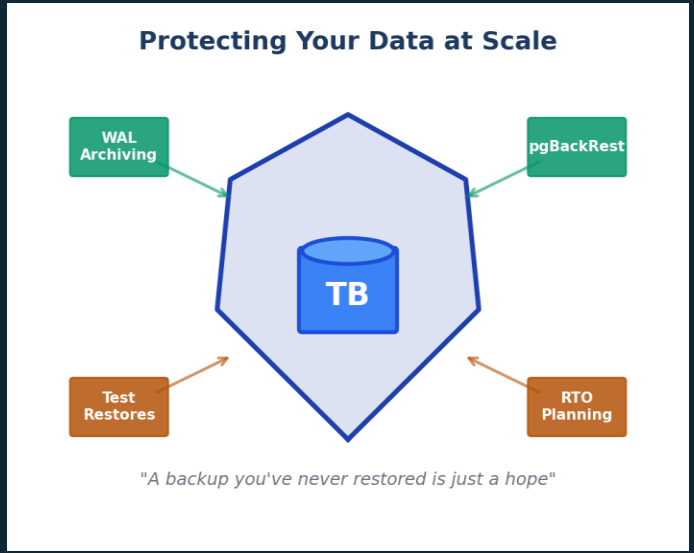
Performance Tuning Checklist

- shared_buffers: 25% of RAM (up to 40GB)
- effective_cache_size: 50-75% of RAM
- work_mem: Per-operation memory (start low)
- maintenance_work_mem: 1-2GB for vacuum/index
- random_page_cost: 1.1 for SSD, 4.0 for slower disk
- effective_io_concurrency: 200 for SSD

Profile with pg_stat_* views

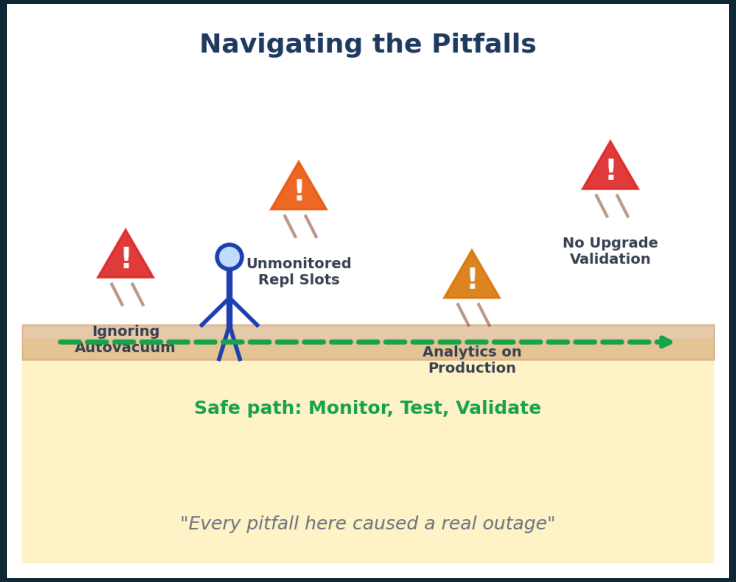
Backup & Recovery at Scale

- pg_basebackup too slow for TB clusters
- Use WAL archiving + incremental backups
- Tools: pgBackRest, Barman, WAL-G
- Test restore regularly (not just backups)
- Point-in-time recovery (PITR) essential
- Consider snapshot-based backups
  (ZFS, LVM)
- Offsite replication for disaster recovery



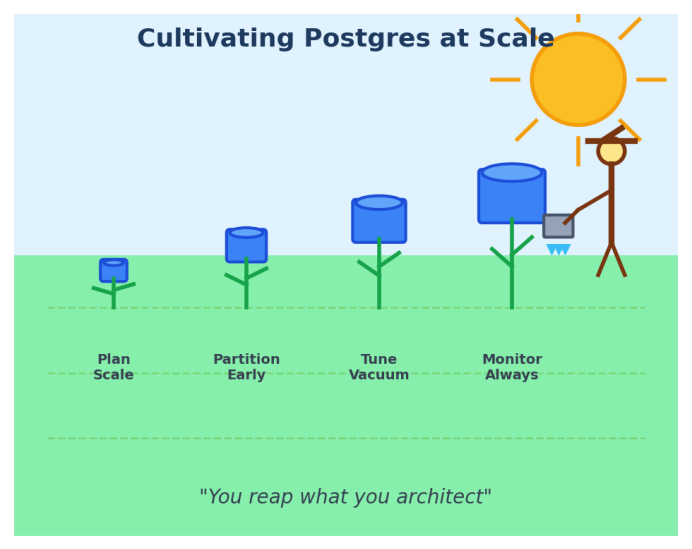
Common Pitfalls at Scale

- Ignoring autovacuum until it's too late
- Not monitoring replication slots
- Underestimating schema change duration
- Running out of transaction IDs
- Insufficient connection pooling
- Not testing failover procedures
- Overlooking disk space for temp files



Hard-Earned Lessons

- Plan for scale from day one
- Partitioning strategy is critical
- Autovacuum tuning is not optional
- Monitor everything, alert on what matters
- Test schema changes on production clones
- Have runbooks for common issues
- Invest in observability infrastructure



Thank You!

Questions?