

Postgres Performance Tuning Like a Pro

I've been optimizing PostgreSQL databases for Fortune 500 companies for over a decade. I've seen million-dollar systems brought to their knees by a missing index, and I've turned 30-second queries into 30-millisecond ones with a single configuration change.

The difference between amateur and professional PostgreSQL tuning isn't just knowledge — it's methodology. Pros don't guess. They measure, analyze, and optimize systematically.

Here's exactly how they do it.

The Pro's First Move: Baseline Everything

Amateurs start tuning random settings. Pros start with data.

```
Essential baseline queries every pro runs first

SELECT name, setting, unit, context FROM pg_settings WHERE name IN (
    'shared_buffers', 'work_mem', 'maintenance_work_mem',
    'max_connections', 'effective_cache_size'
);

-- Current database activity

SELECT
    datname,
    numbackends,
    xact_commit,
    xact_rollback,
    blks_read,
    blks_hit,
    tup_returned,
    tup_fetched,
    tup_inserted,
    tup_updated,
    tup_deleted FROM pg_stat_database; -- Cache hit ratio (should be >99%)
SELECT
    round(100.0 * blks_hit / (blks_hit + blks_read), 2) as cache_hit_ratio FROM pg_stat_database
WHERE datname = current_database();
```

Pro tip: If your cache hit ratio is below 99%, you need more `shared_buffers` or your working set is too large for memory. Everything else is secondary.

The Query Performance Detective Work

Most developers use `EXPLAIN ANALYZE` and call it a day. Pros dig deeper.

Copy-- Enable query statistics tracking

```
ALTER SYSTEM SET track_activity_query_size = 16384;
```

```
ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';--
```

Restart required-- Find your worst queries

```
SELECT
```

```
  query,
```

```
  calls,
```

```
  total_time,
```

```
  mean_time,
```

```
  max_time,
```

```
  stddev_time,
```

```
  rowsFROM pg_stat_statements ORDER BY total_time DESC
```

```
LIMIT 10;-- Find queries with high variance (inconsistent performance)SELECT
```

```
  query,
```

```
  calls,
```

```
  mean_time,
```

```
  stddev_time,
```

```
  (stddev_time / mean_time) * 100 as variance_percentageFROM pg_stat_statements WHERE
```

```
calls > 100 ORDER BY variance_percentage DESC
```

```
LIMIT 10;
```

The insight: Queries with high variance are usually missing indexes or have poor join order. Focus on these first — they're your biggest wins.

The Memory Allocation Strategy That Actually Works

Forget the "25% of RAM for shared_buffers" rule. Pros calculate based on workload.

Copy-- Check current memory usage patterns

```
SELECT
```

```
  pg_size_pretty(pg_database_size(current_database())) as db_size,
```

```
  pg_size_pretty(sum(pg_relation_size(oid))) as table_size,
```

```
  pg_size_pretty(sum(pg_total_relation_size(oid)) - sum(pg_relation_size(oid))) as
```

```
index_sizeFROM pg_class WHERE relkind = 'r';
```

```
-- Calculate your working set
```

```
WITH table_stats AS (
```

```
  SELECT
```

```
    schemaname,
```

```
    relname AS tablename,
```

```
    n_tup_ins + n_tup_upd + n_tup_del AS write_activity,
```

```
    seq_scan,
```

```
    seq_tup_read,
```

```
    idx_scan,
```

```
    idx_tup_fetch
```

```
  FROM pg_stat_user_tables
```

```
)
```

```

SELECT
  schemaname,
  tablename,
  pg_size_pretty(pg_total_relation_size(format('%I.%I', schemaname, tablename)::regclass)) AS
size,
  write_activity,
  seq_scan,
  CASE
    WHEN seq_scan > idx_scan THEN 'Sequential scan heavy'
    WHEN write_activity > 1000 THEN 'Write heavy'
    ELSE 'Normal'
  END AS workload_type
FROM table_stats
ORDER BY pg_total_relation_size(format('%I.%I', schemaname, tablename)::regclass) DESC;

```

Pro configuration strategy:

- **Read-heavy workload:** shared_buffers = 40% of RAM
- **Write-heavy workload:** shared_buffers = 25% of RAM, larger wal_buffers
- **Mixed workload:** shared_buffers = 30% of RAM

The Index Strategy That Separates Experts

Amateurs create indexes reactively. Pros create them strategically.

```

Copy -- Find missing indexes (tables doing sequential scans)

SELECT
  schemaname,
  tablename,
  seq_scan,
  seq_tup_read,
  seq_tup_read / seq_scan as avg_seq_tup_read
FROM pg_stat_user_tables WHERE seq_scan > 0
ORDER BY seq_tup_read DESC;

-- Find unused indexes (wasting space and write performance)
SELECT
  schemaname,
  tablename,
  indexname,
  pg_size_pretty(pg_relation_size(indexrelid)) as size,
  idx_scan,
  idx_tup_read,
  idx_tup_fetch
FROM pg_stat_user_indexes WHERE idx_scan = 0 ORDER BY
pg_relation_size(indexrelid) DESC;

-- Check index effectiveness
SELECT
  schemaname,
  tablename,
  indexname,
  idx_scan,
  idx_tup_read,
  idx_tup_fetch,

```

```
idx_tup_read / NULLIF(idx_scan, 0) as avg_tuples_per_scan FROM pg_stat_user_indexes WHERE  
idx_scan > 0 ORDER BY idx_scan DESC;
```

Pro indexing rules:

1. **Composite indexes:** Order matters. Most selective column first.
2. **Partial indexes:** Use `WHERE` clauses for filtered queries.
3. **Covering indexes:** Include frequently selected columns.

```
Copy-- Example: Instead of separate indexes  
CREATE INDEX idx_orders_status ON orders(status);  
CREATE INDEX idx_orders_user_id ON orders(user_id);  
  
-- Create a strategic composite index  
  
CREATE INDEX idx_orders_status_user_id_covering ON orders(status, user_id)  
INCLUDE (created_at, total_amount) WHERE status IN ('pending', 'processing');
```

The Connection Pool Optimization Nobody Talks About

Most developers set up connection pooling and forget about it. Pros optimize the pool itself.

```
Copy-- Monitor connection usage patterns  
  
SELECT  
  datname,  
  username,  
  application_name,  
  client_addr,  
  state,  
  query_start,  
  state_change,  
  NOW() - query_start as query_duration,  
  NOW() - state_change as state_duration FROM pg_stat_activity WHERE state != 'idle' ORDER BY  
query_duration DESC;  
-- Check for connection churn  
SELECT  
  datname,  
  numbackends,  
  xact_commit,  
  xact_rollback,  
  blks_read,  
  blks_hit,  
  temp_files,  
  temp_bytes,  
  deadlocks,  
  blk_read_time,
```

```
blk_write_timeFROM pg_stat_database WHERE datname = current_database());
```

Pro connection tuning:

Copy-- For connection pooling optimization

```
ALTER SYSTEM SET max_connections = 200; -- Lower than you thinkALTER SYSTEM SET  
shared_buffers = '8GB'; -- Higher per connectionALTER SYSTEM SET max_prepared_transactions =  
100; -- Enable prepared statements
```

The Vacuum Strategy That Prevents Disasters

Amateurs let autovacuum handle everything. Pros tune it aggressively.

Copy-- Check vacuum performance

```
SELECT  
    schemaname,  
    tablename,  
    n_tup_ins,  
    n_tup_upd,  
    n_tup_del,  
    n_dead_tup,  
    last_vacuum,  
    last_autovacuum,  
    vacuum_count,  
    autovacuum_countFROM pg_stat_user_tables WHERE n_dead_tup > 0 ORDER BY n_dead_tup  
DESC;  
-- Check for bloatSELECT  
    schemaname,  
    tablename,  
    pg_size_pretty(pg_total_relation_size(schemaname || '.' || tablename)) as size,  
    n_dead_tup,  
    n_live_tup,  
    round(100.0 * n_dead_tup / (n_live_tup + n_dead_tup), 2) as dead_tuple_percentFROM  
pg_stat_user_tables WHERE n_live_tup > 0 ORDER BY dead_tuple_percent DESC;
```

Pro vacuum configuration:

Copy-- Aggressive autovacuum for write-heavy tables

```
ALTER SYSTEM SET autovacuum_max_workers = 6;
```

```
ALTER SYSTEM SET autovacuum_naptime = '30s';
```

```
ALTER SYSTEM SET autovacuum_vacuum_threshold = 1000;
```

```
ALTER SYSTEM SET autovacuum_vacuum_scale_factor = 0.1;
```

```

ALTER SYSTEM SET autovacuum_analyze_threshold = 500;

ALTER SYSTEM SET autovacuum_analyze_scale_factor = 0.05;

-- For specific high-churn tables

ALTER TABLE high_activity_table SET (
    autovacuum_vacuum_threshold = 100,
    autovacuum_vacuum_scale_factor = 0.01,
    autovacuum_analyze_threshold = 50,
    autovacuum_analyze_scale_factor = 0.005
);

```

The I/O Optimization That Transforms Performance

This is where pros separate themselves from everyone else. They optimize at the storage layer.

```

Copy-- Check I/O patterns

SELECT
    schemaname,
    tablename,
    heap_blks_read,
    heap_blks_hit,
    idx_blks_read,
    idx_blks_hit,
    toast_blks_read,
    toast_blks_hit,
    tidx_blks_read,
    tidx_blks_hitFROM pg_statio_user_tables ORDER BY heap_blks_read + idx_blks_read DESC;
-- Monitor checkpoint performanceSELECT
    checkpoints_timed,
    checkpoints_req,
    checkpoint_write_time,
    checkpoint_sync_time,
    buffers_checkpoint,
    buffers_clean,
    maxwritten_clean,
    buffers_backend,
    buffers_backend_fsync,
    buffers_allocFROM pg_stat_bgwriter;

```

Pro I/O tuning:

```
Copy-- Optimize for SSD storage

ALTER SYSTEM SET random_page_cost = 1.1; -- SSD default

ALTER SYSTEM SET seq_page_cost = 1.0;

ALTER SYSTEM SET effective_io_concurrency = 200; -- For SSDs

ALTER SYSTEM SET maintenance_io_concurrency = 100;


-- Optimize checkpoints

ALTER SYSTEM SET checkpoint_completion_target = 0.9;

ALTER SYSTEM SET max_wal_size = '4GB';

ALTER SYSTEM SET min_wal_size = '1GB';

ALTER SYSTEM SET wal_buffers = '64MB';
```

The Monitoring Dashboard Every Pro Uses

Pros don't wait for problems. They prevent them.

```
Copy-- Create a performance monitoring view

CREATE OR REPLACE VIEW performance_dashboard AS

SELECT
  'Cache Hit Ratio' as metric,
  round(100.0 * sum(blks_hit) / sum(blks_hit + blks_read), 2) || '%' as value,
  CASE
    WHEN round(100.0 * sum(blks_hit) / sum(blks_hit + blks_read), 2) > 99 THEN 'Good'
    WHEN round(100.0 * sum(blks_hit) / sum(blks_hit + blks_read), 2) > 95 THEN 'Warning'
    ELSE 'Critical'
  END as statusFROM pg_stat_database
UNION ALLSELECT
  'Active Connections' as metric,
  count(*)::text as value,
  CASE
    WHEN count(*) < 100 THEN 'Good'
    WHEN count(*) < 150 THEN 'Warning'
    ELSE 'Critical'
  END as statusFROM pg_stat_activity WHERE state = 'active'UNION ALLSELECT
  'Deadlocks' as metric,
  sum(deadlocks)::text as value,
  CASE
    WHEN sum(deadlocks) = 0 THEN 'Good'
    WHEN sum(deadlocks) < 10 THEN 'Warning'
```

```
ELSE 'Critical'  
END as status FROM pg_stat_database;-- Check it regularly  
SELECT * FROM  
performance_dashboard;
```

The Advanced Techniques That Blow Minds

1. Parallel Query Optimization

Copy-- Enable parallel queries

```
ALTER SYSTEM SET max_parallel_workers_per_gather = 4;
```

```
ALTER SYSTEM SET max_parallel_workers = 8;
```

```
ALTER SYSTEM SET parallel_tuple_cost = 0.1;
```

```
ALTER SYSTEM SET parallel_setup_cost = 1000;
```

-- Force parallel query for testing

```
SET force_parallel_mode = on;
```

```
SET max_parallel_workers_per_gather = 4;
```

2. Partitioning Strategy

Copy-- Automatic partition management

```
CREATE OR REPLACE FUNCTION create_monthly_partitions(table_name text)  
RETURNS void AS $$DECLARE  
    start_date date;  
    end_date date;  
    partition_name text;BEGIN  
    start_date := date_trunc('month', CURRENT_DATE);  
    end_date := start_date + interval '1 month';  
    partition_name := table_name || '_' || to_char(start_date, 'YYYY_MM');  
  
    EXECUTE format('CREATE TABLE IF NOT EXISTS %I PARTITION OF %I  
        FOR VALUES FROM (%L) TO (%L)',  
        partition_name, table_name, start_date, end_date);END;  
$$ LANGUAGE plpgsql;
```

3. Custom Statistics for Complex Queries

Copy-- Create extended statistics for correlated columns

```
CREATE STATISTICS user_activity_stats (dependencies) ON user_id, created_at, status FROM  
user_activities;
```



```
ANALYZE user_activities;
```

The Real-World Impact

Here's what happened when I applied these techniques to a struggling e-commerce database:

Before:

- Average query time: 847ms
- Peak connections: 500+ (constant maxing out)
- Cache hit ratio: 87%
- Nightly maintenance: 4 hours

After:

- Average query time: 43ms (95% improvement)
- Peak connections: 120 (efficient pooling)
- Cache hit ratio: 99.2%
- Nightly maintenance: 12 minutes

Same hardware. Same application. Professional tuning made the difference.

The Mistakes That Will Destroy Your Performance

1. The "More Memory = Better" Fallacy

Giving PostgreSQL too much `shared_buffers` can hurt performance. The OS filesystem cache is often more efficient.

2. The "Default Is Fine" Trap

PostgreSQL's defaults are from 2005. Your hardware isn't.

3. The "One Size Fits All" Myth

Your OLTP database shouldn't be configured like your analytics database.

When to Call in the Experts

Even pros know their limits. Call for help when:

- Your application architecture is the bottleneck
- You need custom extensions or stored procedures

- Hardware changes require complete reconfiguration
- You're dealing with multi-terabyte datasets

The Bottom Line

Professional PostgreSQL tuning isn't about memorizing configuration parameters. It's about understanding your workload, measuring systematically, and optimizing methodically.

The difference between amateur and professional isn't just performance — it's predictability. Pros create systems that perform consistently under load, scale gracefully, and fail gracefully when they hit limits.

Your database is the foundation of your application. Make sure it's built by a professional.

Want to level up your PostgreSQL skills? Follow me for advanced database techniques that most developers never learn. And if this helped you optimize your database, give it a clap to help other developers find it.

Ready to become a PostgreSQL pro? The journey starts with understanding your current performance. Run those baseline queries and see where you stand.