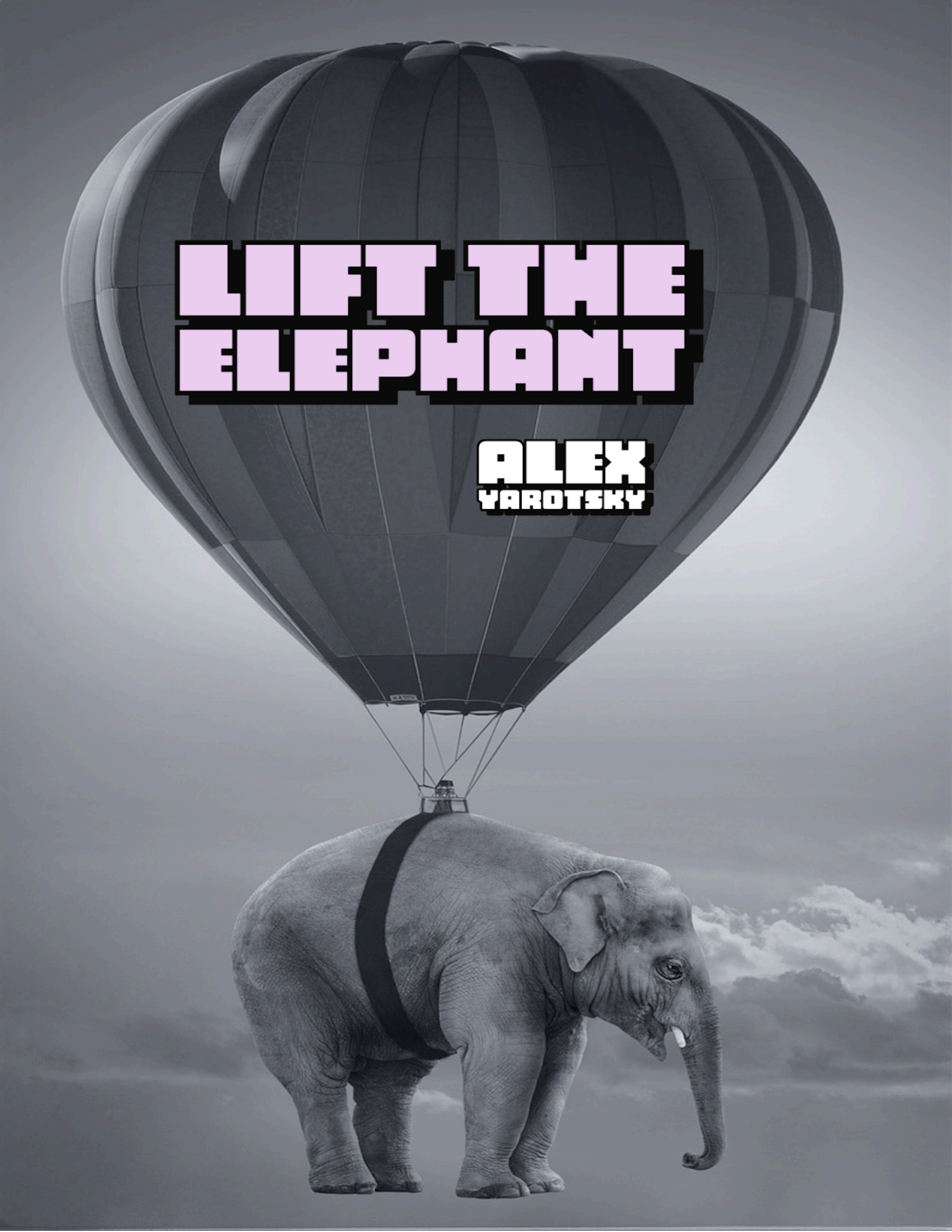
LIFT THE ELEPHANT

ALEX YAROTSKY



Lift the Elephant

Scaling PostgreSQL Beyond Query Optimization

Alex Yarotsky

Lift the Elephant: Scaling PostgreSQL Beyond Query Optimization

Table of Contents

Introduction

Scaling a database is often a journey that starts simply and quickly becomes complicated. PostgreSQL is one of the most trusted and capable databases, but as your application evolves, so do the demands placed on it. What begins as a straightforward setup for data storage can quickly grow into a sprawling ecosystem that requires strategic tuning, scaling, and diligent maintenance.

This book is about navigating that journey. It's not a beginner's guide or a list of theoretical best practices—it's a playbook for tackling the real-world challenges of scaling Postgres in high-demand environments. From parameter tuning and connection pooling to partitioning and active-active setups, I'll walk you through the techniques and strategies that enable Postgres to thrive under pressure.

These lessons come from scaling Hubstaff, a platform that serves thousands of businesses worldwide. As the CTO, I've been in the trenches of database scaling, learning through late-night incident mitigations, unexpected bottlenecks, and breakthrough optimizations. Scaling Postgres has often felt like lifting an elephant: a task that seems impossible until you figure out how to approach it.

Beyond Simple Query Optimization

Most database optimization journeys start with query tuning, and for good reason—it's often the easiest lever to pull. But as your database grows, performance bottlenecks and scaling challenges demand solutions far more sophisticated than simple SQL rewriting. Truly optimizing Postgres requires a holistic approach that encompasses system architecture, runtime behavior, and proactive maintenance.

In this book, we'll explore these deeper layers of Postgres optimization, focusing on topics like:

- **Parameter Tuning:** How to configure Postgres to meet the specific needs of your workload, from memory settings to autovacuum thresholds.
- **Monitoring:** Using metrics to anticipate issues, track performance trends, and respond before small problems become big ones.
- **High-Impact Maintenance:** Performing essential tasks like vacuuming and repacking to keep Postgres running smoothly over time.
- **Connection Pooling:** Managing database connections efficiently as application traffic grows.
- **Active-active and Multi-Reader Setups:** Designing distributed architectures for high availability and concurrency.
- **Partitioning:** Breaking down large datasets to maintain performance and enable horizontal scalability.

Scaling Postgres, Pragmatically

The lessons in *Lift the Elephant* are grounded in real-world experience rather than theory. Every technique I share is one we've used at Hubstaff to tackle actual scaling challenges and keep our platform fast, reliable, and ready for growth. My goal is to give you actionable advice that you can apply to your own database challenges.

Who This Book Is For

This book is for developers, DBAs, and engineering managers who already have experience with Postgres and are ready to level up. Whether you're managing a growing side project or scaling a mission-critical application for millions of users, this book provides the tools, strategies, and mindset to make Postgres work for you.

Ready to Lift?

Scaling Postgres isn't easy, but it's rewarding. With the right knowledge and approach, you can transform Postgres into a high-performance, scalable system that meets your application's toughest demands. Let's dive in and start lifting the elephant.

What is synchronous_commit all about?

Imagine you’re standing at the crossroads of speed and safety. On one side lies performance—blazing fast, no waiting, but with some risks. On the other side, durability—a guarantee that no matter what happens, your data is safe. In PostgreSQL, the **synchronous_commit** parameter lets you choose your path. It’s the decision point where you control how your database prioritizes these competing goals. The key to making the right choice lies in understanding the nuances of this powerful configuration setting.

The Concept: When Performance Meets Assurance

Every transaction in PostgreSQL is written to the Write-Ahead Log (WAL), a journal of changes ensuring data integrity. The **synchronous_commit** parameter determines when a transaction is considered complete. By defining how far WAL records propagate before acknowledgment, this parameter enables a precise balance between performance and durability.

Should PostgreSQL wait for data to be written to a standby server before confirming the transaction, or should it proceed immediately, trusting the primary? Adjusting this parameter aligns your database’s behavior with your application's specific needs.

This is not merely about configuration; it’s about intentionality. Are you prioritizing unshakable reliability, peak performance, or a strategic blend of both? Let’s explore how **synchronous_commit** delivers these options.

Modes of Synchronous Commit

ON: Maximum Durability

Picture a financial system managing sensitive transactions. Losing even a single transaction could lead to financial discrepancies or compliance violations. With **synchronous_commit** set to **ON**, PostgreSQL waits until WAL is written to disk on the primary server and acknowledged by at least one synchronous standby before confirming the transaction.

This mode ensures absolute data safety, even in the event of a primary server failure. While the trade-off is slower performance due to the additional wait time, it's indispensable for systems where data loss is unacceptable—such as payment processing or healthcare records.

REMOTE_WRITE: Balanced Efficiency

Now imagine an analytics platform processing high volumes of data. Speed is essential, but a degree of durability remains necessary. In **REMOTE_WRITE**, PostgreSQL considers the transaction complete once the WAL is successfully written to a synchronous standby, but it doesn't wait for the data to be flushed to disk.

This mode reduces latency while maintaining some assurance against data loss. It's ideal for scenarios where performance is critical but absolute durability isn't required, such as inventory tracking or near-real-time analytics.

LOCAL: Primary-Centric Safety

In environments relying on the primary server, such as development or staging setups, **LOCAL** mode is a practical choice. Transactions are confirmed once WAL is written and flushed to the primary server's disk, with no dependency on standbys.

This mode ensures faster performance while maintaining data durability on the primary server. It's well-suited for workloads where replication lag is acceptable or secondary systems are non-critical.

OFF: Performance Over Assurance

For logging systems or ephemeral data, where speed trumps durability, **OFF** mode is ideal. Transactions are acknowledged immediately after being logged in memory, without waiting for WAL writes or replication.

While this mode maximizes throughput, it risks losing recent transactions if the server crashes. It’s a calculated trade-off, perfect for workloads like telemetry or temporary metrics where occasional data loss is acceptable.

REMOTE_APPLY: Synchronous Consistency

The **REMOTE_APPLY** mode, a relatively newer addition, ensures the highest level of consistency. Transactions wait until WAL is written, flushed, and applied on the standby, making the data immediately queryable on both the primary and standby servers.

This guarantees synchronized consistency at the cost of higher latency. It’s ideal for systems requiring consistent reads across primary and standby servers, such as critical analytics platforms.

Practical Scenarios and Recommendations

Choosing the right mode depends on your workload’s demands. Consider these examples:

- **Critical Transactions:** Payment systems and order processing should use **ON** for guaranteed durability.
- **Moderately Critical Data:** Inventory updates or high-throughput analytics benefit from **REMOTE_WRITE**, balancing speed and safety.
- **Non-Critical Operations:** Use **OFF** for logs and telemetry to prioritize performance.

- **Synchronized Queries:** Applications requiring identical reads across servers should use **REMOTE_APPLY** for consistency.

The Value of Contextual Tuning

The true power of **synchronous_commit** lies in its flexibility. PostgreSQL allows this parameter to be set globally, at the database level, or even per transaction. This granularity enables developers to optimize specific processes without overburdening the entire system.

For example, critical transactions can use **REMOTE_WRITE** during peak hours, while less critical operations default to **OFF**. Dynamically adjusting the parameter through session-level commands ensures that your database remains agile and responsive to changing demands.

Monitor, Adapt, Perfect

Effective tuning requires continuous monitoring. Use **pg_stat_replication** to track replication lag and performance. Evaluate how different modes impact latency and throughput under real-world workloads. Monitoring these metrics helps you refine configurations and maintain optimal performance.

Additionally, PostgreSQL's **synchronous_standby_names** parameter lets you specify which standbys must confirm transactions, giving you precise control over replication. By balancing these settings, you can achieve the ideal mix of durability and performance.

Wrapping Up

The **synchronous_commit** parameter is more than just a technical setting—it's a strategic tool. It empowers you to define how your database responds to the competing demands of speed and safety. By understanding its modes and applying them thoughtfully, you can build a system that perfectly aligns with your application’s needs.

Closing Thoughts

If you've made it this far, you now have a mental model for PostgreSQL scaling that goes well beyond **EXPLAIN ANALYZE**. You understand that database performance isn't a single problem with a single solution—it's a system of interacting forces that demand attention at every layer, from memory allocation and vacuum tuning to connection pooling and distributed architectures.

The Scaling Mindset

When I started scaling Postgres at Hubstaff, I made the mistake most engineers make: I treated each performance issue as an isolated incident. A slow query here, a spike in I/O there. Fix it, move on, repeat. It took painful experience to realize that scaling a database is fundamentally about building systems that prevent problems rather than heroically solving them after the fact.

The chapters in this book reflect that shift in thinking. Parameter tuning isn't about finding magic numbers—it's about understanding what your workload actually demands and configuring Postgres to meet it. Monitoring isn't about dashboards that look impressive—it's about catching transaction ID wraparound or checkpoint storms before they wake you up at 3 am. Autovacuum isn't a set-and-forget feature—it's a continuous negotiation between maintenance overhead and query performance that requires active management as your data grows.

What Separates Good From Great

In my experience, the difference between a Postgres deployment that merely works and one that scales reliably comes down to three habits.

First, measure before you tune. It's tempting to copy configuration values from a blog post or a conference talk. But your workload is not my workload. The settings that keep Hubstaff running smoothly might cause problems on your system. Use **pg_stat_statements**, monitor your checkpoint frequency, watch your bloat ratios, and let the data guide your decisions. Every recommendation in this book comes with the implicit caveat: verify it against your own metrics.

Second, understand the trade-offs. Nearly every tuning decision in Postgres involves a trade-off. Raising **work_mem** speeds up sorts but risks memory exhaustion under concurrency. Aggressive autovacuum keeps bloat in check but competes with your queries for I/O. Synchronous replication guarantees durability but adds latency. There are no free lunches in database engineering. The goal isn't to eliminate trade-offs—it's to make them deliberately, with full awareness of what you're gaining and what you're giving up.

Third, build for the next order of magnitude. The decisions you make at 100 GB of data will not hold at 1 TB. The connection pooling strategy that works for 50 concurrent users will buckle at 500. Partitioning that seems premature today becomes an emergency migration six months from now. You don't need to over-engineer, but you do need to think one step ahead. When you're designing your schema, your replication topology, or your vacuum strategy, ask yourself: what breaks when this doubles?

The Elephant Doesn't Stay Still

PostgreSQL itself continues to evolve. Each major release brings improvements to parallel query execution, partitioning, vacuum performance, logical replication, and the query planner. The fundamentals covered in this book—understanding how Postgres manages memory, handles concurrency, and maintains data integrity—will remain relevant regardless of version. But the specific thresholds, defaults, and best practices will shift as the database matures.

Stay engaged with the PostgreSQL community. Read the release notes for each major version. Test new features against your workload before adopting them. The engineers contributing to Postgres are solving problems at the frontier of what's possible with a relational database, and their work often obsoletes workarounds that were necessary just a few years ago.

Your Turn

Scaling Postgres is not a destination. It's a practice. You'll tune parameters, stabilize for a while, then hit a new ceiling as your application grows. You'll add read replicas, rethink your partitioning strategy, reconfigure your connection pooler, and revisit chapters of this book with fresh eyes and harder questions.

That's exactly how it should be. Every scaling challenge you solve deepens your understanding of how Postgres works under the hood, and that understanding compounds over time. The engineer who has fought through a transaction ID wraparound scare, debugged a bloated table that brought queries to a crawl, or migrated to an active-active setup under production load—that engineer has intuitions that no amount of documentation can replace.

I wrote this book because I believe the PostgreSQL ecosystem deserves more practical, experience-driven guidance on scaling. Too much of the existing material is either too theoretical to act on or too superficial to trust in production. My hope is that the lessons from Hubstaff's scaling journey have given you concrete tools and strategies you can apply to your own systems—and the confidence to tackle problems that once seemed impossible.

The elephant is heavy. But you know how to lift it now.