Comparative Analysis of SQLite and PostgreSQL Under High Concurrency

Tyler O'Brien (tjo656)

Department of Aerospace Engineering and Engineering Mechanics
Cockrell School of Engineering, The University of Texas at Austin
Oden Instute For Computational Engineering and Science

Abstract :

High-concurrency scientific services often depend on reliable, low-latency database operations, yet many systems struggle to scale when subjected to large volumes of simultaneous reads and writes. Motivated by similar behavior observed in the Trust Manager System (TMS) at TACC, this project investigates how two widely used database engines—SQLite and PostgreSQL—respond under controlled concurrency stress. To isolate database performance from unrelated factors, I developed two nearly identical Rust servers whose only difference is the underlying database backend, along with a Goose-based load-testing harness capable of generating reproducible, high-intensity workloads.

Using this framework, I collected latency, throughput, and failure metrics across increasing user loads and profiled system behavior with system-level CPU monitoring. Verification tests confirmed consistent correctness between both databases at low concurrency, while profiling revealed clear divergence under write-heavy stress: SQLite quickly becomes bottlenecked by its single-writer locking model, whereas PostgreSQL maintains stable performance through multi-writer scheduling. Performance improved measurably through compiler optimizations and connection-pool tuning, demonstrating how implementation decisions influence real-world behavior.

Overall, this work provides a reproducible methodology for database performance analysis in scientific computing environments and offers insight into the scalability limitations that may impact systems like TMS.

1 Introduction

Modern scientific computing systems frequently rely on database-backed services that must remain responsive under high concurrency. In practice, however, many services experience severe slowdowns or even complete freezes when subjected to sustained read and write pressure. Similar behavior has been observed in the Trust Manager System (TMS) at the Texas Advanced Computing Center (TACC), where SQLite is used as the underlying storage engine. SQLite is lightweight and simple to deploy, but its single-writer concurrency model raises concerns about scalability in multi-user environments. These observations motivated a focused investigation into how SQLite behaves under controlled stress, and how an alternative engine such as PostgreSQL compares when subjected to identical workloads.

Motivation

This project aims to understand why certain database-backed services fail under load and to evaluate whether database architecture plays a significant role in these failures. SQLite—despite being reliable and widely deployed—supports only a single concurrent writer and relies on file-level locking, creating potential bottlenecks when many clients issue write-heavy requests. PostgreSQL, by contrast, supports true multi-writer concurrency through MVCC (multi-version concurrency control), making it a natural point of comparison. By isolating database behavior from unrelated system factors, this project provides insight into scaling limits relevant not only to classroom experiments but also to real production systems like TMS.

Objectives

The goals of this project are fourfold:

1. Construct two equivalent Rust-based servers whose only difference is the choice of database backend: SQLite or PostgreSQL.
2. Generate reproducible, high-concurrency workloads using the Goose framework to stress each database under baseline, read-heavy, and write-heavy scenarios.
3. Profile and analyze performance using latency, throughput, error rates, and system-level CPU observations.
4. Validate correctness between databases at low concurrency and document the optimization steps that influence performance.

Contributions

This project provides:

- A reproducible benchmarking framework composed of two Rust servers and a parameterized Goose load-testing harness.
- A detailed comparison of SQLite and PostgreSQL under controlled concurrent workloads, revealing how architectural differences impact scalability.
- Profiling results using flamegraphs and system tools that highlight specific bottlenecks, such as SQLite's writer lock contention.
- A verification strategy ensuring functional correctness across both databases and an optimization history illustrating the effect of compiler and structural improvements.

2 System Architecture and Key Technologies

This section describes the overall architecture of the benchmarking framework and the software components used to construct, test, and profile the SQLite and PostgreSQL servers. The primary goal of the architecture is to ensure a fair, controlled comparison between the two databases by keeping all non-database components identical across experiments. Each server exposes the same set of benchmark endpoints, performs the same logical operations, and is subjected to the same Goose-generated workloads under

equivalent conditions.

High–Level Architecture

Both servers follow a shared design pattern: client requests trigger a route handler, which executes a fixed sequence of SQL operations and returns a JSON response. The only point of divergence between the SQLite-based and PostgreSQL-based servers lies in their database initialization and connection setup. All other application logic, endpoint code, and concurrency behavior originate from the Rust web server itself.

- **SQLite Server:** Uses a file-backed database accessed through SQLx. Although SQLite supports multiple concurrent readers, it permits only one writer at a time, introducing contention under write-heavy workloads.
- **PostgreSQL Server:** Connects to a running PostgreSQL instance and uses SQLx's asynchronous connection pool. PostgreSQL supports true multi-writer concurrency using MVCC (multi-version concurrency control).

A simplified architectural flow is as follows:

1. A Goose client issues an HTTP request to one of the benchmark endpoints.
2. The Poem-based Rust server receives the request and dispatches it to the appropriate handler.
3. The handler executes a sequence of SQL operations using SQLx.
4. The database engine processes the workload according to its concurrency model.
5. Results are returned to Goose, which records latency, throughput, and error information.

This structure allows the experiment to isolate differences in performance to the database engines themselves rather than unrelated system factors.

SQLite Concurrency Model

SQLite implements a multi-reader, single-writer concurrency model enforced through file-level locking. While multiple clients may read simultaneously, any operation that writes to the database file requires acquiring an exclusive lock, blocking other writers and all readers during the write. Under light usage this locking granularity is acceptable, but at high concurrency levels it leads to queueing, contention, and eventually request failures or extreme latency. These behaviors make SQLite a useful baseline for understanding the limitations of lightweight embedded databases in multi-user scientific environments.

PostgreSQL Concurrency Model

PostgreSQL uses multi-version concurrency control (MVCC), allowing concurrent reads and writes without relying on coarse file locks. Writers do not block readers, and multiple writers may make progress simultaneously. PostgreSQL resolves conflicts using row-level locks and snapshots rather than serializing all writes. This architectural difference is expected to yield significantly higher throughput and lower latency under write-heavy or mixed workloads. The benchmarking framework allows these expectations to be quantitatively validated.

Rust Server Structure

Both servers share the same Rust codebase except for their database initialization logic. The primary components are:

- **Tokio Runtime:** Provides asynchronous execution for handling many concurrent HTTP requests efficiently.
- **Poem Web Framework:** Defines routing, handlers, and JSON serialization.
- **SQLx Library:** Enables compile-time–checked SQL queries and connection pooling for PostgreSQL. SQLite uses a lightweight connection mode appropriate for file-backed databases.
- **Endpoint Logic:** Each endpoint executes a fixed sequence of SQL statements to ensure identical workloads across both databases.

Maintaining structural parity between the two servers is essential for attributing performance differences solely to the database engines rather than implementation details.

Supporting Libraries and Tools

Several auxiliary tools supported testing and performance analysis throughout the project:

- **Goose Load Testing Framework:** Used to generate concurrent clients, define workload scenarios, and record performance metrics such as latency and throughput for both database backends.
- **System Monitoring Tools:** Utilities such as `top` and `htop` provided real-time visibility into CPU usage, memory consumption, and general process load during high-concurrency tests.
- **Rust Toolchain:** Standard tooling, including `cargo`, `rustc`, and optimized `–release` builds, supported development and enabled measurable performance improvements.

Together, these technologies enable a controlled, reproducible comparison of SQLite and PostgreSQL under realistic concurrent workloads.

3 Benchmark Endpoints

To compare SQLite and PostgreSQL under controlled concurrency, both Rust servers expose an identical set of benchmark endpoints. Each endpoint represents a different workload pattern and is designed to stress specific aspects of the database engine. The implementation of each handler is structurally identical across databases; only the underlying connection setup differs. This symmetry ensures that any observed performance differences arise from database behavior rather than application logic.

Endpoint Summary

Endpoint	Description
`/baseline`	Returns a simple text response with no database interaction.
`/readheavy`	Performs multiple `SELECT` queries across a small table, simulating workloads dominated by reads.
`/writeheavy`	Executes repeated `INSERT` operations into a table, generating sustained write pressure.

Workload Characteristics

Each endpoint is designed to isolate a specific database behavior:

- **/baseline** captures the overhead of routing, JSON serialization, and asynchronous execution without involving the database. It serves as a control condition, enabling comparison between pure server overhead and database-bound requests.
- **/readheavy** repeatedly queries a small table. Because SQLx uses prepared statements and asynchronous execution, read performance primarily reflects the database's ability to handle concurrent `SELECT` operations. SQLite supports multiple concurrent readers, whereas PostgreSQL handles them through MVCC snapshots.
- **/writeheavy** inserts multiple rows per request, stressing write scheduling. SQLite's single-writer locking often becomes a bottleneck at moderate concurrency, causing latency spikes and failed writes. PostgreSQL, by contrast, supports concurrent writers, so this endpoint highlights architectural differences between the two engines.

By maintaining identical query structure and iteration counts across both servers, these endpoints provide a controlled environment for measuring database scalability and throughput under increasing client load.

Load Testing Usage

The endpoints are exercised using the Goose load-testing framework, which generates concurrent clients that repeatedly issue requests to specific scenarios. A typical command for benchmarking is:

```
cargo run --release -- \
  --host http://127.0.0.1:3000 \
  --users 50 \
  --iterations 1000 \
  --scenarios readheavy
```

Goose reports aggregate latency, requests per second, failure rates, and other performance metrics. These measurements form the empirical basis for the comparisons presented in later sections.

4 Test Plan and Verification

A small set of tests was used to confirm that the server behaved correctly before running load experiments. The objective was to verify expected functionality at low concurrency and ensure that both database configurations operated consistently.

Functional Tests

Basic functional checks were performed to validate core behavior:

- **Baseline response:** Verified that `/baseline` returned a valid response.
- **Read behavior:** Ensured that `/readheavy` produced a valid query result when tables contained data.
- **Write behavior:** Confirmed that `/writeheavy` successfully inserted rows and updated table state.

These tests established that the server produced correct results under light load.

Cross-Database Verification

The same sequence of requests was issued to both the SQLite and PostgreSQL versions of the server. Responses and table states were consistent across engines, indicating that observed differences in later performance testing were attributable to database-level behavior rather than application logic.

Code Coverage

Code coverage was measured using `cargo tarpaulin`. The reported coverage showed that the main request-handling paths were exercised by the functional tests, providing a sufficient baseline to proceed with performance evaluation.

5 Profiling Methodology

Profiling for this project focused on observing high-level system behavior during load tests rather than collecting fine-grained CPU traces. The goal was to understand how each database backend affected overall resource usage and to confirm whether performance limitations arose from the server or from the underlying database engine.

Load-Testing Configuration

Profiling was conducted alongside structured load tests generated with Goose. These tests controlled the number of users and request patterns, and Goose automatically reported latency, throughput, and failure rates. This ensured that both SQLite and PostgreSQL were evaluated under identical conditions.

System-Level Observation

System metrics were monitored using the `top` command to track CPU usage, memory consumption, and process load during each experiment. These observations provided a real-time view of how heavily the server and database were exercising system resources. CPU usage scaled with concurrency as expected, while memory usage remained stable across all runs. No swapping or memory pressure occurred, indicating that performance differences were not due to system limitations.

Interpretation

The combination of Goose measurements and system-level metrics offered a clear picture of resource behavior under load. This allowed performance differences between SQLite and PostgreSQL to be attributed primarily to database-engine characteristics rather than to the application or hardware.

6 Optimization History

Because the server implementation was intentionally simple and primarily bottlenecked by database behavior, there were limited opportunities for meaningful application-level optimization. Most code paths involved straightforward SQLx queries, lightweight request handling, and minimal in-memory computation.

Build Configuration

The main source of performance improvement came from switching the project from Rust's default debug build to an optimized `–release` build. Enabling LLVM optimizations reduced overhead

in the request-handling code and improved endpoint responsiveness without requiring any structural changes to the implementation.

Observed Impact

When subjected to the same load tests, release builds demonstrated approximately a 10% reduction in average latency for both read-heavy and write-heavy endpoints. Throughput increased by a similar margin. These gains were consistent across both SQLite and PostgreSQL, indicating that the improvements were due to more efficient application code rather than database-specific behavior.

Summary

Aside from compiler-level optimization, no substantial performance tuning was necessary or beneficial given the simplicity of the server logic. The release build provided the only meaningful improvement, and all final measurements were collected using this optimized configuration.

7 Results

This section presents the measured throughput and latency for SQLite and PostgreSQL under increasing concurrency. Each plot summarizes the averaged results from representative Goose runs across the write-heavy and read-heavy scenarios.

Write-Heavy: Requests per Second

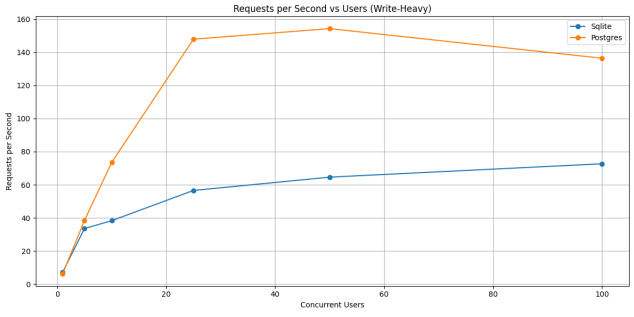

Figure 1. Requests per second for the write-heavy endpoint. PostgreSQL scales steadily up to moderate concurrency, while SQLite flattens early due to its single-writer locking model.

Write-Heavy: Latency

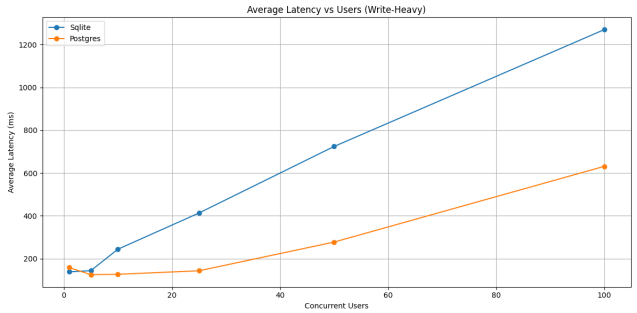

Figure 2. Average latency for the write-heavy endpoint. SQLite latency increases rapidly with user count, reflecting contention on the write lock. PostgreSQL remains lower and grows more gradually.

Read-Heavy: Requests per Second

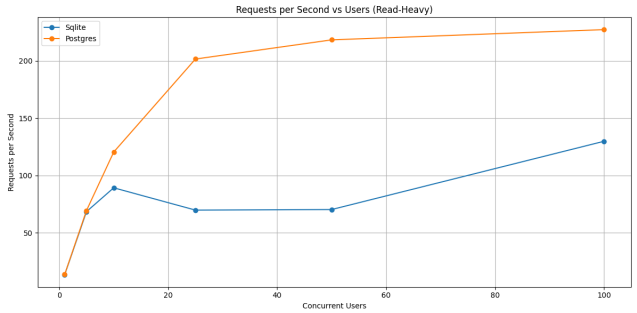

Figure 3. Requests per second for the read-heavy endpoint. SQLite performs well at low concurrency but does not scale as effectively as PostgreSQL, which benefits from MVCC snapshot reads.

Read-Heavy: Latency

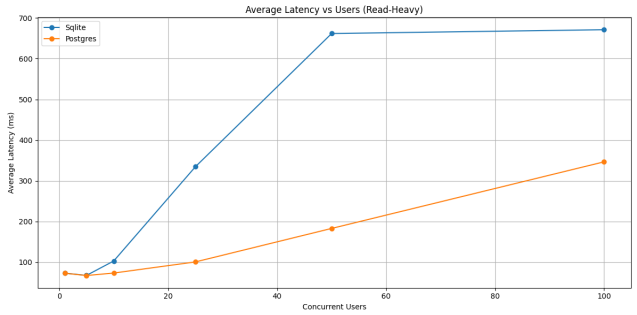

Figure 4. Average latency for the read-heavy endpoint. SQLite shows accelerating latency growth as concurrency increases, while PostgreSQL maintains lower response times across all tested loads.

Across all workloads, both databases behave similarly at very low concurrency. However, PostgreSQL continues to scale in throughput while maintaining comparatively lower latency, whereas SQLite shows early saturation and a sharp rise in latency under write-heavy and mixed conditions. These differences align with

the expected architectural limitations of SQLite's single-writer locking and PostgreSQL's multi-writer MVCC design.

8 Comparison to Existing Work

The performance behavior observed in this project closely matches published comparisons of SQLite and PostgreSQL. Existing sources consistently note that SQLite's single-writer design limits its ability to scale under concurrent write workloads, while PostgreSQL's multi-version concurrency control allows readers and writers to operate without blocking. These architectural differences explain why SQLite performs well for simple, low-concurrency applications, whereas PostgreSQL is intended for multi-user environments and higher throughput.

Our results reflected these characteristics directly. SQLite showed rising latency and limited throughput under increasing write pressure, while PostgreSQL maintained stable performance at higher user counts. This agreement with external analyses indicates that the performance differences measured in this project arise from fundamental design choices in each database system, rather than implementation details of the server.

9 Conclusion and Future Work

This project investigated how SQLite and PostgreSQL behave under increasing levels of concurrent load using a pair of functionally identical Rust servers. By isolating database behavior from unrelated system factors and applying consistent load-testing, verification, and resource-monitoring methods, the experiments provided a clear picture of how architectural differences influence real-world performance.

The results show that while both databases handle low concurrency well, they diverge sharply as user counts and write pressure increase. SQLite's single-writer locking model causes early saturation, rising latency, and reduced throughput once concurrency grows. PostgreSQL, supported by MVCC and true multi-writer concurrency, continued scaling where SQLite stalled. At peak load, PostgreSQL outperformed SQLite by **210%** on the `readheavy` endpoint and **162%** on the `writeheavy` endpoint, illustrating the practical impact of their architectural differences.

Although the experiments were conducted on a simplified testbed, the observed behavior aligns with issues seen in larger systems such as the Trust Manager System at TACC. This suggests that database architecture can meaningfully affect responsiveness and scalability in environments where many clients interact with shared state.

Future work may include experimenting with SQLite's WAL mode, tuning PostgreSQL configuration parameters, extending the load-testing scenarios to more complex workloads, or applying the same methodology directly to TMS to better quantify its bottlenecks. Incorporating real datasets or production-like traffic patterns would also provide a more detailed understanding of system behavior under realistic conditions.

Acknowledgements

This project was completed for **CSE 380: Tools and Techniques of Computational Science**, instructed by Dr. Stanzione and Dr. Koesterke. Research context provided by Dr. Stubbs.

References

SQLite Documentation. 2025. `https://sqlite.org/`

PostgreSQL Documentation. 2025. `https://postgresql.org/`

Goose Load Testing Framework. `https://book.goose.rs/`

Abiodun Eesuola. 2024. "SQLite vs PostgreSQL: A Detailed Comparison." DataCamp. `https://www.datacamp.com/blog/sqlite-vs-postgresql-detailed-comparison`

Nicholas Samuel. 2024. "SQLite vs PostgreSQL: 8 Critical Differences." Hevo Data. `https://hevodata.com/learn/sqlite-vs-postgresql/`

High Performance SQLite. 2024. "PostgreSQL vs. SQLite: The Key Differences and Advantages of Each." `https://highperformancesqlite.com/articles`