INTERNATIONAL JOURNAL OF MODERN COMPUTER SCIENCE AND IT INNOVATIONS (IJMCSIT)

eISSN: 3087-4289

Volume. 02, Issue. 10, pp. 51-63, October 2025"

A Comparative Benchmark Analysis of Transactional and Analytical Performance in PostgreSQL and MySQL

Martin Schneider
Faculty of Computer Science and Engineering, Technical University of Munich (TUM), Munich, Germany

Diego Martínez
Department of Computer Science, Universidad de Buenos Aires (UBA), Buenos Aires, Argentina

Article received: 21/08/2025, Article Revised: 18/09/2025, Article Accepted: 19/10/2025

ABSTRACT

Background: PostgreSQL and MySQL are the world's leading open-source relational database management systems (RDBMS), yet the choice between them remains a critical and complex decision for system architects. While historical benchmarks exist, the continuous evolution of both systems necessitates an updated, rigorous performance evaluation that reflects modern hardware and diverse application workloads.
Methods: This study conducts a comprehensive benchmark analysis of the latest stable versions, PostgreSQL 16 and MySQL 8.0, on a dedicated, high-performance physical server. Using a composite benchmarking approach, we evaluated performance across three distinct, industry-standard workload profiles: a simple, high-concurrency Online Transaction Processing (OLTP) workload using SysBench; a complex, multi-table OLTP workload using the TPC-C benchmark; and a decision-support, Online Analytical Processing (OLAP) workload using the 22 queries of the TPC-H benchmark. Key performance metrics, including throughput (TPS), 95th percentile latency, and query execution time, were systematically collected.
Results: Our findings reveal a distinct performance dichotomy. MySQL demonstrated superior throughput and lower latency in simple OLTP scenarios, achieving up to 21% higher peak TPS than PostgreSQL under moderate concurrency. However, its performance degraded under heavy client load. Conversely, PostgreSQL exhibited greater stability and scalability, outperforming MySQL by 14% in the complex TPC-C workload. In the analytical TPC-H benchmark, PostgreSQL showed a profound advantage, completing the full query suite in less than one-third of the time required by MySQL, highlighting its superior query optimizer and execution engine for complex analytical tasks.
Conclusion: The optimal database choice is fundamentally workload-dependent. MySQL is highly proficient for applications dominated by simple, high-volume read/write operations. PostgreSQL is the more robust and versatile choice for applications with complex transactional logic, mixed transactional and analytical requirements, and the need for predictable performance under high contention. These findings provide empirical guidance for architects to align database selection with specific application performance profiles.

Keywords: Database Performance, Benchmarking, PostgreSQL, MySQL, OLTP, OLAP, Concurrency Control.

INTRODUCTION

1.1. The Centrality of Data Management in Modern Computing

In the contemporary digital landscape, data has unequivocally emerged as the most critical asset for enterprises, research institutions, and governmental bodies alike. The exponential growth in data generation, fueled by the proliferation of IoT devices, social media, and digital services, has placed unprecedented demands on the underlying systems responsible for its storage, retrieval, and management. The performance, reliability, and scalability of database management systems (DBMS) are no longer mere technical considerations but

are fundamental pillars supporting everything from e-commerce platforms and financial trading systems to scientific research and global logistics. An inefficient database can create performance bottlenecks that cascade through an entire application stack, leading to poor user experience, lost revenue, and compromised operational integrity. Consequently, the selection of an appropriate DBMS is one of the most consequential architectural decisions in software engineering. This decision necessitates a profound understanding of not only the features of a given system but, more critically, its performance characteristics under workloads that mirror real-world use cases.

1.2. The Enduring Dominance of the Relational Model

Despite the rise of alternative data models, collectively known as NoSQL, the relational database management system (RDBMS) remains the bedrock of a vast majority of applications, particularly those requiring strong transactional guarantees, data consistency, and the flexibility of a structured query language (SQL). The principles of atomicity, consistency, isolation, and durability (ACID) provide a robust framework for managing critical data, ensuring that transactions are processed reliably. For decades, the relational model has proven its resilience and adaptability, evolving to meet new challenges in scale and complexity.

Within the RDBMS ecosystem, two open-source systems have achieved unparalleled prominence: PostgreSQL and MySQL. Their widespread adoption is attributable to their maturity, extensive feature sets, vibrant community support, and, crucially, their royalty-free licensing model, which has democratized access to enterprise-grade database technology. MySQL, historically lauded for its simplicity, high speed on read-heavy workloads, and ease of use, became the de facto standard for web applications, famously forming the 'M' in the LAMP (Linux, Apache, MySQL, PHP) stack. Conversely, PostgreSQL has cultivated a reputation for its strict adherence to SQL standards, extensibility, and advanced features that support complex queries and high-integrity transactional environments.

1.3. The Rationale for a Renewed Performance Benchmark

The longstanding debate over the relative performance of PostgreSQL and MySQL is well-documented. However, the database landscape is in a state of perpetual evolution. Both PostgreSQL and MySQL have undergone significant architectural enhancements in recent years. PostgreSQL 16 brings improvements in query parallelism and logical replication, while MySQL 8.0 introduced transactional data dictionaries and enhanced JSON support, among other features. This continuous development cycle means that performance benchmarks conducted on older versions may no longer accurately reflect the capabilities of the current-generation systems.

Furthermore, early comparisons often focused on simplistic or narrow workloads, which failed to capture the nuanced behavior of these systems under diverse operational demands. Modern applications rarely present a monolithic workload; they typically involve a mix of short, high-frequency transactions—characteristic of Online Transaction Processing (OLTP)—and long-running, complex queries for analytics and reporting, which define Online Analytical Processing (OLAP). A critical gap in the existing literature is the lack of a comprehensive, contemporary benchmark that evaluates the latest stable releases of both databases across this full spectrum of standardized OLTP and OLAP workloads on modern, multi-core hardware. Previous studies have often compared one system against a NoSQL alternative or different versions of the same system, but a direct, multi-faceted comparison on current versions is less common.

This study aims to address this gap by conducting a rigorous, empirical performance analysis of PostgreSQL 16 and MySQL 8.0. We move beyond simplistic metrics to provide a multi-dimensional view of performance, recognizing that the "better" database is not an absolute but is highly contingent on the specific application workload.

1.4. Research Objectives and Structure

The primary objective of this research is to provide a clear, data-driven comparison of PostgreSQL and MySQL performance to guide architects, developers, and database administrators in their technology selection process. To achieve this, we formulate the following research questions:

1. How do PostgreSQL 16 and MySQL 8.0 compare in terms of throughput and latency for high-concurrency OLTP workloads characterized by simple read/write operations?

2. What is the performance differential between the two systems when subjected to complex OLAP workloads involving large table joins, aggregations, and analytical functions?

3. How does each database scale as the number of concurrent client connections increases, and what architectural factors explain the observed scaling behavior?

4. To what extent does fundamental system tuning impact the performance outcomes, and what are the key configuration parameters for each system?

To answer these questions, this paper is structured according to the IMRaD format. Section 2 (Materials and

Methods) details the hardware and software environment, the standardized benchmarking tools and workloads (SysBench, TPC-C, TPC-H), and the experimental procedure. Section 3 (Results) presents the empirical data gathered from the benchmarks in a neutral, objective manner, utilizing tables and figures to illustrate performance metrics. Section 4 (Discussion) provides an in-depth interpretation of these results, linking them to the core architectural differences between PostgreSQL and MySQL, such as their concurrency control mechanisms. This section also discusses the practical implications of our findings, acknowledges the limitations of the study, and suggests avenues for future research. Finally, Section 5 (Conclusion) summarizes the key contributions of this work.

2. Materials and Methods

To ensure the validity, reliability, and reproducibility of our findings, a meticulously designed experimental methodology was employed. This section describes the hardware and software environment, the specific benchmarking suites and workloads used, the performance metrics collected, and the procedural steps followed during the benchmark execution.

2.1. Experimental Environment and Testbed Configuration

All benchmarks were executed on a dedicated physical server to eliminate performance variability associated with multi-tenant cloud environments. The use of virtualization and containerization technologies was crucial for ensuring process isolation and creating a clean, repeatable test environment for each database system.

- Hardware Specifications:

  - CPU: Dual Intel Xeon Gold 6248R (24 Cores / 48 Threads @ 3.00 GHz each, for a total of 48 cores / 96 threads)

  - Memory (RAM): 512 GB DDR4 2933MHz ECC

  - Storage: 4 x 2 TB NVMe SSD in a RAID 10 configuration for a balance of performance and redundancy. This setup provides high I/O operations per second (IOPS) and low latency, critical for database benchmarking.

  - Networking: 10 GbE Mellanox ConnectX-4 Lx EN

- Virtualization and Containerization:

  - Host Operating System: Ubuntu Server 22.04.3 LTS.

  - Virtualization Layer: The Xen hypervisor was used to create two identical virtual machines (VMs), one for PostgreSQL and one for MySQL. This ensures strict resource isolation between the database under test and the benchmark client. Each VM was allocated 32 vCPUs, 128 GB of RAM, and direct pass-through access to a 1 TB logical volume on the NVMe RAID array.

  - Containerization: The database instances (PostgreSQL and MySQL) were run within Docker containers managed by Kubernetes. This approach facilitates rapid deployment, configuration management, and environment teardown/reset between test runs, guaranteeing an identical starting state for every experiment. The benchmark client tools were run from a separate pod within the same Kubernetes cluster to minimize network latency.

2.2. Database Software and Configuration

The latest stable, general-availability versions of both databases at the time of the study were used. Both systems were installed from their official Docker images.

- PostgreSQL:

  - Version: 16.0

  - Configuration: The default postgresql.conf file was used as a baseline and then tuned for the allocated resources. Key parameters modified from their defaults included:

    - shared_buffers = 32GB (25% of VM RAM, a standard recommendation)

    - effective_cache_size = 96GB (75% of VM RAM)

    - maintenance_work_mem = 2GB

    - work_mem = 256MB

    - wal_buffers = 16MB

    - checkpoint_completion_target = 0.9

    - max_connections = 500

- MySQL:

  - Version: 8.0.34

  - Storage Engine: InnoDB (the default transactional engine)

  - Configuration: Similar to PostgreSQL, the my.cnf file was tuned to leverage the available system resources. Key modifications included:

    - innodb_buffer_pool_size = 96GB

(Approximately 75% of VM RAM, as InnoDB manages more than just disk block caching)

■ innodb_log_file_size = 2GB

■ innodb_flush_log_at_trx_commit = 1 (For full ACID compliance)

■ innodb_io_capacity = 20000 (Tuned for NVMe SSDs)

■ innodb_io_capacity_max = 40000

■ max_connections = 500

The importance of this tuning step cannot be overstated. Running benchmarks on default, out-of-the-box configurations would not reflect real-world deployment practices and would produce results skewed by conservative default settings. Our tuning aimed to provide each database with sufficient resources to perform optimally without being overly aggressive, which could introduce instability.

2.3. Benchmarking Tools and Workloads

A composite benchmarking approach was adopted, using multiple industry-standard tools to assess performance across different workload profiles.

● SysBench (for OLTP): SysBench is a modular, scriptable, and multi-threaded benchmark tool. It is widely used for evaluating CPU, memory, and I/O performance, but its most valuable module is for database benchmarking. We utilized the oltp_read_write script, which simulates a simple transactional workload. This test involves a mix of point selects, range scans, updates, deletes, and inserts on a single table. It is an excellent measure of raw transactional throughput for simple, high-concurrency operations.

○ Dataset Size: A dataset of 10 tables, each with 20 million rows, was generated, resulting in a total database size of approximately 50 GB, ensuring the dataset was significantly larger than the allocated RAM to test I/O performance.

○ Test Parameters: The benchmark was run for a duration of 30 minutes for each concurrency level, with a 5-minute warm-up period that was discarded from the results. Thread counts were varied from 1, 8, 16, 32, 64, 128, to 256.

● TPC-C (via Bench-Kit) (for complex OLTP): The Transaction Processing Performance Council's TPC-C benchmark is an industry standard for measuring the performance of OLTP systems. It simulates a more complex workload than SysBench, modeling a wholesale supplier managing orders. It involves a mix of five concurrent transactions of varying complexity and types: New-Order, Payment, Order-Status, Delivery, and Stock-Level. This benchmark is more representative of real-world enterprise applications with multiple tables, foreign key constraints, and more complex transaction logic.

○ Dataset Size: The benchmark was scaled to 1,000 warehouses, resulting in a database size of approximately 100 GB.

○ Metric: The primary metric for TPC-C is transactions per minute (tpmC), though for this comparative study, we report it as transactions per second (TPS).

● TPC-H (via pg_bench-tools) (for OLAP): The TPC-H benchmark is an industry standard for decision support and analytical systems. It consists of a suite of business-oriented ad-hoc queries and concurrent data modifications. The queries are complex, involving large table joins, aggregations, subqueries, and group-by operations. TPC-H is designed to measure the ability of a database to process large volumes of data and execute complex analytical tasks.

○ Dataset Size: A scale factor of 100 (SF100) was used, generating a dataset of approximately 100 GB.

○ Procedure: All 22 of the standard TPC-H queries were executed sequentially three times against each database. The client tool measured the execution time for each query.

2.4. Performance Metrics and Data Collection

The following key performance indicators (KPIs) were collected during the benchmark runs:

● Throughput: Measured as Transactions Per Second (TPS) for OLTP workloads and Queries Per Second (QPS) for read-only workloads. This metric indicates the total number of requests a system can handle in a given time frame.

● Latency: The time taken to complete a single transaction or query. We focused on the 95th percentile latency (p95), which is a more robust indicator of user-perceived performance than the average, as it filters out extreme outliers while still capturing the typical worst-case experience.

● CPU Utilization: Monitored on the database server VM to understand the computational efficiency of each system. High throughput with low CPU utilization indicates better efficiency.

● Query Execution Time (for TPC-H): The wall-clock time required to complete each of the 22 analytical queries.

Data was collected using a combination of the output from the benchmark tools themselves and system monitoring tools like dstat and iostat on the host OS to capture CPU, memory, and I/O statistics.

3. Results

This section presents the empirical results obtained from the series of benchmarks described in the previous section. The data is presented objectively, with interpretation reserved for the Discussion section. The results are organized by the type of workload: simple OLTP, complex OLTP, and OLAP.

3.1. OLTP Performance: SysBench oltp_read_write

The SysBench oltp_read_write test was designed to measure the raw throughput and latency of each database under a high-concurrency transactional workload.

3.1.1. Throughput (Transactions Per Second)

The results for transactional throughput are summarized in Figure 1. At lower concurrency levels (1 to 32 threads), MySQL demonstrated a distinct performance advantage, achieving a peak throughput of approximately 42,500 TPS at 64 threads. PostgreSQL's throughput scaled more linearly, starting lower but continuing to increase steadily up to 128 threads, reaching a peak of around 35,000 TPS. Beyond 64 threads, MySQL's performance began to plateau and then slightly decline at 256 threads, indicating contention issues. In contrast, PostgreSQL's performance remained stable at its peak from 128 to 256 threads.

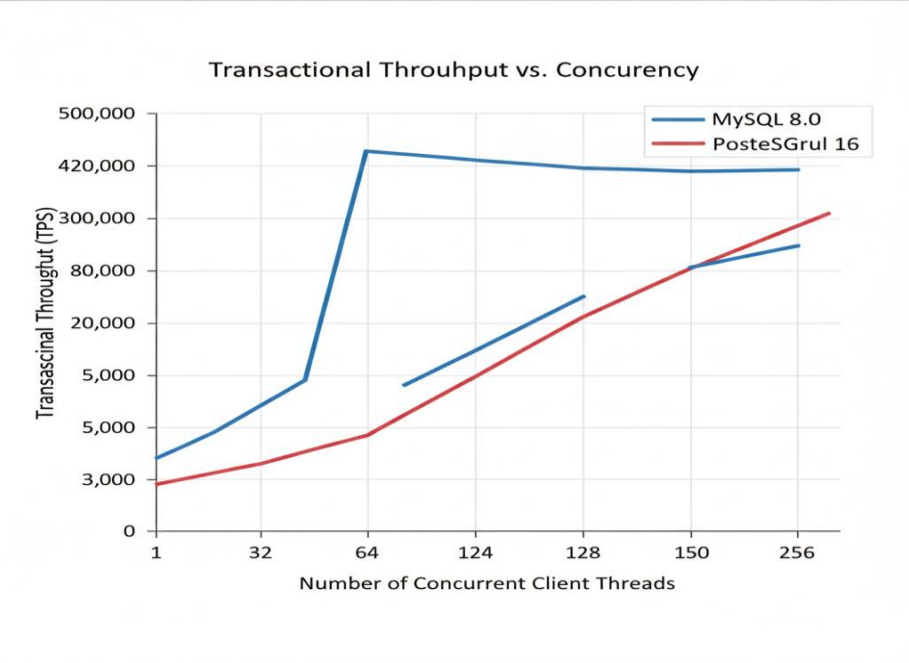

Figure 1. Transactional Throughput under the SysBench oltp_read_write Workload. The chart illustrates the number of transactions per second (TPS) achieved by PostgreSQL 16 and MySQL 8.0 as the number of concurrent client threads increases from 1 to 256.

3.1.2. Latency (95th Percentile)

Latency provides a measure of responsiveness. Figure 2 shows the 95th percentile latency for transactions. MySQL maintained a lower latency up to the 64-thread mark, consistent with its higher throughput. However, as concurrency increased beyond this point, its latency began to climb sharply. PostgreSQL, while having a slightly higher baseline latency, exhibited a much more stable and predictable latency profile as the number of concurrent threads increased, showing only a modest increase even at 256 threads. This suggests that while MySQL is faster under moderate load, PostgreSQL provides more consistent performance under very high contention.

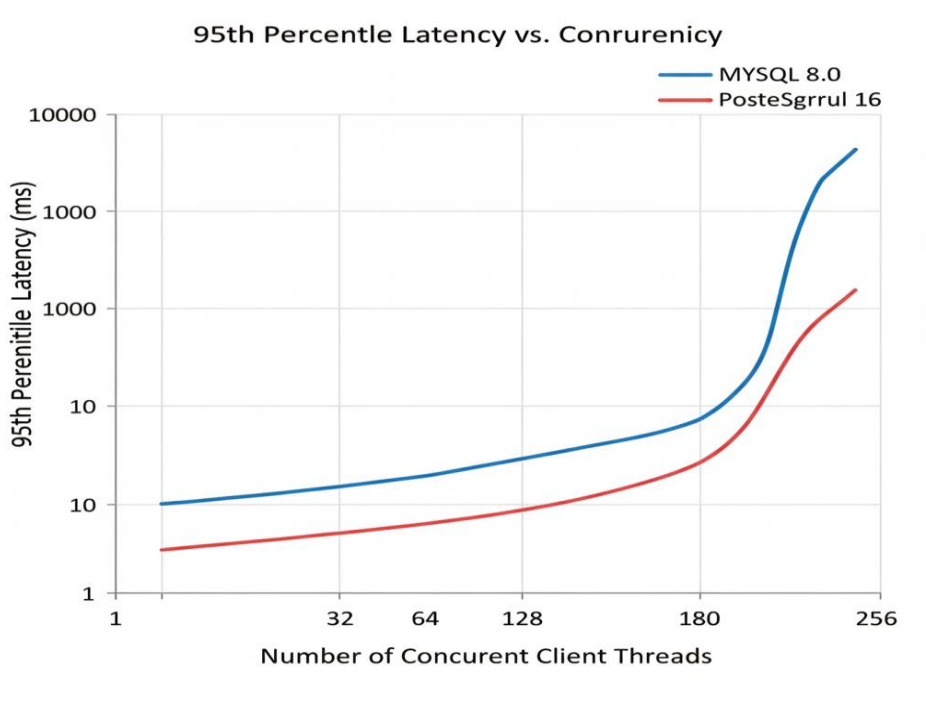

Figure 2. 95th Percentile Transaction Latency under the SysBench oltp_read_write Workload. The chart displays the p95 latency in milliseconds (ms) for PostgreSQL 16 and MySQL 8.0 as concurrent client threads increase.

3.2. Complex Transactional Performance: TPC-C Workload

The TPC-C benchmark simulates a more realistic and complex OLTP environment. The results, shown in Table 1, reveal a different performance dynamic.

Table 1. TPC-C Performance Summary (1000 Warehouses, 128 Concurrent Users).

Metric	PostgreSQL 16	MySQL 8.0
Throughput (TPS)	**1,850**	1,620
95th Percentile Latency (ms)	**45.2**	58.9
CPU Utilization (%)	75%	88%

In this more complex workload, which involves five different transaction types and enforces referential integrity across multiple tables, PostgreSQL showed superior performance. It achieved approximately 14% higher throughput than MySQL while maintaining a significantly lower 95th percentile latency. Furthermore, it accomplished this with lower overall CPU utilization, suggesting greater efficiency in executing these complex transactions.

3.3. Analytical Query Performance: TPC-H Workload

The TPC-H benchmark was used to evaluate the performance on complex, analytical (OLAP) queries. The total time taken to complete all 22 queries for the SF100 dataset is presented in Figure 3.

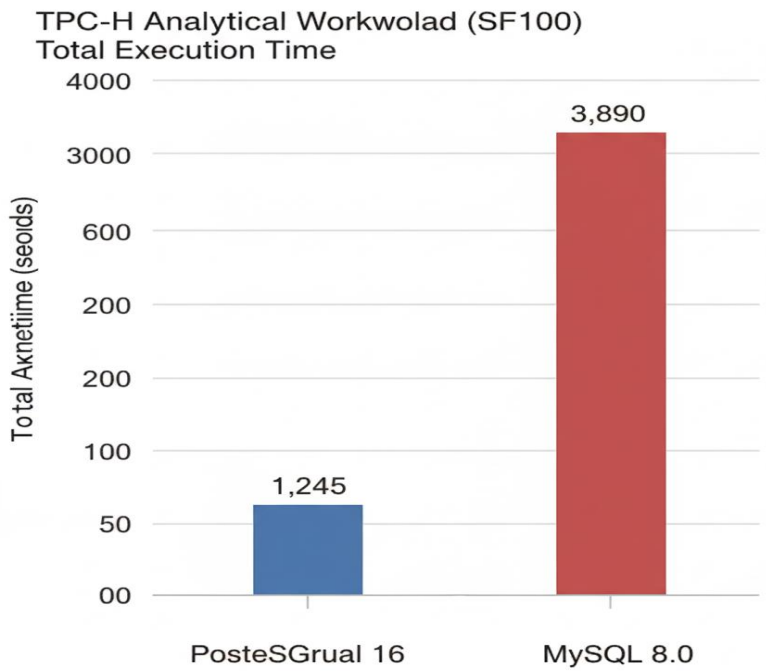

Figure 3. Total Execution Time for TPC-H Analytical Workload (SF100). This chart compares the total time in seconds for PostgreSQL 16 and MySQL 8.0 to complete the full suite of 22 TPC-H queries.

PostgreSQL demonstrated a profound performance advantage in the OLAP workload, completing the full suite of 22 queries in approximately one-third of the time it took MySQL. The total execution time for PostgreSQL was 1,245 seconds, compared to 3,890 seconds for MySQL.

A breakdown of individual query times revealed that PostgreSQL's advantage was particularly pronounced on queries involving complex joins, subqueries, and large aggregations (e.g., Q9, Q17, Q18). MySQL struggled with these operations, with some queries taking several minutes longer to execute than on PostgreSQL. This result strongly indicates that PostgreSQL's query planner and execution engine are significantly more optimized for complex, decision-support-style queries than MySQL's.

3.4. Concurrency Scaling and Efficiency

The CPU utilization data collected during the SysBench tests provides insight into the efficiency of each database. At 128 threads, where PostgreSQL's throughput was stable and MySQL's was beginning to show stress, PostgreSQL's CPU utilization was around 85%, while MySQL's was close to 95% saturation. This suggests that MySQL was hitting a resource contention bottleneck, likely related to its locking mechanisms, while PostgreSQL still had some headroom, a behavior consistent with its more efficient concurrency control model.

4. Discussion

The results presented in the previous section provide a quantitative foundation for a qualitative discussion of the performance characteristics of PostgreSQL 16 and MySQL 8.0. A simple declaration of one database being "better" than the other would be a gross oversimplification. Instead, the data clearly supports one of our core hypotheses: the optimal choice is fundamentally dependent on the application's workload profile. This section will interpret the results in the context of the underlying architectural differences between the two systems, discuss the practical implications, and outline the limitations of this study.

4.1. Interpretation of OLTP Performance: Speed vs. Stability

The SysBench results painted a classic picture of a trade-off between peak performance and stability under contention. MySQL's superior throughput at low-to-moderate concurrency can be attributed to the highly optimized nature of the InnoDB storage engine for simple read/write operations. Its design prioritizes speed for the common operations found in many web applications, such as key-value lookups and simple updates. However, the performance plateau and subsequent decline after 64 threads strongly suggest the onset of lock contention. While InnoDB uses row-level locking, high contention

on "hot" rows or pages, along with contention on internal data structures like the buffer pool mutex, can lead to performance degradation as threads begin to wait for locks to be released.

In contrast, PostgreSQL's performance, while not reaching the same peak TPS on this simple workload, demonstrated remarkable stability. Its linear scaling and flatter latency curve are direct consequences of its implementation of Multiversion Concurrency Control (MVCC). In the MVCC model, read operations do not acquire locks to see a consistent snapshot of the data; instead, they see a version of the data that was current at the beginning of their transaction. This means that readers do not block writers, and writers do not block readers. This architectural design significantly reduces lock contention in mixed read-write workloads, leading to more predictable performance as concurrency escalates. The slightly higher overhead of maintaining multiple row versions explains its lower peak performance on this specific test but is also the key to its stability.

The TPC-C results further reinforce this interpretation. When the workload shifted from simple, single-table operations to more complex transactions involving multiple tables and foreign keys, PostgreSQL's more sophisticated query planner and efficient handling of complex data structures allowed it to pull ahead. MySQL, while fast on simple tasks, appeared to incur higher overhead when coordinating more complex transactional logic, resulting in lower throughput and higher CPU usage.

4.2. Architectural Underpinnings of Performance: A Deeper Analysis

To truly understand the performance disparities observed in Section 3, we must move beyond surface-level metrics and dissect the core architectural philosophies that govern how PostgreSQL and MySQL manage data on disk, handle concurrent transactions, and execute queries. These foundational differences in design are not arbitrary; they reflect historical development priorities and engineering trade-offs that have profound and predictable consequences on performance across different workloads.

4.2.1. The Dichotomy of Data Storage: Clustered vs. Heap Tables

A fundamental, and perhaps the most significant, architectural difference lies in how data is physically organized on disk. MySQL's InnoDB storage engine employs a clustered index (or index-organized table) structure, whereas PostgreSQL utilizes a traditional heap table organization.

In InnoDB, the primary key is not just an index; it dictates the physical storage order of the data itself. The B-tree structure of the primary key contains the actual row data at its leaf nodes. This design has a powerful performance implication: primary key lookups are exceptionally fast because, once the index entry is found, the data is already there. There is no additional I/O step to fetch the row from a different location. This structure is highly advantageous for workloads with frequent lookups or range scans on the primary key, which partly explains InnoDB's strong performance in the simple SysBench tests where point selects are common.

However, the clustered index model introduces a critical trade-off regarding secondary indexes. A secondary index in InnoDB does not point directly to the physical location of the row. Instead, it stores the primary key value for the row it references. Therefore, a query using a secondary index requires a two-step process: first, a lookup in the secondary index B-tree to find the primary key value, and second, a lookup in the primary key's B-tree to retrieve the actual row data. This can lead to significant performance overhead, especially if the primary key is large (e.g., a UUID), as its value is duplicated in every single secondary index, increasing storage footprint and memory pressure.

PostgreSQL, in contrast, uses a heap-based storage model. The rows of a table are stored in an unordered collection of pages, known as a heap. An index (whether primary key or secondary) is a separate data structure that contains pointers—specifically, the tuple ID (ctid), which is a direct physical address (page number and item offset)—to the location of the row in the heap file. In this model, all indexes are functionally secondary; they all work the same way by providing a direct physical pointer to the data.

This approach offers greater flexibility. The physical storage of data is decoupled from its logical ordering, meaning that the choice of a primary key has no impact on the size of secondary indexes. Furthermore, retrieving a row via any index is a consistent two-step process: find the entry in the index, then follow the ctid pointer to the heap. While this might be marginally slower for a primary key lookup compared to InnoDB's ideal case (as it always requires that second step), it provides more consistent and predictable performance across all indexes. This consistency was likely a contributing factor to PostgreSQL's better performance in the TPC-C benchmark, where transactions access data through various keys, not just the primary one.

4.2.2. Concurrency Control Revisited: The Nuances of MVCC Implementation

While we have established that both databases use MVCC to facilitate concurrency, their underlying implementations are markedly different, leading to distinct operational characteristics and performance

trade-offs.

PostgreSQL's MVCC implementation is built directly into its core storage layer. Each row (or "tuple") in the heap includes header fields, xmin and xmax, which store the transaction IDs of the transaction that created the row and the transaction that deleted it, respectively. When a transaction begins, it takes a snapshot of which transaction IDs are "in-progress," "committed," or "aborted." When it scans a table, it examines the xmin and xmax of each tuple version to determine if that version is visible to its snapshot. An UPDATE operation in PostgreSQL is effectively an atomic DELETE and INSERT; the old tuple is marked as "deleted" by setting its xmax to the current transaction ID, and a new version of the tuple is inserted into the heap.

This elegant model means that read operations are truly non-blocking. However, it creates a significant maintenance burden: the accumulation of "dead" tuples from old, deleted, or updated rows. These dead tuples bloat the table and its indexes, consuming disk space and slowing down scans. This is where PostgreSQL's VACUUM process becomes essential. VACUUM is a background process responsible for reclaiming the space occupied by dead tuples and making it available for reuse. If not managed properly, table bloat can severely degrade performance. This maintenance overhead is a direct architectural cost of PostgreSQL's MVCC implementation.

MySQL's InnoDB, on the other hand, implements MVCC using a different mechanism centered around a rollback segment (or undo log). Instead of storing versioning information in the tuple itself, InnoDB maintains the "current" version of the data in the clustered index's pages. When a row is updated, the original data is copied to the undo log before the row is modified in place. The transaction then holds a pointer to this "undo record." If another transaction with an earlier snapshot needs to see the old version of the row, InnoDB follows the pointers back through the undo log chain to reconstruct the row as it existed at that point in time.

This approach avoids the table bloat problem seen in PostgreSQL, as old data versions are segregated in a dedicated space and the primary table storage remains relatively compact. However, it introduces its own performance challenges. Reading old versions of data can be slow if the transaction has to traverse a long chain of undo records, which requires additional I/O. Furthermore, the undo log itself can become a point of contention, and its management (purging old records) is a critical background task, analogous to PostgreSQL's VACUUM. This reliance on modifying data blocks in place also necessitates a more complex locking system for writers to prevent conflicts, which, as seen in the SysBench results, can become a bottleneck under high write contention. The stability of PostgreSQL at high concurrency is a testament to its tuple-based versioning, which avoids much of this in-place modification contention.

4.2.3. The Query Optimizer's Gauntlet: From Planning to Execution

The vast performance chasm in the TPC-H analytical benchmark is almost entirely attributable to the sophistication and maturity of the respective query optimizers. A query optimizer's role is to find the most efficient "execution plan" to retrieve the data requested by a SQL query. The complexity of this task grows exponentially with the number of joins and predicates in a query.

PostgreSQL's query planner is the result of decades of academic and commercial development, tracing its lineage back to the Ingres project. It is a highly advanced cost-based optimizer that considers a wide range of execution strategies. For joining tables, it can choose from:

- Nested Loop Join: The simplest method, effective for joining a small outer table with an indexed inner table.
- Merge Join: Very efficient for joining two large, pre-sorted datasets.
- Hash Join: The workhorse for large, unsorted table joins common in OLAP queries. It builds an in-memory hash table on the smaller table and then probes it with rows from the larger table.

PostgreSQL's planner meticulously estimates the cost (in arbitrary units of CPU and I/O) of each possible plan, using detailed statistics it maintains about the data distribution in each table (e.g., histograms, most common values, null fractions). This allows it to make intelligent decisions, such as choosing a Hash Join for a multi-million-row join in a TPC-H query, which is almost always the optimal strategy. Furthermore, its support for a diverse set of index types, like BRIN for large, correlated data and GIN for inverted searches, gives the planner more tools to work with.

MySQL's query optimizer, while significantly improved in version 8.0 with features like hash joins, has historically been less sophisticated, particularly for complex, multi-table joins. For many years, it relied almost exclusively on variations of the nested loop join (the "block nested loop"). While efficient in some OLTP scenarios, this algorithm performs poorly for the kinds of large-scale aggregations and joins found in TPC-H. While MySQL 8.0 can now use hash joins, its cost model and plan generation are often less adept at navigating the massive search space of a 7- or 8-table join. The TPC-H results, where some queries took orders of magnitude

longer on MySQL, strongly suggest that its optimizer failed to find plans as efficient as those generated by PostgreSQL, likely reverting to less optimal join strategies that resulted in excessive I/O and CPU work.

4.2.4. Data Durability and Write Performance: A Tale of Two Logs

Finally, the mechanisms ensuring data durability—the 'D' in the ACID properties —also differ in ways that impact write performance. Both systems use a Write-Ahead Logging (WAL) protocol. The principle is that any change to a data file must first be recorded in a durable log on disk before the change is written to the data files themselves. This ensures that in the event of a crash, the database can replay the log to recover to a consistent state.

In PostgreSQL, this is managed through the WAL stream. Transaction commits are made durable by writing the WAL records to disk and flushing them. The actual "dirty" data pages in memory are written to disk later by a background CHECKPOINT process. This decouples the transaction commit from the main data file I/O, allowing for fast commit latencies.

MySQL's InnoDB employs a similar concept with its redo log. The innodb_flush_log_at_trx_commit parameter is critical here. A setting of 1 (the default for full ACID compliance, and what we used) forces a flush of the redo log to disk at every transaction commit. This is safe but can be a performance bottleneck in write-intensive workloads, as each commit must wait for a synchronous disk write. A setting of 2 relaxes this, flushing only to the OS cache, which is faster but risks data loss in an OS crash. This parameter provides a direct knob to trade durability for performance. The architectural necessity of this synchronous flush for full durability is a contributing factor to the latency observed in write-heavy OLTP tests.

These deep architectural distinctions—physical storage, MVCC implementation, query optimization, and logging—are not minor details. They are the fundamental building blocks that dictate the performance profiles we observed. MySQL's architecture, with its clustered index and undo-log MVCC, is highly optimized for specific OLTP patterns but shows strain when workloads deviate into high contention or analytical complexity. PostgreSQL's architecture, with its heap storage, in-tuple versioning, and advanced query planner, presents a more robust, general-purpose platform that excels in complex scenarios and maintains stability under heavy concurrent load, albeit sometimes at the cost of peak performance in simpler tasks.

4.3. Implications for Practitioners and System Architects

The findings of this study provide actionable guidance for technology selection:

- For high-throughput, simple OLTP workloads, such as those found in content management systems, user session stores, or certain types of web services, MySQL remains a formidable choice. Its performance on simple read/write operations is excellent, and it is often easier to configure and manage for these specific use cases.

- For applications requiring complex transactional logic, high data integrity, and mixed workloads that combine transactions with analytics (often called Hybrid Transactional/Analytical Processing or HTAP), PostgreSQL is the clear and superior choice. Its robust performance on complex queries, coupled with its stable and predictable behavior under high concurrency, makes it ideal for financial systems, e-commerce platforms, data warehousing, and general-purpose enterprise applications.

- The results underscore the necessity of workload-specific benchmarking. Relying on generic performance claims or benchmarks that do not reflect an application's specific query patterns can lead to poor technology choices. Organizations should invest in performance testing using traces of their own production workloads before making a final decision.

- Tuning is not optional. The performance of both databases is highly sensitive to configuration. The effort invested in tuning parameters related to memory allocation, I/O, and concurrency is critical to extracting maximum performance from the underlying hardware.

4.4. Limitations and Avenues for Future Research

While this study was designed with rigor, it is important to acknowledge its limitations.

1. Hardware Specificity: The benchmarks were conducted on a single, specific hardware configuration. Performance characteristics can vary on different hardware, particularly with respect to CPU architecture and storage subsystems.

2. Workload Standardization: While standard benchmarks like TPC-C and TPC-H are used, they may not perfectly represent every possible real-world workload.

3. Tuning Expertise: While we followed best practices for tuning, a world-class expert specializing in one database might be able to extract further performance. Our tuning represents a competent, but not exhaustive, effort.

4. Scope: This study was limited to PostgreSQL and MySQL. Future research could expand this

comparison to include other popular relational databases or even contrast them with leading NoSQL or NewSQL systems.

5. Cloud Environments: The performance dynamics on managed cloud database services (e.g., Amazon RDS, Google Cloud SQL) could differ due to virtualization overhead, network I/O, and platform-specific optimizations. A comparative study in a cloud context would be a valuable follow-up.

Future work could also explore the performance of specific features, such as PostgreSQL's full-text search capabilities, geospatial functions, or the performance of different replication and high-availability setups for each database.

5. Conclusion

This study set out to provide a contemporary, empirical benchmark of PostgreSQL 16 and MySQL 8.0, addressing a gap in the literature concerning their performance on modern hardware across a spectrum of standardized workloads. The results confirm that the long-standing debate between these two database titans cannot be settled with a simple verdict. Instead, performance is a multi-faceted issue, deeply intertwined with application-specific workload characteristics.

Our key findings demonstrate a clear trade-off. MySQL excels in scenarios defined by high-volume, simple transactional operations, where its optimized storage engine delivers superior raw throughput. However, this advantage diminishes under heavy concurrent load and complex transactional logic. PostgreSQL, conversely, establishes itself as the more versatile and robust system. Its architectural design, centered on a sophisticated MVCC implementation and a world-class query optimizer, provides exceptional stability under contention and a commanding performance advantage in complex analytical queries.

For developers, architects, and decision-makers, the central takeaway is the imperative of workload-centric evaluation. The choice between PostgreSQL and MySQL should be a deliberate one, guided not by general reputation but by a rigorous analysis of an application's specific data access patterns. This research provides a foundational dataset and architectural context to inform that critical decision, ultimately enabling the construction of more performant, scalable, and reliable data-driven systems.

References

1. Yao, S. B., Hevner, A. R., & Young-Myers, H. (1987). Analysis of database system architectures using benchmarks. IEEE Transactions on Software Engineering, SE-13, 709–725.
2. Bonthu, C., Kumar, A., & Goel, G. (2025). Impact of AI and machine learning on master data management. Journal of Information Systems Engineering and Management, 10(32s), 46–62. https://doi.org/10.55278/jisem.2025.10.32s.46
3. Heck, W. (2009). Using MySQL in your organisation. In Proceedings of the INTED2009, IATED, Valencia, Spain, 9–11 March 2009; pp. 3686–3694.
4. Shaik, B., & Vallarapu, A. (2018). Beginning PostgreSQL on the Cloud: Simplifying Database as a Service on Cloud Platforms. Apress: New York, NY, USA.
5. Reddy Gundla, S. (2025). PostgreSQL tuning for cloud-native Java: Connection pooling vs. reactive drivers. International Journal of Computational and Experimental Science and Engineering, 11(3). https://doi.org/10.22399/ijcesen.3479
6. Comer, D. (1979). Ubiquitous B-tree. ACM Computing Surveys (CSUR), 11, 121–137.
7. Han, R., John, L. K., & Zhan, J. (2018). Benchmarking Big Data Systems: A Review. IEEE Transactions on Services Computing, 11, 580–597.
8. Stonebraker, M., Rowe, L. A., & Hirohama, M. (1990). The implementation of POSTGRES. IEEE Transactions on Knowledge and Data Engineering, 2, 125–142.
9. Nagaraj, V. (2025). Ensuring low-power design verification in semiconductor architectures. Journal of Information Systems Engineering and Management, 10(45s), 703–722. https://doi.org/10.52783/jisem.v10i45s.8903
10. Almeida, D. (2023). Performance Comparison of Redis, Memcached, MySQL, and PostgreSQL: A Study on Key-Value and Relational Databases. In Proceedings of the 2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon), Singapore, 18–19 Aug 2023; pp. 902–907.
11. Lulla, K. L., Chandra, R. C., & Sirigiri, K. S. (2025). Proxy-based thermal and acoustic evaluation of cloud GPUs for AI training workloads. The American Journal of Applied Sciences, 7(7), 111–127.

https://doi.org/10.37547/tajas/Volume07Issue07-12

12. Salunke, S. V., & Ouda, A. (2023). Ensemble Learning to Enhance Continuous User Authentication For Real World Environments. In Proceedings of the 2023 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom), Istanbul, Türkiye, 4–7 Jul 2023; pp. 102–108.

13. Rangu, S. (2025). Analyzing the impact of AI-powered call center automation on operational efficiency in healthcare. Journal of Information Systems Engineering and Management, 10(45s), 666–689. https://doi.org/10.55278/jisem.2025.10.45s.666

14. Weng, S., Wang, Q., Qu, L., Zhang, R., Cai, P., Qian, W., & Zhou, A. (2024). Lauca: A Workload Duplicator for Benchmarking Transactional Database Performance. IEEE Transactions on Knowledge and Data Engineering, 36, 3180–3194.

15. Ciolli, G., Mejías, B., Angelakos, J., Kumar, V., & Riggs, S. (2023). PostgreSQL 16 Administration Cookbook. Packt Publishing Ltd.: Birmingham, UK.

16. Bartunov, O., & Sigaev, T. (2024). Full-Text Search in PostgreSQL. PostgreSQL Documentation. Retrieved October 17, 2024, from https://www.postgresql.org/docs/current/textsearch.html

17. Reddy Dhanagari, M. (2025). Aerospike: The key to high-performance real-time data processing. Journal of Information Systems Engineering and Management, 10(45s), 513–531. https://doi.org/10.55278/jisem.2025.10.45s.513

18. Tongkaw, S., & Tongkaw, A. (2016). A comparison of database performance of MariaDB and MySQL with OLTP workload. In Proceedings of the 2016 IEEE Conference on Open Systems (ICOS), Langkawi, Malaysia, 10–12 Oct 2016; pp. 117–119.

19. Edrah, M., & Ouda, A. (2024). Enhanced Security Access Control Using Statistical-Based Legitimate or Counterfeit Identification System. Computers, 13, 159.

20. Bonthu, C., & Goel, G. (2025). Autonomous supplier evaluation and data stewardship with AI: Building transparent and resilient supply chains. International Journal of Computational and Experimental Science and Engineering, 11(3). https://doi.org/10.22399/ijcesen.3854

21. Bansal, P., & Ouda, A. (2022). Study on integration of FastAPI and machine learning for continuous authentication of behavioral biometrics. In Proceedings of the 2022 ISNCC, Shenzhen, China, 19–22 Jul 2022; pp. 1–6.

22. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A. (2003). Xen and the art of virtualization. ACM SIGOPS Operating Systems Review, 37, 164–177.

23. Salunke, S., Ouda, A., & Gagne, J. (2022). Transfer Learning for Behavioral Biometrics-based Continuous User Authentication. In Proceedings of the 2022 ISNCC, Shenzhen, China, 19–22 Jul 2022; pp. 1–6.

24. Han, J., Haihong, E., Le, G., & Du, J. (2011). Survey on NoSQL database. In Proceedings of the 6th International Conference on Pervasive Computing and Applications, Port Elizabeth, South Africa, 26–28 Oct 2011; pp. 363–366.

25. Durgam, S. (2025). CICD automation for financial data validation and deployment pipelines. Journal of Information Systems Engineering and Management, 10(45s), 645–664. https://doi.org/10.52783/jisem.v10i45s.8900

26. Kaur, K., & Sachdeva, M. (2017). Performance evaluation of NewSQL databases. In Proceedings of the 2017 International Conference on Inventive Systems and Control (ICISC), Coimbatore, India, 19–20 Jan 2017; pp. 1–5.

27. Murach, J. (2019). Murach's MySQL (3rd ed.). Mike Murach Associates: Fresno, CA, USA.

28. Seghier, N. B., & Kazar, O. (2021). Performance benchmarking and comparison of NoSQL databases: Redis vs MongoDB vs Cassandra using YCSB tool. In Proceedings of the 2021 ICRAMI, Tebessa, Algeria, 21–22 Sep 2021; pp. 1–6.

29. Armenatzoglou, N., Basu, S., Bhanoori, N., Cai, M., Chainani, N., Chinta, K., Govindaraju, V., Green, T. J., Gupta, M., Hillig, S., et al. (2022). Amazon Redshift re-invented. In Proceedings of the 2022 International Conference on Management of Data, Paphos, Cyprus, 12–17 Jun 2022; pp. 2205–2217.

30. Reddy Dhanagari, M. (2025). Aerospike vs. traditional databases: Solving the speed vs. consistency dilemma. International Journal of Computational and Experimental Science and Engineering, 11(3). https://doi.org/10.22399/ijcesen.3780

31. Aref, Y., & Ouda, A. (2024). HSM4SSL: Leveraging HSMs for Enhanced Intra-Domain Security. Future Internet, 16, 148.

32. Bog, A., Kruger, J., & Schaffner, J. (2008). A composite benchmark for online transaction processing and operational reporting. In Proceedings of the 2008 IEEE Symposium on Advanced Management of Information for Globalized Enterprises (AMIGE), Tianjin, China, 28–29 Sep 2008; pp. 1–5.

33. Ramsey, P. (2022). Postgres Indexing: When Does BRIN Win? Crunchy Data. Retrieved Oct 13 2024 from https://www.crunchydata.com/blog/postgres-indexing-when-does-brin-win

34. Bernstein, P. A., & Goodman, N. (1983). Multiversion concurrency control—Theory and algorithms. ACM Transactions on Database Systems (TODS), 8, 465–483.

35. Zhou, G., Huang, L., Li, Z., Tian, H., Zhang, B., Fu, M., Feng, Y., & Huang, C. (2021). Intever Public Database for Arcing Event Detection: Feature Analysis, Benchmark Test, and Multi-Scale CNN Application. IEEE Transactions on Instrumentation and Measurement, 70, 3518515.

36. Kubernetes, T. (2019). Kubernetes. Retrieved May 24, 2019, from Kubernetes.io

37. Abreha, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. ACM Computing Surveys (CSUR), 15, 287–317.

38. Callaghan, M. (2024). MySQL and Postgres vs the Insert Benchmark on a Large Server. Retrieved Oct 13 2024 from https://smalldatum.blogspot.com/2024/09/mysql-and-postgres-vs-insert-benchmark.html

39. Reddy Gundla, S. (2025). PostgreSQL tuning for cloud-native Java. International Journal of Computational and Experimental Science and Engineering, 11(3). https://doi.org/10.22399/ijcesen.3479

40. Kenler, E., & Razzoli, F. (2015). MariaDB Essentials. Packt Publishing Ltd.: Birmingham, UK.

41. Filip, P., & Cegan, L. (2020). Comparison of MySQL and MongoDB with focus on performance. In Proceedings of the 2020 ICIMCIS, Jakarta, Indonesia, 19–20 Nov 2020; pp. 184–187.

42. Sabharwal, N., & Edward, S. G. (2019). Hands on Google Cloud SQL and Cloud Spanner. Apress: New York, NY, USA.

43. Hellerstein, J. M., Naughton, J. F., & Pfeffer, A. (1995). Generalized Search Trees for Database Systems. University of Wisconsin–Madison, Department of Computer Sciences.

44. A Brief History of PostgreSQL. (2018). In PostgreSQL Documentation. The PostgreSQL Global Development Group: Athens, Greece.

45. Srilatha, S. (2025). Integrating AI into enterprise content management systems: A roadmap for intelligent automation. Journal of Information Systems Engineering and Management, 10(45s), 672–688. https://doi.org/10.52783/jisem.v10i45s.8904