

Scaling Distributed Databases

Thilak Raj Murugan

University of Wisconsin–Madison
Madison, WI, USA
tmurugan2@wisc.edu

Viswanath Pathmanaban

University of Wisconsin–Madison
Madison, WI, USA
pathmanaban@wisc.edu

Gokulnath Sourirajan

University of Wisconsin–Madison
Madison, WI, USA
sourirajan@wisc.edu

Shruthi Gopalakrishnan

University of Wisconsin–Madison
Madison, WI, USA
sgopalakris9@wisc.edu

1 Introduction

1.1 Background

Distributed databases are designed to maintain performance and reliability as data volumes grow from megabytes to hundreds of gigabytes or beyond. Among these, Hybrid Transactional and Analytical Processing (HTAP) systems such as TiDB aim to support both transactional and analytical workloads within a single architecture [5, 3]. TiDB achieves this by integrating a row-based transactional layer optimized for low-latency read/write operations with a columnar analytical layer optimized for large-scale queries. As data grows, HTAP systems face two major scaling challenges: performance isolation and data freshness. Performance isolation ensures that read heavy analytical queries do not interfere with the latency of transactional operations, while data freshness concerns how quickly newly written data becomes visible to analytical queries. Scaling TiDB to hundreds of gigabytes increases the complexity of synchronizing data across nodes and maintaining consistent performance. Techniques such as asynchronous replication between TiDB’s TiKV (transactional) and TiFlash (analytical) engines help mitigate these issues but introduce trade-offs between throughput and freshness.

In contrast, NoSQL systems such as Apache Cassandra are designed primarily for horizontal scalability and fault tolerance across distributed clusters [1, 2]. Cassandra employs a decentralized, peer-to-peer architecture with data partitioned and replicated across nodes using consistent hashing. Writes are first stored in memory and persisted to immutable SSTables on disk, allowing for high write throughput even as data grows into hundreds of gigabytes. Its tunable consistency model enables trade-offs between latency and data correctness, supporting high availability under varying workloads. However, as Cassandra scales, operational challenges emerge: compaction overhead increases, cache efficiency decreases, and repairs across replicas become costlier. Unlike HTAP systems, Cassandra is optimized for simple read and write queries rather than analytical workloads, and its scalability depends heavily on well-chosen partitioning and data models. TiDB focuses on unifying transactional and analytical processing within one system, while Cassandra emphasizes elastic horizontal growth to handle high-throughput, eventually consistent workloads at scale. Benchmarking tools such as NoSQLBench further enable systematic evaluation of these trade-offs [4].

1.2 Motivation

The increase in high-scale, data-centric applications across industries puts distributed databases at the center of modern infrastructure. Companies are increasingly depending on platforms that support high-throughput transactional workloads in addition to latency-sensitive analytical queries and fault tolerance. This motivates a closer look at how design choices in systems like TiDB and Cassandra play out in real circumstances, particularly when data sizes grow to hundreds of gigabytes and clusters undergo node addition, removal, or failure. Understanding these operational trade-offs—replication delay, rebalancing cost, and tail latency under contention—is important to decide which systems are suitable for modern workloads demanding both performance and fault tolerance.

Our interest in this project stems from the opportunity to bridge systems theory with practice by running controlled benchmarks and analyzing scaling behaviors directly. In addition, this project provides a hands-on opportunity to study critical behaviors that emerge only at scale, such as replication overhead, load balancing when nodes fail or join, and the latency implications of consistency models. Benchmarking against workloads and dataset sizes for TiDB and Cassandra enables us to examine not only raw performance but also system robustness under normal operation loads. These experiences will help place into perspective the relative strengths and weaknesses of two prominent distributed database paradigms while giving hands-on experience with infrastructure setup, benchmarking tools, and cluster management.

1.3 Problem Statement

This project examines how distributed SQL systems like TiDB and wide-column NoSQL stores like Apache Cassandra behave under scale and cluster dynamics. As datasets grow toward ~175 GB and cluster topology changes occur, both systems encounter distinct and overlapping challenges, including data partitioning and rebalancing, replication trade-offs, storage management, tail latency under contention, and operational complexity when deployed on GCP and Docker.

This project frames an empirical comparison and systems study that:

- Benchmarks TiDB across dataset sizes from small (~1 GB) to large (~175 GB) and Cassandra by scaling the number of read/write operations from 50k to 200M under different workloads.

- Measures how replication, rebalancing, and load redistribution behave during dynamic topology events (adding a node, removing a node, and node failure).

We quantify key operational and performance metrics like throughput and latency, replication lag, time to fully rebalance after topology changes, CPU/I/O/network utilization, and storage overhead — and analyze how these metrics correlate with configuration choices (replication factor, consistency level, partitioning strategy) and workload characteristics (read/write ratio, key skew). The core research questions are:

- (1) How do TiDB and Cassandra scale in throughput and latency as dataset size increases from MBs to GB? Where do bottlenecks appear?
- (2) What is the cost (in time, network, and performance impact) of adding a new node or recovering from a node failure for each system, and how does that cost change with dataset size?
- (3) How do replication semantics and consistency configurations affect user-visible latency and recovery behavior under node churn?

By focusing on measurable, repeatable experiments, initially on a local 1 GB setup and then scaled to a cloud deployment, the project aims to produce actionable guidance for practitioners choosing between distributed SQL HTAP engines and NoSQL wide-column stores for mid-to-large datasets, and to illuminate the operational tradeoffs encountered when scaling transactional databases to hundreds of gigabytes.

2 TiDB

2.1 Literature Review

TiDB is a distributed, cloud-native hybrid transactional and analytical processing (HTAP) database designed to support both low-latency transactional workloads and high-throughput analytical queries within a single unified system. Its architecture combines the strong consistency guarantees of traditional relational databases with the horizontal scalability and fault tolerance expected from modern distributed systems. By maintaining MySQL compatibility, TiDB enables seamless migration from existing relational workloads while extending them to distributed environments.

2.1.1 Architectural Design TiDB follows a shared-nothing architecture (Figure 1) with a strict separation between compute and storage layers, allowing each layer to scale independently. The compute layer consists of stateless TiDB servers that are responsible for SQL parsing, query optimization, and execution. Since these servers do not maintain persistent state, they can be elastically scaled or replaced in the event of failures without affecting data availability.

The storage layer is composed of two specialized engines designed for different workload characteristics. TiKV is a distributed, row-oriented key-value store optimized for low-latency transactional reads and writes, while TiFlash is a columnar storage engine optimized for large-scale analytical queries. This dual-engine design enables TiDB to efficiently route OLTP queries to TiKV and OLAP queries to TiFlash, providing native HTAP capabilities within the same system.

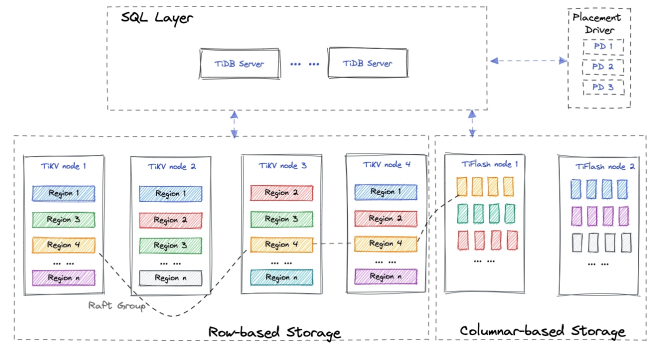


Figure 1: TiDB Architecture

2.1.2 Transaction Processing and Consistency TiDB provides strong consistency guarantees through the use of the Raft consensus protocol. Data in TiKV is divided into regions, each of which is replicated across multiple nodes. For every region, a Raft leader is elected to coordinate writes and ensure linearizable consistency in the presence of failures. This design allows TiDB to tolerate node and network failures while maintaining correctness.

Distributed transactions are implemented using a Percolator-style two-phase commit protocol, supported by globally synchronized timestamps. These timestamps are provided by the Placement Driver (PD), a central metadata service that manages region placement, replica scheduling, and load balancing across the cluster.

2.1.3 Performance Isolation To prevent analytical workloads from interfering with transactional performance, TiFlash nodes operate as Raft learners and asynchronously replicate data from TiKV.

This asynchronous replication model ensures that analytical queries observe near real-time data while maintaining strict performance isolation from OLTP workloads. Although this introduces a small amount of staleness for analytical reads, it significantly improves system throughput and reduces contention, making TiDB well-suited for mixed workload environments.

2.1.4 Scalability and Fault Tolerance TiDB achieves horizontal scalability by allowing independent scaling of compute and storage resources. Adding additional TiDB servers increases query processing capacity, while adding TiKV or TiFlash nodes expands storage and throughput. Region rebalancing and replica placement are handled automatically by the Placement Driver, reducing operational complexity.

Fault tolerance is ensured through Raft-based replication, automatic leader re-election, and transparent failover at the SQL layer. Together, these mechanisms allow TiDB to provide high availability and elasticity, making it suitable for large-scale, cloud-native deployments.

2.2 TPC-C Benchmark Evaluation

2.2.1 Introduction TPC-C is an industry-standard benchmark for evaluating online transaction processing (OLTP) systems. Unlike analytical workloads such as TPC-H, TPC-C models a high-throughput transactional environment composed of short, highly concurrent read-write operations. The benchmark mixes five transaction types

including NEW_ORDER, PAYMENT, and DELIVERY—that stress TiDB’s transaction scheduler, TiKV’s Raft replication, and the storage engine’s ability to handle frequent updates and row-level contention.

In this section, we study TiDB’s OLTP performance under TPC-C with a focus on how throughput, latency, and resource utilization evolve as the dataset grows and as client concurrency increases. These experiments allow us to observe when the system remains comfortably within its transactional capacity and when it transitions into CPU, I/O, or Raft-induced saturation.

2.2.2 Experiment Setup TPC-C experiments were executed on the same TiDB cluster deployment used for analytical tests, provisioned using TiUP Playground on a Google Cloud Platform (GCP) virtual machine. The cluster consisted of a TiDB SQL layer, multiple TiKV storage nodes, and a Placement Driver (PD). All components were left at their default configuration unless required by the benchmark.

The TPC-C workload was driven using `go-tpc tpcc`. A small sanity test with 10 warehouses validated schema creation, data loading, and transactional correctness before larger experiments were run. Two types of studies were then conducted: (1) a data-scaling experiment, where the warehouse count was increased while holding concurrency constant, and (2) a concurrency experiment, where client threads were varied on a fixed warehouse configuration. Each run used a fixed-duration execution window, and logs were captured for later analysis.

System behavior during each run was monitored through TiDB Dashboard and Grafana, which provided SQL latency distributions, TiDB and TiKV CPU utilization, disk I/O throughput, and Raft-related metrics. These measurements enabled direct correlation between throughput degradation or latency spikes and the underlying resource constraints in the TiDB-TiKV stack.

2.3 TPC-C Results

2.3.1 Sanity Validation (10 Warehouses) Before conducting large-scale experiments, a baseline validation was performed using 10 warehouses and 8 client threads. Both “no-wait” and “wait” modes were executed to confirm transactional correctness and cluster stability. At this scale, latency remained consistently low, TiDB and TiKV CPU utilization stayed well below saturation, and throughput showed no irregular fluctuations. These results confirm that the system was correctly configured and that subsequent scaling trends reflect genuine load effects rather than configuration errors.

Two representative metrics were used for validation. The “Database Time by SQL Type” graph shows uniformly low and stable execution times, indicating minimal queueing delays. The TiDB CPU graph also remains flat, confirming that the SQL layer is operating far below capacity.

2.3.2 Data Scaling Behavior We next examine how transactional performance evolves as dataset size increases. TPC-C was executed at 100 and 400 warehouses under the same workload settings. Table 1 summarizes key system metrics.

At 100 warehouses, the system exhibits ample headroom. Storage growth is modest, and TiKV maintains low Raft log pressure. Transaction latencies remain stable, and both TiDB and TiKV CPU utilization indicate that neither the SQL layer nor the storage layer is close to saturation.

W	Peak Storage	Avg Lat.	TiDB CPU	TiKV CPU
100	~74 GB	60–70 ms	44%	34%
400	~193 GB	80–100 ms	144%	101%

Table 1: Scaling effects at 100 and 400 warehouses.

At 400 warehouses, the system enters a different operational regime. Storage grows linearly, but CPU utilization increases disproportionately. TiDB CPU surpasses 140%, while TiKV CPU exceeds 100%, indicating that Raft replication, write scheduling, and MVCC updates are consuming nearly all available resources. Small spikes in database execution time and a noticeable increase in jitter suggest begin-commit delays caused by Raft followers struggling to keep up with log application. TiKV I/O activity also becomes more erratic, driven by larger Raft logs, compaction cycles, and greater write amplification.

These behaviors collectively indicate that scaling constraints arise primarily from the storage layer. Although TiDB can ingest larger datasets without difficulty, sustained transactional throughput becomes increasingly bound by Raft replication and I/O bottlenecks, rather than SQL execution.

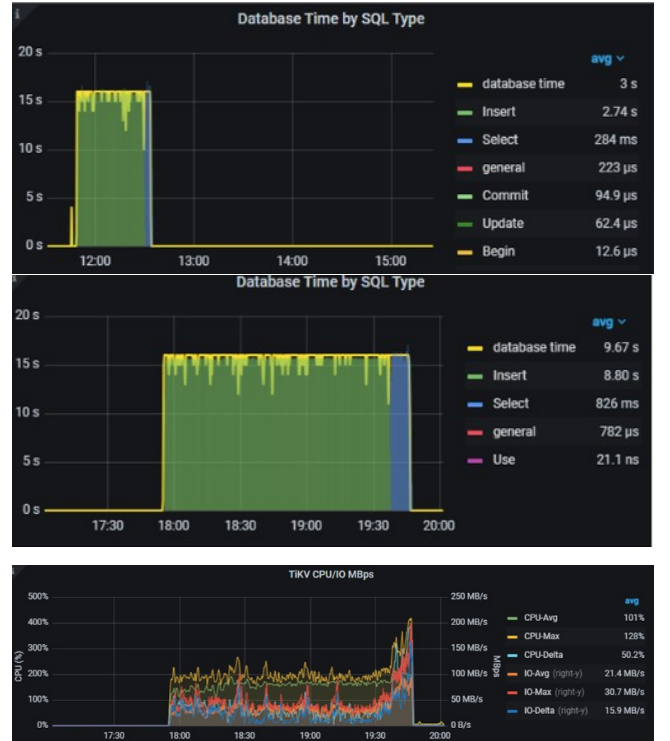


Figure 2: Impact of dataset size on SQL execution time and TiKV I/O

Figure 2 shows that increasing warehouse count from 100 to 400 leads to a clear shift in system behavior: SQL execution time distributions flatten at 100 warehouses but become increasingly erratic at 400, with longer and more variable execution phases for write-heavy operations such as Insert and Commit. This degradation

directly corresponds to higher TiKV disk I/O throughput, where sustained peaks and irregular bursts indicate that Raft replication and log writing are approaching device limits. The resulting CPU increase on TiKV nodes suggests that background Raft tasks—log application, replication, and compaction—begin to dominate execution time, reducing the cluster’s ability to serve foreground OLTP requests. Thus, scaling the dataset stresses TiKV’s storage and consensus layers before it exhausts TiDB compute capacity.

2.3.3 Concurrency Effects (200 Warehouses) To isolate the effect of client-level parallelism, the number of warehouses was fixed at 200 while thread count was varied across 16, 32, and 64 clients. Table 2 summarizes overall system throughput and latency trends.

Threads	tpmC	Total TPM	Avg Lat. (NewOrder/Pay)
16	5765	12802	82 / 57 ms
32	6355	14070	145 / 105 ms
64	4413	9793	299 / 234 ms

Table 2: Concurrency scaling at 200 warehouses

At 16 threads, the cluster remains under-loaded. CPU usage is moderate, commit latency is predictable, and the system does not yet exploit the full parallelism available in TiDB’s SQL executors or TiKV’s raftstore workers. Increasing to 32 threads brings the system to its peak throughput: latency rises relative to the 16-thread run but remains stable, TiDB and TiKV CPU utilization reach high yet sustainable levels, and Raft replication proceeds without backlog. This configuration provides the best balance between throughput and response time.

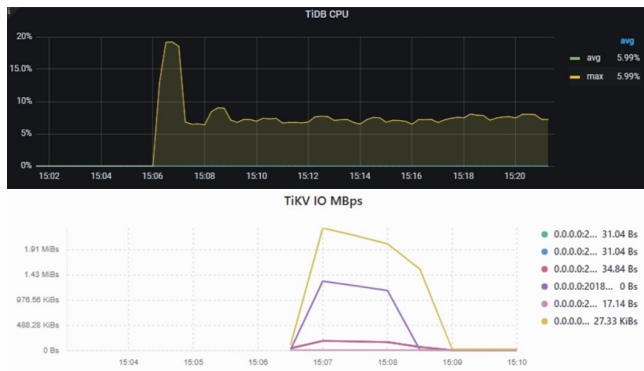


Figure 3: Concurrency impact on TiDB CPU and TiKV I/O

At 64 threads, performance degrades sharply. Throughput drops by nearly 30%, while latency more than doubles. Although TiKV I/O spikes indicating significantly more disk activity—this work does not translate into higher throughput. Instead, raftstore and scheduler queues saturate, commit latency grows, and the system performs increasing amounts of wasted work due to lock retries and transaction restarts. These symptoms indicate that the cluster is operating beyond its optimal concurrency envelope and that further parallelism only amplifies contention rather than improving throughput.

Figure 3 further illustrates this trend: at moderate thread counts, TiDB CPU remains well below saturation and TiKV shows proportional increases in traffic. At 64 threads, however, TiDB exceeds 130% CPU utilization, creating queuing delays in SQL planning and commit processing, while TiKV exhibits short, intense I/O spikes characteristic of backpressure from Raft message retries and stalled write batches. This mismatch—rapid CPU growth alongside bursty, inefficient I/O signals the shift from a throughput-limited regime to a contention-limited regime, where excessive concurrency generates more conflicts than useful work and causes overall throughput to collapse.

Across all TPC-C experiments, a consistent pattern emerges: scaling behavior is primarily determined by TiKV’s ability to sustain Raft replication and manage write-intensive workloads. While TiDB itself scales cleanly with additional CPU load, the storage layer becomes the limiting factor both as datasets grow and as concurrency increases. Data scaling introduces greater replication and compaction overhead, while concurrency scaling amplifies lock contention and increases Raft log pressure. These findings highlight the importance of tuning concurrency and provisioning adequate storage bandwidth when deploying TiDB for write-heavy OLTP workloads.

2.4 TPC-H Benchmark Evaluation

2.4.1 Introduction TPC-H is a decision-support benchmark designed to evaluate analytical workloads consisting of complex SQL queries with large table scans, multi-way joins, and aggregations. These queries place significant stress on query execution engines, memory usage, and disk I/O, making them well suited for studying analytical performance in distributed database systems.

In this section, we evaluate TiDB’s analytical performance using the TPC-H benchmark, focusing on how query execution behavior changes with dataset scale, client concurrency, and cache state. We analyze both overall query latency and per-query execution characteristics to identify dominant queries and performance bottlenecks. In addition, system-level metrics from TiDB Dashboard are used to relate observed performance trends to underlying resource utilization, providing insight into the practical scalability limits of TiDB for analytical workloads.

2.4.2 Experiment Setup All TPC-H experiments were conducted on a TiDB cluster deployed using TiUP Playground on a Google Cloud Platform (GCP) Compute Engine virtual machine. The cluster consisted of TiDB compute nodes, TiKV storage nodes, and a Placement Driver (PD), providing a distributed SQL execution environment. The virtual machine was provisioned with sufficient CPU and memory resources to support analytical workloads.

TiDB was configured with default settings unless explicitly required by the benchmark. Cluster status and system metrics were monitored using TiDB Dashboard, which was enabled during all runs to ensure cluster stability and to collect system-level observations for later analysis.

The TPC-H benchmark was executed using `tiup bench tpch`. The benchmark consists of 22 analytical SQL queries designed to evaluate decision-support workloads. Query correctness checks

were enabled for scale factor 1, as required by the benchmark specification. For larger scale factors, queries were executed without correctness validation to reduce execution overhead.

Analytical performance was evaluated by varying three key parameters:

- **Scale factor (SF)**, which controls dataset size.
- **Client concurrency**, which determines the number of parallel query executions.
- **Cache state**, comparing cold-start runs against subsequent warm-cache runs.

For data scaling experiments, TPC-H was executed at increasing scale factors while keeping concurrency fixed. For concurrency experiments, the scale factor was held constant and the number of client threads was varied. Cold versus warm cache behavior was evaluated by comparing the first execution after cluster startup with subsequent runs.

Each experiment was executed using a shell script to ensure consistent configuration across runs and to capture detailed benchmark logs. These logs were later parsed to extract per-query execution latency for analysis in the Results section.

2.5 TPC-H Results

2.5.1 Data Scaling Behavior We first evaluate how analytical query performance changes as the TPC-H scale factor increases. At smaller scale factors, overall execution time is dominated by a small subset of complex queries, while simpler queries contribute marginally to total runtime. As the scale factor increases, this imbalance becomes more pronounced.

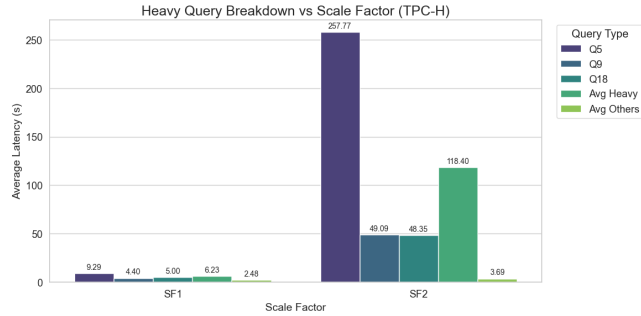


Figure 4: Query Latency vs Scale Factor

As shown in Figure 4, increasing the scale factor leads to a significant increase in total runtime, with Q5, Q9, and Q18 emerging as dominant contributors. These queries involve large joins and aggregations, and their execution time grows disproportionately with dataset size. In contrast, lighter queries exhibit relatively modest increases in latency.

System-level metrics from TiDB Dashboard (Figure 4) show that this increase in query latency coincides with higher disk I/O activity and increased CPU utilization. This indicates that data access and intermediate result materialization become key bottlenecks as the dataset grows. Although TiDB scales storage capacity effectively, analytical performance does not scale linearly with data size due to increasing pressure on memory and disk I/O.

2.5.2 Concurrency Effects Next, we examine the effect of client concurrency on TPC-H performance while keeping the scale factor fixed. At low concurrency, queries execute sequentially with predictable latency. Increasing concurrency initially improves overall utilization, but further increases lead to diminishing returns.

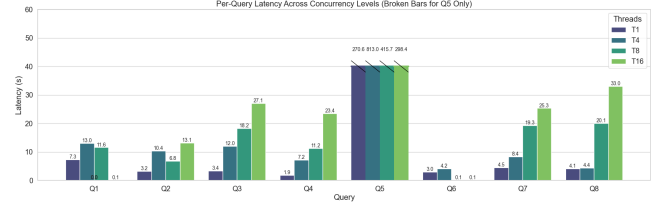


Figure 5: Per-Query Latency across Concurrency Levels

As shown in Figure 5, increasing concurrency causes heavy queries such as Q5, Q9, and Q18 to experience sharp latency increases, while lighter queries remain relatively stable. At higher thread counts, some heavy queries fail to complete due to memory pressure, resulting in incomplete query sets for those runs. This behavior indicates that analytical queries compete for shared resources, and that parallel execution amplifies memory and I/O contention rather than improving throughput.

These results highlight that, for TPC-H workloads, higher concurrency does not necessarily translate to better performance. Instead, excessive parallelism can overwhelm available resources, leading to stalled or failed query execution.

2.5.3 Cold vs. Warm Cache Behavior Finally, we compare cold-cache and warm-cache executions to understand the impact of caching on analytical performance. A cold run corresponds to the first execution after cluster startup, while a warm run benefits from cached data pages.



Figure 6: Cold vs Warm Cache: Latency and Disk I/O

As shown in Figure 6, warm-cache execution significantly reduces disk I/O activity and leads to smoother execution timelines. While overall query latency improves in warm runs, the magnitude

of improvement varies across queries. CPU and aggregation-heavy queries benefit more from caching, whereas some I/O-heavy queries show limited or inconsistent improvement.

This behavior suggests that caching primarily reduces repeated disk access, but does not eliminate the computational cost of large joins and aggregations. Consequently, while warm-cache execution improves stability and reduces I/O pressure, it does not fully mitigate the performance challenges posed by complex analytical queries.

Across all TPC-H experiments, the following trends are observed:

- Analytical performance is dominated by a small number of complex queries.
- Query latency grows non-linearly with dataset size.
- Increasing concurrency amplifies resource contention and can lead to query failures.
- Caching reduces disk I/O and stabilizes execution, but does not eliminate compute-heavy bottlenecks.

These results demonstrate that while TiDB can execute analytical workloads effectively at moderate scale, TPC-H performance is constrained by memory, disk I/O, and query complexity, particularly under high concurrency.

3 Apache Cassandra

3.1 Literature Review

Apache Cassandra is a distributed NoSQL database designed to prioritize high availability, fault tolerance, and linear horizontal scalability. Unlike HTAP systems, Cassandra is primarily optimized for OLTP-style workloads that require predictable low-latency access at large scale. Its design follows the principles of eventual consistency and decentralization, enabling robust operation even under partial failures.

3.1.1 Decentralized Architecture Cassandra employs a fully decentralized peer-to-peer architecture in which all nodes are equal and capable of serving client requests. There is no master node or centralized coordinator, eliminating single points of failure. This symmetry allows Cassandra clusters to scale incrementally by adding nodes without requiring significant reconfiguration.

Cluster membership and failure detection are handled through a gossip protocol, which enables nodes to exchange state information in a scalable and fault-tolerant manner. Because metadata is distributed across the cluster, Cassandra avoids centralized bottlenecks and remains resilient under high churn and failure scenarios.

3.1.2 Data Partitioning Cassandra partitions data using consistent hashing, mapping each partition key to a token in a circular hash space known as the token ring (Figure 7). Each node in the cluster is responsible for one or more contiguous token ranges, which define the subset of data it owns.

This partitioning strategy minimizes data movement when nodes are added or removed, as only the affected token ranges need to be reassigned. By hashing partition keys into a large 64-bit token space, Cassandra ensures uniform data distribution and balanced load across nodes, which is critical for achieving linear scalability.

3.1.3 Data Replication Data replication in Cassandra is governed by a configurable replication factor, which specifies the number

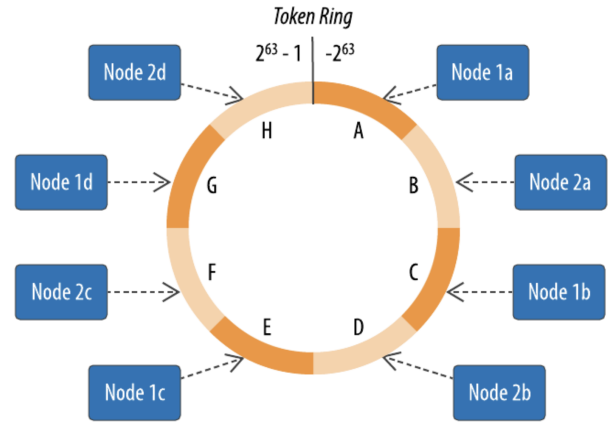


Figure 7: Token ring formed by Cassandra nodes

of replicas maintained for each data item. All replicas are treated equally, and any replica can serve read or write requests. Cassandra allows clients to specify the desired consistency level for each operation, enabling explicit trade-offs between latency, availability, and consistency.

Replica placement is determined by the chosen replication strategy. SimpleStrategy is typically used for single data center deployments, while NetworkTopologyStrategy distributes replicas across racks and data centers to improve fault tolerance and support geo-distributed deployments.

3.1.4 Write Path and Storage Engine Cassandra's write path is optimized for high throughput and low latency. Incoming writes are first appended to a commit log and stored in in-memory structures before being flushed to disk as immutable SSTables. This append-only design minimizes random disk I/O and allows Cassandra to sustain heavy write workloads even as dataset sizes grow significantly.

Over time, multiple SSTables are merged through compaction processes, which reclaim space and improve read performance. While compaction is essential for long-term efficiency, it introduces background overhead that can impact performance if not carefully managed.

3.2 Scaling Cassandra and NoSQL Benchmarking

3.2.1 Introduction As NoSQL databases scale to support large datasets and high request rates, several system-level challenges arise. In Apache Cassandra, compaction overhead becomes increasingly significant as data volume grows, since background compactions introduce additional disk I/O and can impact tail latency. Maintaining efficient caching is also critical for sustaining low read latencies, particularly when the working set exceeds available memory. Furthermore, as replication factors increase, repairs across replicas incur higher network and coordination costs, affecting overall system efficiency and operational complexity.

To systematically evaluate these scaling challenges, NoSQL-Bench is used as the benchmarking framework. NoSQLBench is an

open-source, extensible benchmarking suite designed for NoSQL systems, including Cassandra. It supports realistic workload emulation by generating synthetic datasets and executing configurable read and write access patterns that approximate real-world application behavior. Workload definitions are expressed declaratively using YAML templates, enabling fine-grained control over schema initialization, ramp-up phases, and steady-state benchmarking. This flexibility makes NoSQLBench well-suited for analyzing throughput scalability, latency behavior, and performance bottlenecks as system load increases.

Together, Cassandra and NoSQLBench provide a controlled experimental environment for studying the impact of scale on NoSQL database performance, forming the foundation for the experimental evaluation presented in this work.

3.2.2 Experimental Setup The experimental evaluation was conducted using Apache Cassandra deployed in a Docker-based environment to ensure isolation, reproducibility, and consistency across benchmark runs. Cassandra was executed inside a Docker container using a single-node configuration with a replication factor of one, allowing the study to focus on node-level performance without the influence of replication or inter-node communication overheads.

All benchmarks were executed on the host machine using NoSQLBench as the workload generator, which connected to the Cassandra instance through the CQL interface exposed by the container. Client-side load generation was configured with a fixed concurrency of 16 threads to maintain consistent request pressure across experiments. Containerization ensured that Cassandra’s runtime environment, dependencies, and configuration remained stable across all workload scales.

Benchmark execution followed a structured pipeline consisting of three phases: schema initialization, dataset ramp-up, and steady-state benchmarking. The ramp-up phase was used to populate the dataset and warm Cassandra’s caches, mitigating cold-start effects and ensuring that measured performance reflected steady-state behavior. Performance metrics were collected exclusively during the main execution phase.

To evaluate scalability, the total number of operations was increased systematically from 50K to 500K and 2M cycles, allowing throughput and latency trends to be observed as dataset size and runtime increased. Performance measurements included throughput, average latency, tail latency (p95), and total execution time. Latency statistics were obtained using Cassandra’s internal metrics via `nodetool tablehistograms`, complemented by NoSQLBench runtime statistics.

3.3 Cassandra Benchmarking Workload

The benchmarking experiments use the NoSQLBench `cql_keyvalue` workload, a simple key-value access pattern designed to evaluate baseline Cassandra performance. This workload models applications that store and retrieve unstructured or semi-structured data using text-based keys and values, a common usage pattern in NoSQL systems.

The workload operates on a single table consisting of a text primary key and a text value, where the primary key serves as the partition key. The simplicity of the schema enables performance behavior to be attributed directly to Cassandra’s storage engine

and execution paths, without confounding effects from secondary indexes, clustering columns, or complex query patterns.

3.3.1 Execution Phases Each benchmark run is divided into three execution phases:

- **Schema setup phase:** Creates the keyspace and key-value table using Cassandra’s `SimpleStrategy` with a configurable replication factor.
- **Ramp-up phase:** Inserts sequentially generated key-value pairs to populate the dataset and warm Cassandra’s caches and memtables, ensuring the system reaches steady state before measurements are collected.
- **Main execution phase:** Performs read and write operations according to a specified workload mix while collecting performance metrics.

3.3.2 Data Generation and Access Patterns Key and value generation is handled using NoSQLBench bindings. During the ramp-up phase, sequential keys and values are used to populate the dataset uniformly. In the main execution phase, keys are selected using a uniform random distribution over a large keyspace to avoid artificially favoring cache locality. Values are generated using hashed distributions to prevent repeated writes of identical data.

3.3.3 Read and Write Mix The main execution phase supports varying read and write ratios through separate operation blocks while scaling the total operation count:

- **Balanced workload (1:1):** Equal proportions of reads and writes.
- **Read-heavy workload (9:1):** Emphasizes read-dominated access patterns.
- **Write-heavy workload (1:9):** Emphasizes sustained write throughput.

3.4 Results

3.4.1 Throughput and Total Execution Time Figure 8 illustrates the throughput trends and total execution time for the Cassandra workload under varying operation counts (50K, 500K, and 2M cycles) and access patterns (balanced, read-heavy, and write-heavy).

Across all workload types, throughput increased consistently as the number of cycles grew. At 50K cycles, throughput remained modest, reflecting fixed initialization and coordination overheads. As the workload scaled to 500K and further to 2M cycles, throughput improved significantly, stabilizing around 9–9.5K operations per second. This trend indicates that Cassandra efficiently amortizes overheads at higher request volumes and sustains stable request processing rates under increased load.

Total execution time exhibited near-linear growth with respect to the number of cycles. The execution time increased proportionally from tens of seconds to several minutes at 2M cycles. The similarity in execution times across read-heavy, write-heavy, and balanced workloads suggests that Cassandra handled mixed and skewed access patterns with comparable efficiency in this single-node configuration.

Overall, the throughput and execution time results demonstrate predictable scaling behavior, indicating that Cassandra maintains consistent performance characteristics as workload intensity increases.

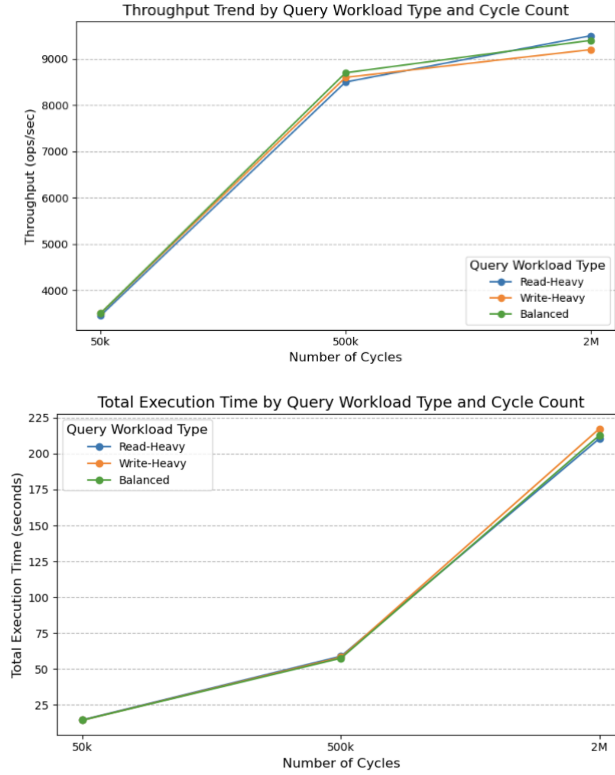


Figure 8: Throughput and Execution Time

3.4.2 Read and Write Latency Figure 9 presents the read and write latency distributions across different workload scales, focusing on average latency as well as higher percentiles (p75, p95, and p99).

Read latency remained low and stable across all cycle counts. Average read latency consistently stayed below 0.02 ms, while p95 latency remained under 0.1 ms even at the largest scale of 2M cycles. Although minor fluctuations were observed in higher percentiles, no significant latency degradation occurred as the workload scaled, indicating effective read path optimization and cache utilization.

Write latency exhibited similarly stable behavior. Average write latency remained around 0.01 ms across all tested scales, with p95 latency consistently below 0.04 ms. Higher-percentile write latencies (p99) showed slight variation at lower scales but stabilized as the workload increased, suggesting efficient write batching and compaction handling under sustained load.

Importantly, neither read-heavy nor write-heavy workloads caused noticeable latency spikes, even at higher operation counts. This stability indicates minimal degradation under higher load.

4 Conclusion

4.1 TiDB

Across both TPC-H and TPC-C, our evaluation highlights the fundamental trade-offs TiDB faces when scaling analytical and transactional workloads. For analytical processing, TiDB performs well at small scales, but runtime quickly becomes dominated by a small set

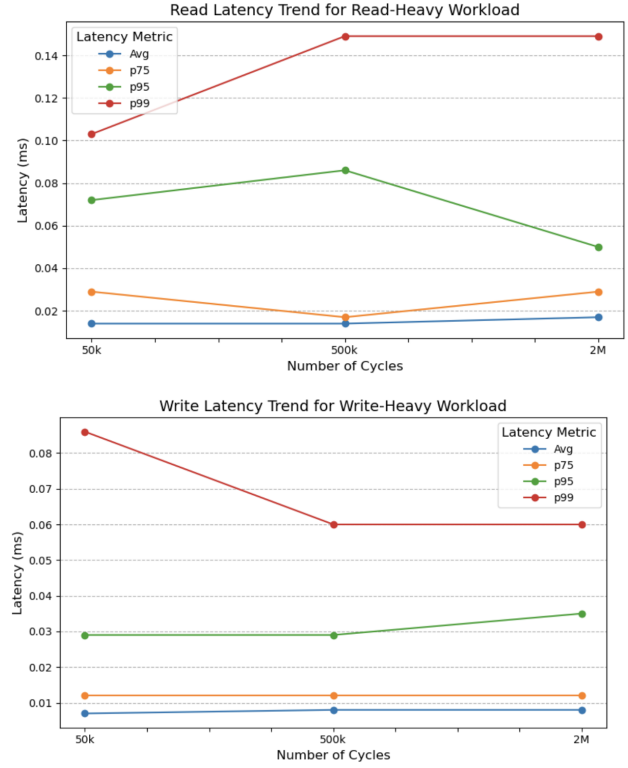


Figure 9: Read and Write Latency

of complex join-heavy queries whose latency grows non-linearly with dataset size. Increasing concurrency further amplifies memory and I/O contention, causing throughput to plateau or decline. Warm-cache runs reduce disk stalls, yet the execution of large scans and aggregations remains constrained by compute and intermediate data movement, revealing practical limits in TiDB’s analytical scalability.

Transactional results from TPC-C show a similar pattern: TiDB maintains stable throughput and predictable commit latency at moderate warehouse counts and concurrency, but performance deteriorates once the system exceeds its optimal load envelope. At high thread counts, Raft replication, lock conflicts, and scheduler queue buildup lead to superlinear latency growth and a measurable drop in throughput. Together, these findings demonstrate that while TiDB’s distributed architecture provides strong consistency and horizontal growth, real scalability is governed by workload characteristics, internal coordination costs, and resource contention. Effective deployment therefore requires careful tuning of workload intensity, concurrency, and cluster configuration.

4.2 Cassandra

The experimental results highlight Cassandra’s ability to scale predictably under increasing workload sizes. Throughput improves with scale, execution time grows linearly, and both read and write latencies remain consistently low across all tested scenarios. These results demonstrate that, for the NoSQL workload and single-node

deployment used in this study, Cassandra delivers stable and efficient performance under balanced, read-heavy, and write-heavy access patterns.

References

- [1] Apache Software Foundation. 2025. Apache cassandra. GitHub repository. Retrieved Oct. 17, 2025 from <https://github.com/apache/cassandra>.
- [2] Jeff Carpenter and Eben Hewitt. 2022. *Cassandra: The Definitive Guide: Distributed Data at Web Scale*. (3rd ed.). O'Reilly Media, Sebastopol, CA. ISBN: 9781492097133.
- [3] Elena Milkai, Yannis Chronis, Kevin P. Gaffney, Zhihan Guo, Jignesh M. Patel, and Xiangyao Yu. 2022. How good is my htap system? In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*. doi:10.1145/3514221.3526148.
- [4] NoSQLBench Contributors. 2025. Nosqlbench. GitHub repository. Retrieved Oct. 17, 2025 from <https://github.com/nosqlbench/nosqlbench>.
- [5] PingCAP. 2025. Tidb. GitHub repository. Retrieved Oct. 17, 2025 from <https://github.com/pingcap/tidb>.