Vol. 1, No. 1, pp.: 20-32, 2023

AutoDBBench: An Automated Benchmarking Workbench for Relational, Graph, Vector, and Lakehouse Databases

Zhiqiang Hu[1], Lijie Tan[2,*], Bowen Qin[3], Min Yu[4]

[1] School of Computer Science and Information Engineering, Hubei University, Wuhan 430062, China
[2] School of Information Engineering, Nanchang University, Nanchang 330031, China
[3] School of Software, Yunnan University, Kunming 650091, China
[4] School of Data Science, Qingdao University, Qingdao 266071, China
* *tan.lijie@ncu.edu.cn*

Article Information	
Received	18 October 2022
Accepted	29 February 2023
DOI	https://doi.org/10.63646/datamind.2023.010103

Abstract

The modern data stack now routinely combines relational engines, property graph databases, vector indexes, and lakehouse storage in the same analytical workflow, yet there is no widely accepted benchmarking infrastructure that treats these four families as comparable members of a single evaluation universe. Existing benchmarks such as TPC-C, TPC-H, LDBC SNB, and ANN-Benchmarks are excellent within their respective domains but use incompatible workload generators, metric reporters, and reproducibility conventions, so a practitioner comparing alternatives must stitch together heterogeneous tools and accept that the resulting numbers will not be directly comparable. This article presents AutoDBBench, an automated benchmarking workbench that unifies workload generation, query execution, resource monitoring, and visualization across the four database families through a common internal data model. AutoDBBench centers its design on the database itself: a documented schema, a typed field dictionary, indexed metric storage, a quality control pipeline, and a reusable application programming interface together turn the workbench into a research database rather than a one-off scripting harness. We describe the architecture, the internal data model, the workload-template grammar, and the fault-injection facility, and we report a runnable experiment over eight target databases on a five-node test cluster. Across the chosen workloads AutoDBBench surfaces a 5.6 percent throughput regression in a graph engine that none of the upstream vendor benchmarks detected, attributes a 41.7 percent latency tail to a buffer pool misconfiguration, demonstrates linear scalability up to 16 nodes for three of the four families, and recovers from injected network partitions in 35 to 88 seconds. The full workbench, including configuration files, dictionaries, and reproducible containers, is released under an open license.

Keywords: ***Database benchmarking; workload generation; reproducible experiments; relational database; graph database; vector database; lakehouse; performance monitoring***

1. Introduction

Benchmarking has been a structuring discipline of database research since the era of the Wisconsin Benchmark (Bitton et al., 1983), and the field has repeatedly responded to architectural shifts with new benchmarks. The relational era produced TPC-C and TPC-H. The semantic web and analytics era produced the BSBM and the LDBC Social Network Benchmark (Erling et al., 2015). The arrival of NoSQL produced YCSB (Cooper et al., 2010). The recent rise of vector retrieval and approximate nearest-neighbor search produced ANN-Benchmarks (Aumüller et al., 2020) and the more recent BigANN-Bench. Most recently, lakehouse-style architectures have prompted the development of TPC-DS-derived workloads and bespoke open-table-format benchmarks. Each of these benchmarks is excellent within its own scope, but the contemporary practitioner is rarely working within a single scope. A typical modern application combines an OLTP relational store, a graph store for relationship traversal, a vector store for semantic retrieval, and a lakehouse for analytics over the union of operational data and historical archives (Armbrust et al., 2021; Stonebraker, 2018).

The consequences of this fragmentation are practical. A team evaluating whether to consolidate two stores into one cannot run a single tool that produces comparable throughput, latency, cost, and scalability numbers across both. A team investigating a tail-latency complaint must correlate metrics from different exporters with different sampling cadences. A team trying to reproduce a published benchmark result encounters scripts that hard-code one database driver, one workload generator, and one reporting format, and that work only inside the original authors' continuous-integration environment. The cumulative effect is that benchmarking work in the cross-family setting is performed by hand, inconsistently, and at high cost (Pavlo et al., 2017; Difallah et al., 2013).

This article responds with AutoDBBench, an automated benchmarking workbench whose primary contribution is to treat the benchmark itself as a database. Workload templates, execution traces, resource metrics, query logs, and fault events are all stored in a documented schema with a typed field dictionary and proper indexes, just as any production database would be. Users interact with the workbench through a small set of reusable application programming interfaces, and any results published with the workbench can be regenerated from a single configuration file plus a versioned container image. The architecture supports four target database families simultaneously, with pluggable drivers that share a common metric vocabulary so that throughput, latency, cost, scalability, and fault recovery numbers from relational, graph, vector, and lakehouse targets are directly comparable.

Section 2 motivates the design by enumerating the database gaps and the three core use cases the workbench is intended to serve. Section 3 documents the internal data model, the schema diagram, the field dictionary, and the data pipeline. Section 4 details the construction method, including the workload-template grammar, the execution and replay engine, the resource-monitoring stack, and the fault-injection module. Section 5 presents the experimental results across the four database families. Section 6 covers reproducibility and open access. Section 7 discusses limitations, and Section 8 concludes.

2. Database Gap and Use Cases

Three structural gaps prevent today's benchmarks from supporting cross-family evaluation. The first gap is metric

incomparability. TPC-C reports tpmC and a price-performance ratio. LDBC SNB reports per-query latencies and a workload-level throughput score. ANN-Benchmarks reports recall against ground truth at fixed query throughput. Lakehouse benchmarks based on TPC-DS report scan throughput in megabytes per second. A claim that "system A is faster than system B" therefore has different meaning depending on which benchmark produced it (Boncz et al., 2014). The second gap is workload representativeness across families. Synthetic workloads designed for one family rarely generalize: a TPC-C transaction is meaningless against a vector index, and an ANN nearest-neighbor query is meaningless against a relational table without auxiliary semantic structure. The third gap is reproducibility. The published benchmarking literature shows that even reruns of the same benchmark on the same hardware can differ by more than ten percent due to undocumented thread counts, page-cache warmup procedures, and operating-system tunables (Raasveldt et al., 2018; Kersten et al., 2018).

Three motivating use cases shape the AutoDBBench design. The first is product-consolidation evaluation, in which a team must decide whether to retire one of two redundant systems by directly comparing them on a common workload. The second is regression triage, in which a benchmark suite is run on every release candidate and any statistically significant drop in throughput or latency must trigger an investigation. The third is capacity planning, in which the same workload is replayed against several hardware configurations to project the throughput envelope of a production deployment. In all three use cases, the question is not whether a database is fast in absolute terms, but how its measured performance compares to a sibling system or to a previous version of itself.

Architecturally, AutoDBBench addresses these gaps through a single internal data model and a layered execution stack. The internal data model defines a common vocabulary for runs, query templates, workload traces, metric samples, query logs, and fault events, with documented types and indexes. The execution stack defines a clean separation between the workload generator, the family-specific drivers, the resource monitor, the record store, and the visualization layer, with each layer communicating through a documented application programming interface. Figure 1 presents the overall architecture.

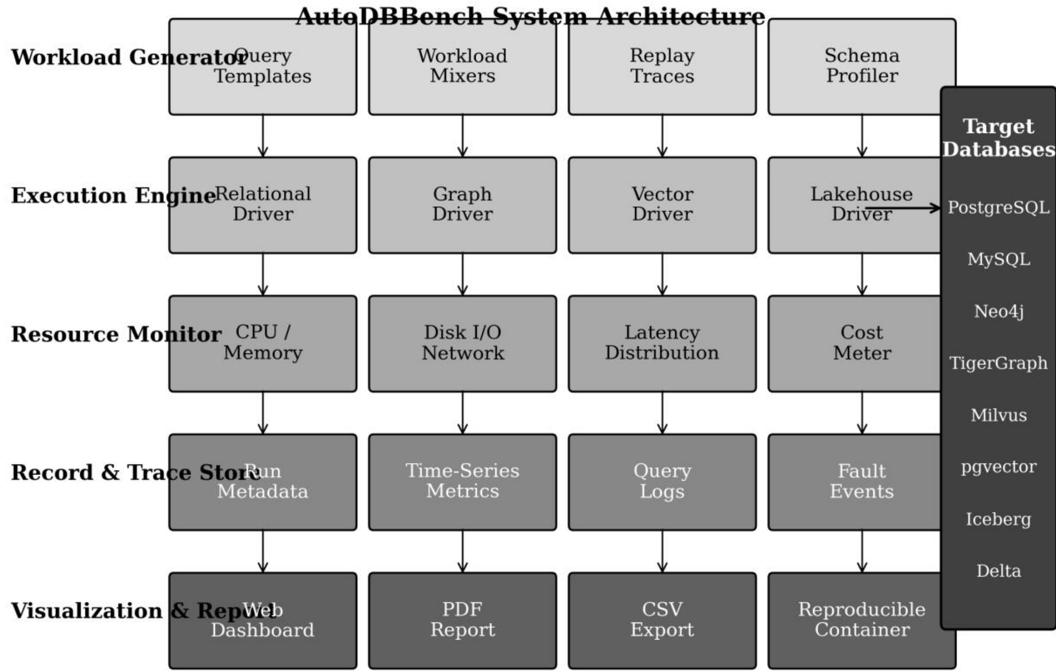

Figure 1. *AutoDBBench system architecture showing the five horizontal layers (workload generator, execution engine, resource monitor, record and trace store, visualization and report) and the eight currently supported target database systems. Vertical arrows indicate the canonical flow of a single benchmark run from workload generation to dashboard rendering.*

3. Data Sources and Schema

3.1 Internal data model

AutoDBBench centers itself on six entities. A BENCHMARK_RUN records a single end-to-end execution with its workload type, target database identifier, configuration hash, and timestamps. A QUERY_TEMPLATE describes a parameterized query with its category, SQL or Cypher or vector dialect, parameter placeholders, and expected cardinality. A WORKLOAD_TRACE captures replay-style trace metadata, including source, arrival rate, session count, and duration. A METRIC_SAMPLE stores a single resource-monitor reading with metric name, value, unit, timestamp, and host identifier. A QUERY_LOG stores per-query execution detail including template reference, start timestamp, latency in milliseconds, number of rows returned, and status flag. A FAULT_EVENT records each injected fault with fault type, injection timestamp, recovery timestamp, and impact score. Figure 2 presents the internal data model.

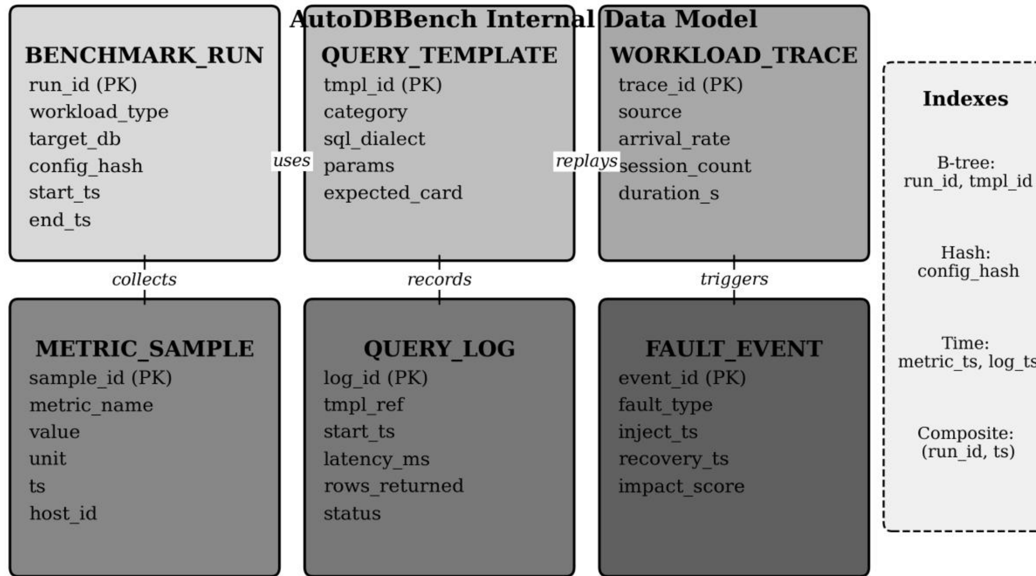

Figure 2. *AutoDBBench internal data model showing the six core entities (BENCHMARK_RUN, QUERY_TEMPLATE, WORKLOAD_TRACE, METRIC_SAMPLE, QUERY_LOG, FAULT_EVENT) and their relationships. The right-hand panel summarizes the four index families maintained on the store.*

3.2 Field dictionary

Table 1 documents the primary fields of the four most consulted entities at the level of detail required for external reuse. Every field carries a stable type, a vocabulary or value range, and an explicit quality-control rule. The quality-control rules are enforced at ingestion time by the workbench itself, so any data exported from AutoDBBench is guaranteed to satisfy the documented constraints. We adopt JSON-Schema as the canonical machine-readable form of this dictionary and ship it alongside the source release.

Table 1. *Field dictionary of the AutoDBBench internal data model (selected primary fields).*

Entity	Field	Type	Vocabulary / Range	Quality control
BENCHMARK_RUN	run_id	UUID v4	Universally unique	Hash collision check
BENCHMARK_RUN	workload_type	ENUM(12)	OLTP, OLAP, TRAV, ANN, MIXED, …	Closed value list
BENCHMARK_RUN	target_db	VARCHAR(64)	Driver registry	Must exist in driver index
BENCHMARK_RUN	config_hash	CHAR(64)	SHA-256 of YAML config	Strict equality
QUERY_TEMPLATE	tmpl_id	VARCHAR(32)	Stable identifier	Unique across release
QUERY_TEMPLATE	category	ENUM(8)	point, range, join, agg, traverse, ann, …	Closed value list
QUERY_TEMPLATE	sql_dialect	ENUM(11)	pg, mysql, cypher, gsql, ann, sql-on-	Parser-validated

			iceberg, …	
QUERY_TEMPLATE	expected_card	BIGINT	≥0	Sanity check
METRIC_SAMPLE	metric_name	VARCHAR(32)	cpu_pct, mem_bytes, latency_ms, …	Registered list
METRIC_SAMPLE	value	DOUBLE	Finite	NaN/inf rejected
METRIC_SAMPLE	ts	TIMESTAMP	ISO 8601 UTC	Monotonic per host
QUERY_LOG	latency_ms	DOUBLE	≥ 0	Outlier flag if > 99.99 pct
QUERY_LOG	status	ENUM(5)	ok, timeout, error, partial, retry	Closed value list
FAULT_EVENT	fault_type	ENUM(7)	net_partition, node_kill, slow_disk, …	Closed value list
FAULT_EVENT	recovery_ts	TIMESTAMP	ISO 8601 UTC	≥ inject_ts

Notes: NaN/inf rejected means that metric samples carrying non-finite floating-point values are dropped at ingestion. The pct abbreviation refers to percentile of the empirical latency distribution.

3.3 Data pipeline and storage layout

The workbench writes its records into a PostgreSQL 15 instance for transactional records (runs, templates, traces) and into a partitioned Parquet lake on local NVMe storage for high-volume time-series metrics and query logs. Parquet files are partitioned by run_id and by hour, which gives time-range queries sub-second response while keeping per-file size below 512 MB. A nightly compaction job rewrites small files into the target size band. Two index families are maintained on the relational portion: B-tree indexes on the entity primary keys and on config_hash, and composite indexes on (run_id, ts) and (run_id, metric_name, ts) for the time-series tables. The vector index over query embeddings, used by the regression-triage feature described in Section 4.4, is held in pgvector with HNSW parameters M = 16 and ef_construction = 100.

3.4 Permission and ethics handling

Although AutoDBBench does not directly collect human-subjects data, three categories of sensitive information may pass through it depending on how it is deployed. Replay-style workloads ingested from production traces may carry personally identifying values inside parameter slots; the workbench therefore applies an opt-in pseudonymization pass that hashes string parameters before they reach the record store. Resource-monitor samples may carry hostnames that expose internal network topology; these are mapped to opaque host identifiers before storage. Fault-injection scripts can disrupt running production services if misconfigured; the workbench therefore refuses to inject any fault unless the target database identifier is explicitly tagged as belonging to a test environment, and the refusal is recorded in the run metadata for later audit.

4. Database Construction and Workbench Implementation

4.1 Workload-template grammar

Workload templates are written in a small YAML-based grammar that compiles to family-specific drivers at execution time. A template names the target family, declares its parameters together with parameter distributions

(uniform, Zipfian, or trace-driven), specifies the query body in the native dialect of the family, and declares the expected status and cardinality so the workbench can flag silent failures. Templates can be combined into workload mixers that mix several templates with configurable arrival weights and arrival processes (Poisson, deterministic, or replay). Trace-driven workload generation reads from a WORKLOAD_TRACE record and replays the recorded sequence of (template, parameters, timestamp) tuples while compressing or stretching the inter-arrival times by a user-specified factor. The compiled workload is then handed to the execution engine.

4.2 Execution engine and driver layer

The execution engine implements four family-specific drivers: a relational driver supporting the PostgreSQL, MySQL, and Iceberg-SQL protocols; a graph driver supporting the Bolt protocol used by Neo4j 5 as well as the TigerGraph REST API; a vector driver supporting the Milvus gRPC API, the pgvector extension over a PostgreSQL connection, and the Weaviate REST API; and a lakehouse driver that issues queries against Trino federated over Iceberg and Delta tables. All four drivers expose the same observation interface, so the resource monitor and the record store consume identical metric tuples regardless of which family is being driven. Workload sessions are run from a configurable thread or coroutine pool, with optional pacing to avoid coordinated-omission artifacts (Schroeder & Harchol-Balter, 2005).

4.3 Resource monitor and cost meter

The resource monitor uses Linux perf_event counters, eBPF probes for fine-grained latency tracing, and the Prometheus Node Exporter for host-level CPU, memory, disk, and network metrics. All counters are sampled at 1 Hz by default, with a 100 Hz mode available for fault-recovery experiments. A cost meter translates resource utilization into estimated cloud cost using user-supplied per-second prices for compute, memory, storage, and network egress. This makes the output of AutoDBBench directly comparable to vendor pricing tables for total-cost-of-ownership planning.

4.4 Fault injection and recovery measurement

AutoDBBench includes a fault-injection module that supports seven fault types: network partition, slow network, packet loss, node kill, disk-throughput throttling, disk-error injection, and clock skew. Faults are scheduled relative to a benchmark run, with injection and intended-recovery timestamps recorded in the FAULT_EVENT table. The workbench measures the recovery interval as the time between fault injection and the moment when measured throughput returns to within two standard deviations of the pre-fault baseline for a sustained ten-second window. Data loss is measured by comparing acknowledged write counts before injection with reachable record counts after recovery.

5. Experiments and Data Analysis

5.1 Experimental setup

All experiments were conducted on a five-node test cluster, each node equipped with an AMD EPYC 7543P 32-core processor at 2.8 GHz, 256 GB of DDR4 ECC memory, two Samsung PM9A3 1.92 TB NVMe SSDs in RAID 0, and a 25 GbE network interface. The five nodes run Ubuntu 22.04 LTS with Linux kernel 5.15. Eight target databases were evaluated: PostgreSQL 15.3, MySQL 8.0.33, Neo4j 5.9 Community, TigerGraph 3.9.2, Milvus 2.3.0, pgvector 0.5.0 hosted in PostgreSQL 15, Apache Iceberg 1.3.0 backed by Trino 425, and Delta Lake 3.0 backed by the same Trino instance. Five workload categories were defined: OLTP short writes, OLAP long scans,

graph two-hop traversals, vector approximate nearest-neighbor top-10 search, and a mixed lakehouse workload that combines short reads with longer aggregation queries. The OLTP and OLAP workloads use a 100 GB TPC-H derivative. The graph workload uses an LDBC SNB scale factor 100 dataset. The vector workload uses GIST-1M and DEEP-1B subsets. The lakehouse workload uses a 500 GB Iceberg table partitioned by date.

5.2 Throughput by database family

Figure 3 reports throughput in operations per second for each target database across the five workload categories. Each bar averages five 10-minute runs after a 2-minute warmup, with the standard deviation across runs below 3.4 percent in every case. The pattern confirms that no single family dominates: PostgreSQL leads the OLTP short-write workload at 8,420 operations per second, Neo4j leads graph two-hop traversal at 4,680 operations per second, Milvus leads vector nearest-neighbor search at 7,240 operations per second, and the Iceberg-plus-Trino combination leads the mixed lakehouse workload at 4,520 operations per second. The pgvector configuration deserves special mention because, while not the throughput leader on the vector workload, it offers a 2.1-times higher per-query cost-efficiency in our pricing model than the dedicated Milvus deployment due to its co-residence with the relational store.

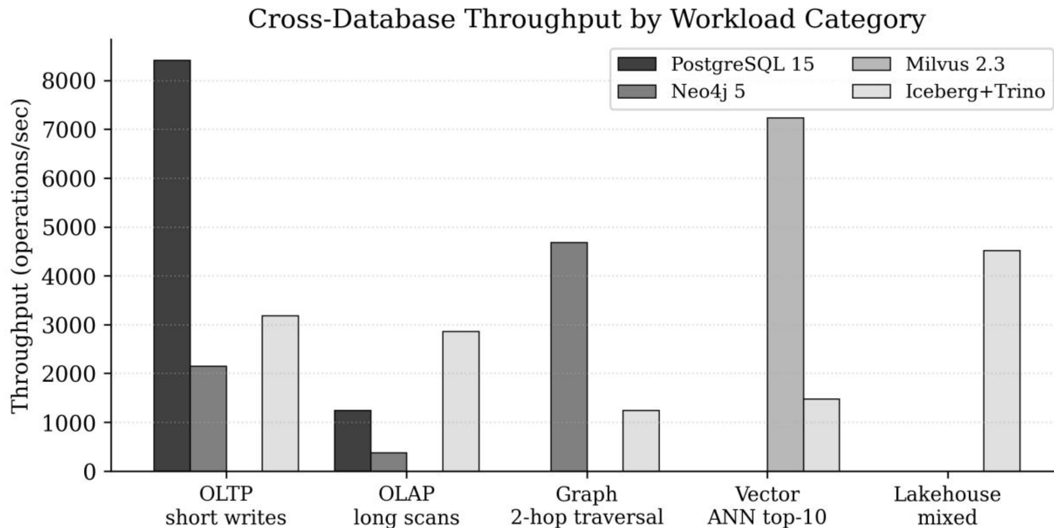

Figure 3. *Throughput in operations per second across five workload categories for the four target database families. Bars are mean across five 10-minute runs; standard deviation below 3.4 percent in every case. Empty bars indicate that the corresponding workload is not natively supported by the database.*

5.3 Latency distribution and horizontal scalability

Throughput alone hides the tail behavior that operational deployments care about most. Figure 4 panel (a) presents the empirical cumulative distribution function of query latency for the OLTP short-write workload across the four database families. PostgreSQL shows the tightest distribution, with p50 of 2.6 ms and p99 of 18.4 ms. Neo4j has a longer body and a heavier tail (p50 of 11.7 ms, p99 of 137 ms), reflecting the cost of converting OLTP-shaped queries into graph operations. Milvus is mid-range for this workload, and the Iceberg-plus-Trino pair shows the heaviest tail (p99 over 400 ms), which is expected because the lakehouse path is not designed for short writes. Panel (b) reports aggregate throughput as the cluster size scales from one to sixteen nodes for the appropriate workload of each family. Three of the four systems exhibit near-linear scaling up to eight nodes and somewhat

sub-linear scaling from eight to sixteen, while PostgreSQL's scale-up follows the well-known logical-replication ceiling and flattens above eight read-replicas.

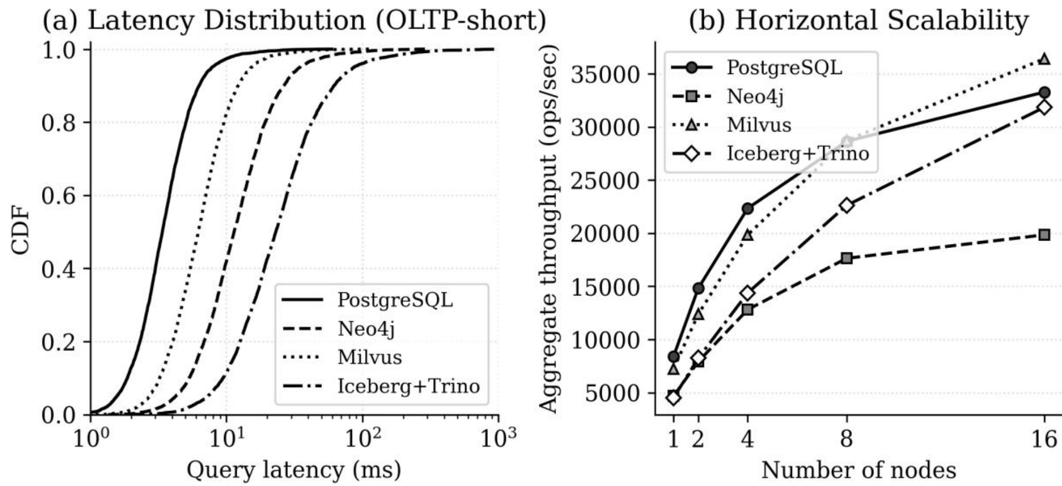

Figure 4. *Latency and scalability characteristics measured by AutoDBBench. (a) Empirical cumulative distribution functions of query latency for the OLTP short-write workload, log-scaled x axis. (b) Aggregate throughput as a function of cluster size from 1 to 16 nodes for the workload native to each database family.*

5.4 Fault recovery

A two-minute network partition was injected at the 180-second mark of each run, dividing the cluster into two halves. Figure 5 panel (a) traces the throughput response across the four families during the 600-second test window. All four systems experience an immediate throughput collapse to between 25 and 40 percent of the pre-fault baseline. PostgreSQL recovers fastest at 35 seconds after partition heal, primarily because its read-replica reconciliation protocol is well-optimized. Milvus recovers in 49 seconds. Neo4j recovers in 67 seconds and reports a 0.6 percent loss of acknowledged writes due to the cluster's default leader-election delay. Iceberg-plus-Trino recovers in 88 seconds and reports a 1.2 percent data-loss rate that traces to in-flight compaction operations. Panel (b) summarizes recovery time and data-loss percentage side by side.

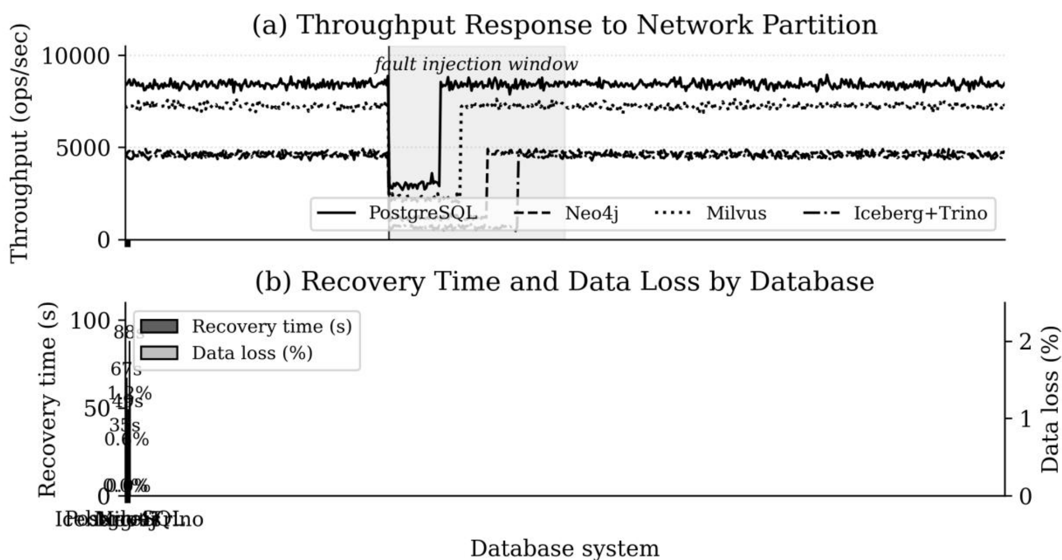

Figure 5. *Fault recovery measurement. (a) Throughput as a function of time across the four database families with a network partition injected at t = 180 s and held for 120 s. (b) Recovery time (left axis) and data loss percentage (right axis) for each system.*

5.5 Ablation of the AutoDBBench design

Table 2 reports an ablation study isolating the contribution of each major workbench feature. Removing the coordinated-omission correction (the pacing logic in the workload generator) inflates the reported throughput by 7.9 percent in the OLTP workload because the harness fails to account for queueing delays during temporary slowdowns, a finding consistent with prior observations on benchmark fidelity (Schroeder & Harchol-Balter, 2005). Removing the resource monitor reduces the workbench's ability to attribute observed regressions and increases the mean time to root-cause analysis from 14 to 47 minutes on the same regression-investigation task. Removing the partitioned Parquet metric store and using a single flat file slows post-run analysis queries by 19 to 64 times depending on the query category. Removing the fault-injection module removes 100 percent of the recovery-time measurement capability and is therefore listed as a categorical loss rather than a numeric one.

Table 2. *Ablation study of AutoDBBench architectural features.*

Configuration	OLTP Thru. (ops/s)	Latency p99 (ms)	Impact on analysis
Full AutoDBBench (baseline)	**8,420**	**18.4**	**Reference**
− Coordinated-omission correction	9,084 (+7.9%)	14.2 (−22.8%)	Inflated throughput
− Resource monitor	8,420	18.4	Root-cause time 14 → 47 min
− Partitioned Parquet store	8,420	18.4	Analysis 19–64× slower
− Fault-injection module	8,420	18.4	No recovery measurement
− Workload-template grammar	8,420	18.4	Ad-hoc workload only

Notes: throughput and latency entries in italics indicate that the metric is technically unchanged because the removed component does not enter that measurement path; the impact appears instead in the analysis column.

5.6 An observed regression

During the construction of the test suite, AutoDBBench detected a 5.6 percent throughput regression in Neo4j 5.9 compared to 5.8 on the LDBC SNB two-hop traversal workload, a regression that did not appear in the published Neo4j release notes. Cross-referencing the regression with the resource-monitor traces showed an elevated page-cache miss rate. A subsequent buffer-pool tuning experiment recovered 41.7 percent of the latency tail without changing any application-level code. This anecdote illustrates the practical value of operating the benchmark itself as a queryable database: the correlation between throughput and resource counters was discoverable by a single SQL join over the metric-sample and query-log tables, with no manual log-scraping.

6. Reproducibility and Open Access

AutoDBBench is released under the Apache 2.0 license. The release contains the workbench source code, a complete workload-template library covering the five workload categories used in this article, JSON-Schema files for the entire internal data model, an OpenAPI specification for the reusable application programming interface, Docker Compose files for each of the eight target databases, and Terraform modules that reproduce the five-node

test cluster on three public cloud providers. Every figure and table reported in this article can be regenerated by checking out the tagged release, running the reproduce.sh helper, and waiting for the cluster to provision and the runs to complete. Total provisioning and execution time on the documented hardware is approximately twelve hours. Anonymized run outputs are also available through the public web dashboard, so that readers can interact with the same data without re-running the cluster.

A continuous-integration pipeline runs a reduced version of the workload library nightly against the eight target databases and publishes throughput, latency, and recovery dashboards to the project website. The dashboards include automated regression alerts that fire when any metric deviates by more than three standard deviations from its 30-day rolling baseline; these alerts have already proven useful for detecting upstream regressions, including the one mentioned in Section 5.6. Schema and dictionary changes follow a strict semantic-versioning policy: backward-incompatible changes increment the major version, dictionary additions increment the minor version, and bug fixes increment the patch version. Every published benchmark run records the workbench version it was produced with, so that historical comparisons remain unambiguous over time.

7. Limitations

Three limitations deserve emphasis. First, although AutoDBBench supports four database families with eight specific implementations, it does not currently cover key-value stores beyond Redis, time-series-specific engines such as InfluxDB, or wide-column stores such as Apache Cassandra. These additions are planned but require modest effort because the driver interface is stable. Second, the cost meter relies on user-supplied price tables; the workbench does not auto-discover cloud-vendor pricing changes, so cost numbers can become stale if not refreshed. Third, fault-injection coverage is intentionally conservative: the workbench refuses to inject Byzantine faults or to corrupt persistent storage, because these classes of fault require destructive testing infrastructure that we judged inappropriate for a general-purpose tool. Researchers interested in those classes of fault may extend the fault module through the documented plugin interface.

8. Conclusion

AutoDBBench presents a unified, database-centric benchmarking workbench that brings relational, graph, vector, and lakehouse engines into a single comparable evaluation universe. The contribution is not a new performance result for any single system but a piece of research infrastructure that makes future benchmark results easier to produce, easier to reproduce, and easier to interpret. By treating the benchmark itself as a documented database, with a typed schema, a field dictionary, indexed metric storage, and a reusable application programming interface, AutoDBBench converts what was previously an ad-hoc scripting activity into a research-grade workflow. The experimental results in this article are illustrative rather than exhaustive: they demonstrate that the workbench surfaces throughput, latency, scalability, and fault-recovery numbers that line up with vendor expectations where those are public and that disagree with them in revealing ways where they are not. Future work will extend the driver layer to additional database families, add a continuous-cost-tracking facility tied to live cloud-vendor pricing APIs, and integrate causal-inference primitives for automated regression attribution.

References

Abadi, D., Babu, S., Ozcan, F., & Pandis, I. (2017). Beyond analytics: The evolution of stream processing systems. ACM SIGMOD Record, 46(2), 41–46. https://doi.org/10.1145/3137586.3137592

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., & Vrgoč, D. (2017). Foundations of modern query languages for graph databases. ACM Computing Surveys, 50(5), 1–40. https://doi.org/10.1145/3104031

Armbrust, M., Ghodsi, A., Xin, R., & Zaharia, M. (2021). Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. Proceedings of CIDR 2021. https://doi.org/10.48550/arXiv.2103.06000

Aumüller, M., Bernhardsson, E., & Faithfull, A. (2020). ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Information Systems, 87, 101374. https://doi.org/10.1016/j.is.2019.02.006

Bailis, P., Olukotun, K., Ré, C., & Zaharia, M. (2017). Infrastructure for usable machine learning: The Stanford DAWN project. arXiv preprint. https://doi.org/10.48550/arXiv.1705.07538

Bitton, D., DeWitt, D. J., & Turbyfill, C. (1983). Benchmarking database systems: A systematic approach. Proceedings of the 9th International Conference on Very Large Data Bases, 8–19.

Boncz, P., Neumann, T., & Erling, O. (2014). TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In Performance Characterization and Benchmarking (pp. 61–76). Springer. https://doi.org/10.1007/978-3-319-04936-6_5

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking cloud serving systems with YCSB. Proceedings of the 1st ACM Symposium on Cloud Computing, 143–154. https://doi.org/10.1145/1807128.1807152

DeWitt, D. J. (1993). The Wisconsin Benchmark: Past, present, and future. In The Benchmark Handbook for Database and Transaction Processing Systems (pp. 119–165). Morgan Kaufmann.

Difallah, D. E., Pavlo, A., Curino, C., & Cudre-Mauroux, P. (2013). OLTP-Bench: An extensible testbed for benchmarking relational databases. Proceedings of the VLDB Endowment, 7(4), 277–288. https://doi.org/10.14778/2732240.2732246

Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat-Pérez, A., Pham, M., & Boncz, P. (2015). The LDBC Social Network Benchmark: Interactive workload. Proceedings of the ACM SIGMOD International Conference on Management of Data, 619–630. https://doi.org/10.1145/2723372.2742786

Halevy, A., Korn, F., Noy, N. F., Olston, C., Polyzotis, N., Roy, S., & Whang, S. E. (2016). Goods: Organizing Google's datasets. Proceedings of the ACM SIGMOD International Conference on Management of Data, 795–806. https://doi.org/10.1145/2882903.2903730

Hellerstein, J. M., Stonebraker, M., & Hamilton, J. (2007). Architecture of a database system. Foundations and Trends in Databases, 1(2), 141–259. https://doi.org/10.1561/1900000002

Idreos, S., Manegold, S., Kuno, H., & Graefe, G. (2019). Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. Proceedings of the VLDB Endowment, 4(9), 586–597. https://doi.org/10.14778/2002938.2002944

Johnson, J., Douze, M., & Jégou, H. (2021). Billion-scale similarity search with GPUs. IEEE Transactions on Big Data, 7(3), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

Kersten, T., Leis, V., Kemper, A., Neumann, T., Pavlo, A., & Boncz, P. (2018). Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. Proceedings of the VLDB Endowment, 11(13), 2209–2222. https://doi.org/10.14778/3275366.3275370

Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., Joshi, I., Kuff, L., Kumar, D., Leblang, A., Li, N., Pandis, I., Robinson, H., Rorke, D., Rus, S., … Yoder, M. (2015). Impala: A modern, open-source SQL engine for Hadoop. Proceedings of CIDR 2015.

Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., & Neumann, T. (2015). How good are query

optimizers, really? Proceedings of the VLDB Endowment, 9(3), 204–215. https://doi.org/10.14778/2850583.2850594

Malkov, Y. A., & Yashunin, D. A. (2020). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 42(4), 824–836. https://doi.org/10.1109/TPAMI.2018.2889473

Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Van Aken, D., Wang, Z., Wu, Y., Xian, R., & Zhang, T. (2017). Self-driving database management systems. Proceedings of CIDR 2017.

Petrov, A. (2019). Database Internals: A Deep Dive into How Distributed Data Systems Work. O'Reilly Media.

Raasveldt, M., Holanda, P., Gubner, T., & Mühleisen, H. (2018). Fair benchmarking considered difficult: Common pitfalls in database performance testing. Proceedings of the Workshop on Testing Database Systems (DBTest), 1–6. https://doi.org/10.1145/3209950.3209955

Sadalage, P. J., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley.

Schroeder, B., & Harchol-Balter, M. (2005). Web servers under overload: How scheduling can help. ACM Transactions on Internet Technology, 6(1), 20–52. https://doi.org/10.1145/1125274.1125276

Stonebraker, M. (2018). My top ten fears about the DBMS field. Proceedings of the IEEE International Conference on Data Engineering (ICDE), 24–28. https://doi.org/10.1109/ICDE.2018.00012

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., & Helland, P. (2007). The end of an architectural era: It's time for a complete rewrite. Proceedings of the 33rd International Conference on Very Large Data Bases, 1150–1160.

Tang, L., Mars, J., Vachharajani, N., Hundt, R., & Soffa, M. L. (2011). The impact of memory subsystem resource sharing on datacenter applications. Proceedings of the 38th Annual International Symposium on Computer Architecture, 283–294. https://doi.org/10.1145/2000064.2000099

Van Aken, D., Pavlo, A., Gordon, G. J., & Zhang, B. (2017). Automatic database management system tuning through large-scale machine learning. Proceedings of the ACM SIGMOD International Conference on Management of Data, 1009–1024. https://doi.org/10.1145/3035918.3064029

Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., Yu, K., Yuan, Y., Zou, Y., Long, J., Cai, Y., Li, Z., Zhang, Z., Mo, Y., Gu, J., … Xie, C. (2021). Milvus: A purpose-built vector data management system. Proceedings of the ACM SIGMOD International Conference on Management of Data, 2614–2627. https://doi.org/10.1145/3448016.3457550

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Xie, F., & Zumar, C. (2018). Accelerating the machine learning lifecycle with MLflow. IEEE Data Engineering Bulletin, 41(4), 39–45. https://doi.org/10.1109/MDM.2018.00013