VeriBench: Analyzing the Performance of Database Systems with Verifiability

Cong Yue
National University of Singapore
yuecong@comp.nus.edu.sg

Meihui Zhang
Beijing Institute of Technology
meihui_zhang@bit.edu.cn

Changhao Zhu
Beijing Institute of Technology
zhuchanghao@bit.edu.cn

Gang Chen
Zhejiang University
cg@zju.edu.cn

Dumitrel Loghin
National University of Singapore
dumitrel@comp.nus.edu.sg

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.sg

ABSTRACT

Database systems are paying more attention to data security in recent years. Immutable systems such as blockchains, verifiable databases, and ledger databases are equipped with various verifiability mechanisms to protect data. Such systems often adopt different threat models, and techniques, therefore, have different performance implications compared to traditional database systems. So far, there is no uniform benchmarking tool for evaluating the performance of these systems, especially at the level of verification functions. In this paper, we first survey the design space of the *verifiability-enabled database systems* along five dimensions: threat model, authenticated data structure (ADS), query processing, verification, and auditing. Based on this survey, we design and implement VeriBench, a benchmark framework for *verifiability-enabled database systems*. VeriBench enables a fair comparison of systems designed with different underlying technologies that share the client-side verification scheme, and focuses on design space exploration to provide a deeper understanding of different system design choices. VeriBench incorporates micro- and macro-benchmarks to provide a comprehensive evaluation. Further, VeriBench is designed to enable easy extension for benchmarking new systems and workloads. We run VeriBench to conduct a comprehensive analysis of state-of-the-art systems comprising blockchains, ledger databases, and log transparency technologies. The results expose the weaknesses and strengths of each underlying design choice, and the insights should serve as guidance for future development.

PVLDB Reference Format:
Cong Yue, Meihui Zhang, Changhao Zhu, Gang Chen, Dumitrel Loghin, and Beng Chin Ooi. VeriBench: Analyzing the Performance of Database Systems with Verifiability. PVLDB, 16(9): 2145 - 2157, 2023.
doi:10.14778/3598581.3598588

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yc1111/VeriBench.

Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598588

1 INTRODUCTION

The advancement of Information Technology has determined many organizations to adopt digital transformation and shift their core business to the cloud. Digitalization helps organizations in improving their business model, enhance collaboration, and increase productivity. However, such digitization also increases the exposure to various threats from internal and external adversaries such as tampering of data, ill-intended content, or actions from malicious collaborators. Hence, there is an increasing demand to protect data security in modern database systems.

A wide range of systems has been equipped with *verifiability* to protect their data. Such systems have various focuses and use different techniques. Verifiable databases [7, 8, 29, 38, 42, 43] guarantee the correctness of query execution and protect the latest state of the database by adopting verifiable computing. The computation is offloaded to an untrusted server, which will execute and generate verifiable results by using cryptographic techniques. The advantage of this approach is that the client will have a constant proof size and lightweight verification burden. However, the server runs expensive cryptographic computations. Certificate transparencies [17, 18, 23, 32] store the keys and certificates, therefore, they focus on the data content and modified history. Merkle trees [24] are constructed over the data to detect any tampering after the data is stored. They expose simple storage API, and are not suitable for update-intensive tasks. Blockchains [5, 10, 37] serve as secure transaction systems, protect both the data and the entire history, and ensure the serializability of the transactions. Blockchains maintain a sequence of hash-chained blocks, and the integrity is guaranteed by replicating the blocks using a Byzantine fault tolerant protocol (e.g., PBFT [9]). Despite the strong security guarantee offered by the blockchains, they suffer from low performance due to the expensive consensus protocols.

In recent years, ledger databases [4, 6, 39, 40] have gained traction due to features such as protecting the integrity of data, history and query results. A ledger database maintains data or logs in the form of an append-only ledger, which can generate proofs for users to verify the integrity. Compared to conventional databases, a ledger database has three main advantages. First, it provides efficient verification. The users only need to maintain a cryptographic hash, called digest, of the ledger and the integrity check can be performed by comparing the digest against the reproduced hash computed from the proof. Second, it reduces the surface of misbehavior. Since the ledger is immutable, all historical data are protected by the digest. Adversaries cannot tamper with history without changing the hashes. Third, it can be publicly verified. Everyone can verify the integrity of the data or the log with the ledger. Compared to blockchains, ledger databases offer high performance without the performance bottleneck at the consensus layer. Ledger databases are therefore suitable for maintaining financial transactions, logistic orders, and healthcare data where the integrity of data evolution history and proof of data lineage are important.

Despite the fact that many *verifiability-enabled database systems* have been designed and implemented in recent years, there is no uniform benchmarking tool to systematically and fairly evaluate existing systems. Traditional database benchmarks such as TPC [21] and YCSB [11] do not consider the verifiability-related features of the systems, therefore, only provide an overall evaluation of the performance. The effects of verifiability-specific design choices are unknown or require extensive adaptation to evaluate. Existing blockchain benchmarking frameworks such as Caliper [19] and BlockBench [14] focus on the consensus and smart contract execution, and cannot be used by other systems that do not support smart contracts. Therefore, it is necessary to build a benchmarking framework for *verifiability-enabled database systems*.

To build such a benchmarking framework, we have to consider three key system issues of *verifiability-enabled database systems*. First, we need to consider the design space and evaluate the effects of design choices on performance. Second, the behavior of different systems can vary significantly. For example, Amazon Quantum Ledger Database (QLDB) [4], blockchains, transparency logs, and some verifiable databases provide on-demand verification. On the other hand, GlassDB [40], Litmus [38], and Concerto [7] enforce continuous verification during the transaction processing. Third, GlassDB, LedgerDB [39], SQL Ledger [6], and Concerto adopt deferred verification, where verification is performed after a time period with a batch of data. All the systems have to be evaluated in a systematic manner that is compatible with different scenarios.

In this paper, we propose VeriBench, a benchmarking framework for *verifiability-enabled database systems*. We address the first key issue by conducting a survey on the design space covered by existing systems. We categorize the design of *verifiability-enabled database systems* based on five components, namely threat model, authenticated data structure (ADS), query processing, verification and auditing. We discuss the design choices of each component and subsequently design VeriBench with verification-aware workloads. Specifically, we include micro-benchmarks to evaluate performance impact on single component, and macro-benchmarks adapted from a key-value benchmark (YCSB) and two OLTP benchmarks (SmallBank and TPC-C) for system-level performance. Lastly, we conduct extensive experiments on existing systems to illustrate the strengths and weaknesses of each design and system, which should be useful for the future development of *verifiability-enabled database systems*.

In summary, we make the following contributions.

- We analyze the design space of *verifiability-enabled database systems* along five dimensions: threat model, authenticated data structures, query processing, verification, and auditing. We discuss how existing systems fit in the design space.
- We implement VeriBench, an open-source benchmarking framework for *verifiability-enabled database systems* which can be used directly or extended to evaluate newly proposed databases.
- We conduct extensive benchmarking and performance analysis of QLDB [4], LedgerDB [39], SQLLedger [6], GlassDB [40], Merkle^2 [18], and Confidential Consortium Framework (CCF) [31]. Our results pinpoint the performance bottlenecks of existing *verifiability-enabled database systems*, and show the advantages of various design choices.

2 DESIGN SPACE ANALYSIS

2.1 State-of-the-art Systems

Amazon Quantum Ledger Database (QLDB) [4] is a ledger database managed in a centralized way by Amazon, on its Amazon Web Services (AWS) cloud. QLDB keeps two tables, one for the current states and another one for the history of the states, as shown in Figure 1. These tables are connected to an append-only ledger implemented with a Merkle tree [24]. This tree contains the hashes of all the states and it is updated every time there is a change in a state's value. This enables instant verification at the expense of lower performance. QLDB does not support external auditors.

LedgerDB was developed as a research project [39] and it is now offered as a service by Alibaba on its cloud [1]. LedgerDB keeps a Merkle tree that is updated in batches using a method called batch-Accumulated Merkle Tree (bAMT). Furthermore, the copy-on-write technique is used to append transactions, thus, reducing the contention in the ledger layer. Skip list indexes (clue indexes) are built over the keys, where the size of each such index is stored in a leaf of a Merkle Patricia Trie (MPT), as shown in Figure 2. A skip list element points to the ledger-stored transaction that modified the corresponding key. LedgerDB implements CFT and workload balancing using a master-workers architecture for its distributed version. It supports auditors and implements deferred verification.

SQLLedger [6] is part of Microsoft SQL Server and it is offered as a service on the Microsoft Azure cloud [26]. It uses Microsoft SQL Server at the storage layer on top of which two tables are stored, one for the ledger and one for the history of the states (or rows). The ledger consists of blocks and each block contains two types of Merkle trees, as shown in Figure 3. The first type is constructed over all the transactions captured in the block. The second Merkle tree type keeps track of the rows updated by a transaction. Hence, the leaves of the transaction Merkle tree store the root hash of the row Merkle trees. SQLLedger implements deferred verification, where clients can send batches of verification requests after the operations are completed. It also supports auditors.

GlassDB [40] is a distributed ledger database system proposed in a recent research project. The system partitions the data into different shards and adopts two-phase commit (2PC) to guarantee the atomicity of cross shards transactions. It applies replication to tolerate node failures. As shown in Figure 4, each shard has a transaction manager, a verifier, and ledger storage. GlassDB introduces a two level POS-tree [41] as the authenticated data structure to improve the efficiency of data access and proof generation. In particular, the lower level POS-tree builds on top of the database states in lexical order, while the upper level POS-tree builds over the entire history root hashes of the lower level POS-tree in chronological order. GlassDB adopts asynchronous ledger updates and batching of transactions to accelerate transaction processing, and applies a deferred verification approach to further improve verification.

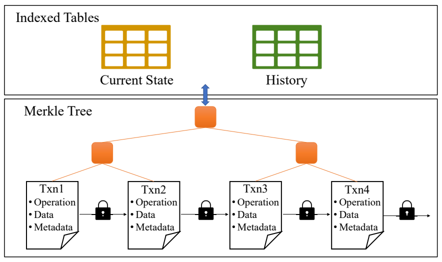


Figure 1: QLDB architecture

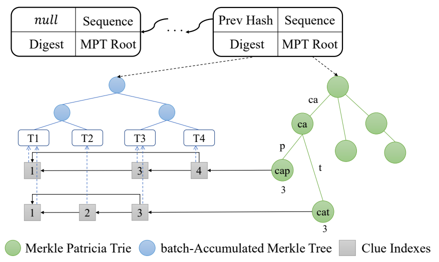


Figure 2: LedgerDB architecture

Confidential Consortium Framework (CCF) [25, 31] (previously known as Coco Framework) is an open-source permissioned (private) blockchain project developed by Microsoft. CCF consists of both public tables and private tables to store data and configurations in key-value format. Like other blockchain systems, CCF persists the history of the key-value data in a tamper-proof encrypted ledger that is protected by Merkle trees. This ledger enables the replay of the state transitions. However, the integrity of each operation on the states is guaranteed by the execution inside a trusted execution environment (TEE). Figure 5 shows the architecture of CCF. All nodes run the same application inside a TEE enclave. Changes are applied to the storage when a majority of the nodes reach agreements. The fully replicated ledger guarantees stronger security consumption with higher performance costs.

Merkle^2 [18] is a research project that designs a transparency log system for low-latency and secure public key storage. As shown in Figure 6, it combines chronological Merkle trees and prefix Merkle trees to obtain the append-only property and to execute the id-based query with low costs, by storing the prefix tree roots in the internal nodes of the chronological trees. The server maintains a chronological forest and periodically publishes the digests to the auditors. Users have to request the digests from the auditors to verify the integrity of the query results.

Next, we describe the design choices made by these systems in five key directions, namely, threat model, authenticated data structure (ADS), query processing, verification, and auditing. A summary of these design choices is presented in Table 1.

2.2 Threat Model

There are two main threat models adopted by *verifiability-enabled database systems*. Firstly, decentralized systems such as blockchains assume they run on fully replicated machines hosted by multiple parties. Each node may behave in a Byzantine manner. These systems have the strongest security assumption on tolerating at most f Byzantine failures with $3f + 1$ replicas while behaving correctly with the help of Byzantine fault tolerant (BFT) protocols, such as PBFT [9]. However, the BFT protocols are costly in terms of execution time and communication, and they do not scale well.

The second threat model assumes a single-party service provider, which can be compromised. The adversaries have full control of the host machine, including altering the network messages and tampering with the data contents. Users can detect any misbehaviour instead of preventing it. This model is adopted by certificate transparencies, ledger databases, and verifiable databases with different use cases. Certificate transparencies protect their data and history from being tampered once stored. Ledger databases verify the correctness of transactions in addition to the data contents. Verifiable databases offer a similar guarantee as ledger databases, except they only protect the current state of the database.

Distributed ledger databases and verifiable databases implement CFT replication to tolerate node failures, in contrast to typical blockchains that use BFT consensus protocols. CCF design [31] supports both CFT and BFT replication, but the currently available implementation [25] only supports CFT.

2.3 Authenticated Data Structures

2.3.1 Ledger. The ledger is the key data structure of blockchains, certificate transparencies, and ledger databases. The ledger is usually a Merkle tree variance. For example, blockchains [5] construct Merkle trees over the world state and transactions for validation. QLDB builds a Merkle tree over the hashes of the transactions as shown in Figure 1. SQLLedger [6] constructs a Merkle tree over the modified data for each transaction, and another Merkle tree over the transaction entries batched in a block. The root hash of the latter Merkle tree is stored in the block entry with the previous block entry's hash to form a hashed chain as shown in Figure 3. LedgerDB [39] adopts a batched accumulated Merkle tree, which employs copy-on-write when new transactions are appended to reduce the contention as shown in Figure 2. To improve the efficiency of verification for the latest versions, Merkle^2, as illustrated in Figure 6, constructs a forest of full Merkle trees over the data in chronological order, and each internal node stores the root hash of a prefix tree built over the data in lexical order. GlassDB replaces the Merkle trees with a two-level POS-trees [41]. A POS-tree is built on top of the database states in lexical order, while the second POS-tree is built over the entire history in chronological order.

2.3.2 Chronological Order vs. Lexical Order. Systems such as QLDB, LedgerDB, SQLLedger, and CCF construct ADS over data in chronological or transaction order. The ADS is used only for integrity proof, and separate index structures are required to query the data. This causes two problems: (1) updating and proof generation become slower when the ADS grows larger, and (2) it requires additional protection of the indexes, e.g., LedgerDB uses a Merkle Patricia Trie (ccMPT) to protect its clue indexes, while the index tables in QLDB and SQLLedger are not hash-protected, leading to the inability to guarantee that the value is the latest. In contrast, systems like GlassDB, Trillian, CONIKS, and Merkle^2 embed additional Merkle tree-based ADS in lexical order, thus, protecting the indexes.

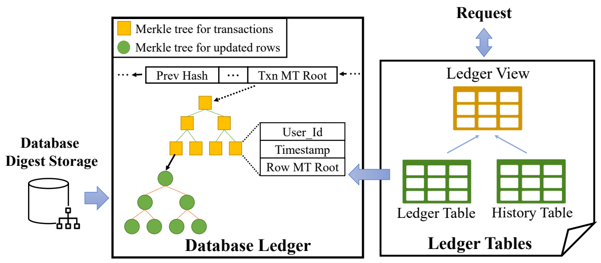


Figure 3: SQLLedger architecture

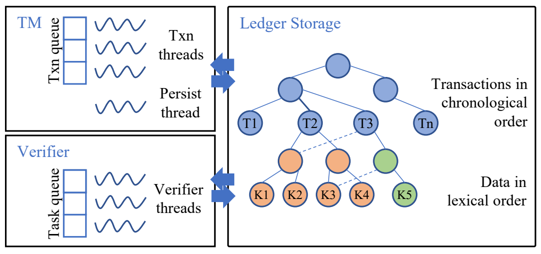


Figure 4: GlassDB architecture

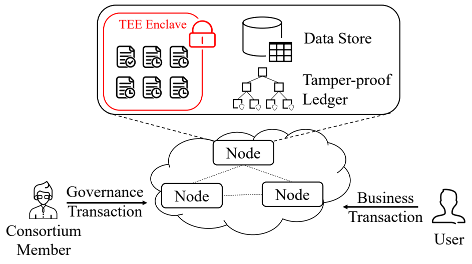


Figure 5: CCF architecture

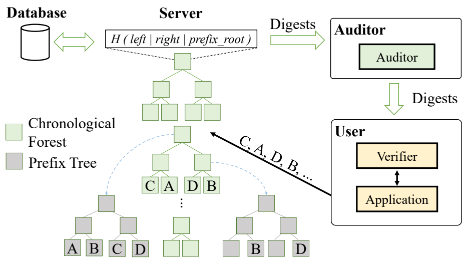


Figure 6: Merkle^2 architecture

2.4 Query Processing

2.4.1 Abstraction. **Key-value vs. relational.** These are the two main data abstraction models used by modern databases. In consequence, *verifiability-enabled database systems* support one of the two models. QLDB [4] and SQLLedger [6] employ relational data abstraction by building additional relational tables and views on top of the ledger. Users can query the data through SQL queries. However, the tables and views are not hash protected, and therefore, they may be tampered or simply return stale data. As shown in Table 1, other systems expose key-value data abstractions. While it is possible to build a relational database on top of a key-value store, additional protection is required for indexing and query, which may cause significant overhead.

Non-transactional vs. transactional. Typical database systems provide transactional abstraction with ACID properties. Verifiable databases and ledger databases follow this design decision. Blockchains, which target exchanging digital assets or general business logic, also provide transactional abstraction. Certificate transparency logs, such as Merkle^2, serving as storage for public keys and certificates, only expose non-transactional abstraction.

2.4.2 Batching. The update of a ledger is costly since it entails cryptographic hash derivation, and incurs high contention, especially at the root node, affecting the implementation of parallelism. QLDB and Merkle^2 implement the ledger update after each individual operation. To reduce the cost of hash derivation and mitigate the read/write contention against proof generation operations, LedgerDB updates its ADS in batches of transactions and adopts copy-on-write when updating the ADS. SQLLedger constructs an individual Merkle tree for each transaction and block, making it contention-free when appending new blocks to the ledger. Similar to LedgerDB, SQLLedger batches multiple transactions in a block to reduce the overhead of calculating block-level hashes. In GlassDB, the transactions from a batch that update the same keys are arranged in a sequence of blocks, based on the number of times the keys are updated. Copy-on-write is applied to reduce contention. Likewise, blockchains batch transactions into one block to increase the throughput. They use either a predefined size or block time to control the size of the batch. Since the bottleneck for blockchain is consensus protocol, the larger the block size, the higher the throughput and latency, and vice versa.

2.4.3 Data partitioning. In the distributed setup of *verifiability-enabled database systems*, data partitioning (or sharding) is often used to improve the performance. It reduces the burden on each node and increases the parallelism. However, an atomic commit protocol needs to be applied to guarantee the ACID properties of distributed transactions. The scalability of such protocol and partitioning strategy is essential to performance.

2.5 Verification

To guarantee the integrity of data and query results, *verifiability-enabled database systems* provide proofs that can be publicly verified by the users or third-party verifiers for each request. The proofs typically consist of a digest which is a hash summary of the database states and a proof generated from ADS. The client, upon receiving the proof, can reproduce the digest. Then, it verifies data integrity by comparing the original and reproduced digests.

2.5.1 On-demand vs. continuous. Verification can be categorized into *on-demand* and *continuous* according to different user scenarios. QLDB, SQLLedger, and CCF only expose on-demand verification API, i.e., they will perform the verification when users specifically request or find any inconsistencies. However, such verification cannot guarantee the system is always in a consistent state. It may continue operating with the incorrect states and incur significant

Table 1: Design space characterization of state-of-the-art *verifiability-enabled database systems.*

Design Dimension	Sub-dimension	QLDB [4]	LedgerDB [39]	SQLLedger [6]	GlassDB [40]	CCF [31]	Merkle^2 [18]
Threat Model	Administration	centralized	centralized	centralized	centralized	decentralized	centralized
	Fault Tolerance	CFT	CFT	CFT	CFT	CFT / BFT	single-node
Authenticated Data Structures	Ledger	Merkle Tree	Merkle Tree	Merkle Tree	POS-tree	Merkle Tree	Merkle Tree
	Order	chronological	chronological	chronological	hybrid	chronological	hybrid
Query Processing	Abstraction	relational	key-value	relational	key-value	key-value	key-value
		transactional	transactional	transactional	transactional	transactional	non-transactional
	Ledger Update	individual	batch	batch	batch	batch	individual
	Partitioning	partitioning	partitioning	partitioning	partitioning	no partitioning	no partitioning
Verification		on-demand	deferred	on-demand	deferred	on-demand	deferred
Auditing		user	auditor	auditor	auditor	user	auditor

loss if verification is not performed timely. In contrast, LedgerDB, GlassDB, and Merkle^2 implicitly call the verification API during transaction processing to ensure every transaction is executed correctly and the system is in a correct state. To achieve this, the systems need to perform extensive verification operations that will impact the performance.

For continuous verification, it is expected that providing proof for each operation is costly. LedgerDB and GlassDB return a promise containing the data and block sequence the data resides to the client for future verification on the completion of an operation. The ADS is updated asynchronously. The client can batch the proof generation request and verification process for higher performance. However, there is a trade-off between performance and security: there is a verification time window when the integrity of data could be temporarily violated. In CCF, the proof can be generated only after the nodes reach consensus and commit a block. Merkle^2, instead, updates the ADS immediately and delegates the verification to a background verifier process.

2.6 Auditing

Systems such as certificate transparencies (e.g., Merkle^2), GlassDB, LedgerDB, and SQLLedger rely on auditors to check the consistency of the ledger and detect malicious behaviors. The auditors will rebuild the ledger based on the logs returned by the system, and compare the digest of the rebuilt ledger with the digest returned by the system. Auditors will notify the users if any mismatch is found. The auditing process is typically expensive. Any third-party entities or powerful users can play the role of the auditor. The systems require at least one honest auditor so that any malicious behaviours can be detected and notifications can be sent to the users. QLDB and CCF do not rely on auditors to check the consistency of the ledger. Consequently, users have to check the consistency of the ledger if required. In fact, typical blockchains rely on the users (or miners) to verify the ledger.

3 VERIBENCH

In this section, we describe the design and implementation of VeriBench, our benchmarking framework for *verifiability-enabled database systems.*

3.1 Architecture

As shown in Figure 7, VeriBench consists of a configuration loader, a workload interface, a database adapter, a benchmark driver, and a log analyzer. The configuration loader loads the parameters of the benchmarks. The workload interface defines task generation and execution API. The database adapter connects to the target database system under evaluation through generic database APIs. The benchmark driver runs the benchmarks with the configuration loader, workload interface, and database adapter. It logs the performance statistics, which will be processed by the log analyzer to obtain the final results. To improve its usability and generality, VeriBench only exposes generic interfaces defined in the workload interface and database adapters.

3.2 API

In this section, we describe the API of the workload generator and database adapter. Users can include customized workloads and *verifiability-enabled database systems* by inheriting these interfaces. First, we describe the API of the workload generator as follows.

`NextTask(conf)` is a function that takes the workload configuration as input and generates the next task including the operation type and a list of parameters. The configuration usually contains the ratio, distribution, and ranges of operations, keys and values. For example, a configuration of SmallBank workload includes $D(operation) = uniform, D(account) = uniform, R(account) = [0, 50000], R(ammount) = [0, 500]$, where D is the distribution, and R is the range. The interface returns the generated tasks, e.g., $< SendPayment, 1, 2, 100 >$, which represents account 1 paying \$100 to account 2.

`ExecuteTransaction(task, db)` is a function that takes the task generated by `NextTask` and the database adapter, `db`, as the input and returns the status of execution. The inherited function implemented by the user shall execute the transaction with a sequence of `Put`, `Get`, and `Verify` operations provided by the database adapter.

Next, we introduce the API of the database adapter. Depending on the database under evaluation, the database adapter may be implemented in different programming languages.

`Put(keys, values)` defines the operation to update or insert a list of keys and values in the database. It will return the proof optionally for databases that do not support deferred verification.

`Get(keys)` defines the operation to get the values of a list of keys from the database. It will return the proof optionally for databases that do not support deferred verification.

`Verify(keys, block_seqs)` is for deferred verification, where a batch of keys could be verified together. The parameters represent a list of keys and the block sequences where the keys are located.

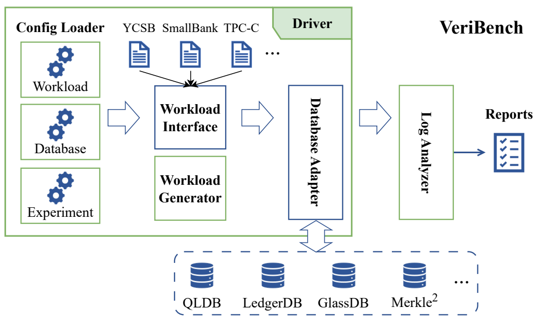


Figure 7: VeriBench architecture

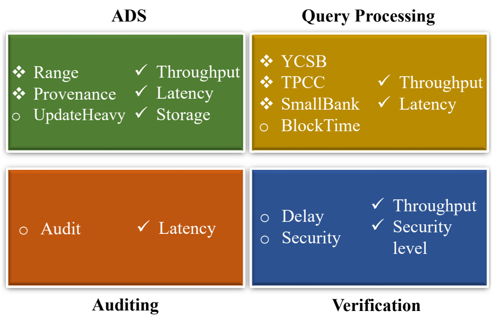


Figure 8: Workloads and metrics of each component

Table 2: Configuration parameters used by VeriBench.

Category	Parameter
Workload	*workload_type*, *config_path*, *init_data_path*
Database	*database_name*, *db_config_path*
Benchmark	*n_shard*, *n_replica* ,*n_client*, *n_thread* *request_rate*, *duration*, *delay*, *block_time*

The inherited function shall get the proofs from the database, verify the proofs, and return the verification result.

`Verify(proof)` is for immediate verification, where the input `proof` can be obtained from the `Get` and `Put` operations. The interface returns the verification result.

`Begin()` is the initialization function for interactive transactions. Users can set a transaction context within this function such as transaction ID or timestamp. A sequence of `Get(.)` and `Put(.)` functions can be called after this.

`Commit()` defines the commit operation for interactive transactions. It is called after a sequence of `Get(.)` and `Put(.)` operations. It optionally returns the promise in case of deferred verification.

`StoredProcedure(type, params)` is used when the target database under evaluation does not expose interactive transaction API. For example, the transactions are defined as smart contracts in a blockchain and stored procedures in CCF. The `StoredProcedure(.)` function takes the transaction type and list of parameters as input, and executes the transaction logic accordingly. The function optionally returns a promise in case of deferred verification.

To extend VeriBench with a new workload, users need to implement the `NextTask` and `ExecuteTransaction`. To support a new database, users need to implement the `Put`, `Get`, and `Verify`.

3.3 Implementation

Configuration Loader is responsible for loading all experiment-related configurations at runtime. It reads the parameters from the command line and configuration files. The configurations include workload configurations, database configurations, and experiment configurations. The detailed configurations are shown in Table 2.

Workload Interface defines the workload's operations, parameters, generation, data initialization, and execution logic. The detailed API can be found in Section 3.2. VeriBench provides default workload implementations for YCSB, TPC-C, and SmallBank. Users can easily add customized workloads by implementing the interfaces. The workload can be initialized with an optional configuration file, specifying any workload-specific parameters such as the number of warehouses for TPC-C, the number of accounts for SmallBank, etc. The parameters are then used in generating the workload and executing the queries.

Database Adapter defines common database operations, transaction semantics, and verification logic. The full API list can be found in Section 3.2. To implement a database adapter for a new system, users need to build a wrapper class containing the original database client, and implemented the defined API by calling the client functions. For systems that adopt deferred verification, an empty promise interface is provided. This is because the promises across systems can differ significantly. Therefore, users need to define their own data structure and deferred verification logic.

Benchmark Driver is executed by all the client processes. To start the benchmark execution, the driver first initializes the workload and database adapter with the given configurations. It spawns threads to generate the workload at the specified request rate. The generated tasks are stored in a queue. Next, the driver spawns a query execution thread, which continuously fetches the task at the beginning of the queue and calls the database adapter to process the query. In the case of online verification, VeriBench enforces the invocation of the $Verify(.)$ function after each query execution. In the case of deferred verification, VeriBench spawns a verification thread, which periodically invokes the $Verify(.)$ function with a promise as input.

Log Analyzer collects and processes the logs from all the nodes, after all benchmark drivers finished the execution. Measurements are taken and logged before and after the query execution and verification. The log analyzer aggregates all these measurements and calculates the corresponding metrics for each workload.

3.4 Metrics

Throughput. The overall throughput of *verifiability-enabled database systems* is measured to evaluate the performance of query and transaction execution.

Latency. VeriBench measures three types of latency. First, we measure the end-to-end latency of the systems to evaluate the query and transaction execution. Second, we measure the client verification latency to evaluate the efficiency of the proofs generated by the ADS. Lastly, we evaluate the latency of verification and

Table 3: Security Level.

Level	Description	Requirement
Level 1	Verifiable Read	(1) Inclusion proof
Level 2	Verifiable Write	(1) Inclusion proof (2) Append-only proof
Level 3	Verifiable Transaction	For each transaction (1) Inclusion proof (2) Append-only proof (3) Current-value proof
Level 4	Secure Transaction	Protect transaction and data

auditing requests to evaluate the efficiency of ADS. For deferred and batched verification, we further count the number of keys being verified for each verification request to evaluate the efficiency of batching.

Scalability. For distributed setups, we measure the throughput and latency with an increasing number of nodes to evaluate how the systems scale out. Similarly, for decentralized setups, we measure the throughput and latency with increasing replicas to evaluate the network message overhead.

Storage. We measure the storage consumption of *verifiability-enabled database systems* to evaluate the space efficiency of different authenticated data structures and techniques.

Security. We assess the security of *verifiability-enabled database systems* by categorizing them into four security levels as shown in Table 3.

- **Level 1.** The databases support verifiable read operations. Inclusion proofs are generated to prove the existence of data in the entire history.
- **Level 2.** The databases support verifiable write operations. The systems are required to generate the inclusion proofs and append-only proofs to verify the correctness of updated data and validate the updated digest, respectively.
- **Level 3.** The databases support verifiable transactions. To achieve this, the databases must verify (1) the data in the read set is the latest, (2) the data in write set exists in the new database state, and (3) the states before and after the transaction are consecutive. Databases must support continuous verification to guarantee all transactions are committed correctly.
- **Level 4.** While databases in the former levels detect the malicious behaviors, databases in this level aims to prevent malicious behaviors from happening by using consensus protocols such as PBFT and proof-of-work, or secure hardware such as Intel SGX and Keystone Enclave.

3.5 Workloads

To facilitate a comprehensive evaluation of *verifiability-enabled database systems*, we include macro-benchmarks adapted from well-known key-value store and OLTP benchmarks, as well as micro-benchmarks targeting the design choices described in Section 2. Due to space limitations, in this paper, we focus on OLTP workloads since these are the primary target of the *verifiability-enabled database systems*. In future work, we will add OLAP workloads and *verifiability-enabled database systems* that are optimized for OLAP to VeriBench. Note that users can easily extend VeriBench with more workloads following the API described in Section 3.2.

3.5.1 Macro-benchmark workloads.

YCSB [11] is extensively used to evaluate key-value stores, be it on-premise or on the cloud. We use it in VeriBench to evaluate *verifiability-enabled database systems* that expose key-value abstractions. To work with *verifiability-enabled database systems*, we adapt the original workload to enable verification for each operation. For the systems adopting continuous verification, the verification phase is already incorporated in the query execution. On the other hand, for systems only adopting on-demand verification, we enforce the invocation of $Verify(.)$ for the key involved in each operation.

SmallBank [3] is a light-weight OLTP workload. There are two tables in our SmallBank workload, namely *saving* and *checking*, to simulate bank services such as querying balances, depositing and transferring money, and amalgamating assets among bank accounts. Each transaction consists of two to four read and write operations. Similarly, we implement all six transaction types in the context of verifiable databases, i.e., each transaction will return the corresponding proof to verify the integrity of the execution.

TPC-C [21] is a more computation-heavy OLTP workload simulating transactions in e-commerce and supply-chain systems. In VeriBench, we optimize the schema for key-value database systems by separating the update-heavy columns from the read-only columns. This significantly reduces the unnecessary conflicts caused by the key-value representation during transaction processing. Moreover, this still guarantees the serializability. Similarly to SmallBank, VeriBench enforces integrity verification after the commit of each TPC-C transaction.

Range query is implemented in VeriBench to evaluate the effectiveness of indexing. This workload queries a random range of keys. In our experiments, the queried keys follow a uniform distribution. Besides the corresponding values, we also request a promise, which is used later to validate the integrity of the ledger.

Provenance is used to evaluate the ability of a system in providing data provenance. Data provenance shows the historical versions of one entry (key) and its origin. It can be used in application-level data audits and it represents an important way to maintain the integrity of data entries. We evaluate data provenance by issuing read-only queries that fetch the latest k versions of a specified key. If k is greater than the total number of historical versions, the entire trace of data alterations is returned.

3.5.2 Micro-benchmark workloads.

Verification workload is used to evaluate the efficiency of verification functions including deferred verification and batching. We generate batches of read and write operations, respectively, and test the performance of verification under different delay and block time. **UpdateHeavy** workload is used to evaluate the update performance and storage consumption of the ADS in the target *verifiability-enabled database systems.* **Audit** workload is used to evaluate the cost of auditing. It contains sequentially generated block sequence numbers, which will be assigned to auditors to perform the audit. **Security** workload is used to validate whether the requirements of each security level, as defined in Table 3, are fulfilled by the systems. For inclusion proof, we initialize the databases with 1,000 keys and then perform get operations on random keys.

Lastly, we issue verification requests on the keys with tampered values. Databases need to detect all errors to pass the test. For current-value proof, we initialize the databases with 1,000 keys, where each key has two versions of values. We conduct the verification with the latest digest and the first version value. Databases need to detect all stale data to pass the test. For append-only proof, we first initialize the databases with 1,000 keys and request for a digest d. We then reload the database with a different set of keys and get a digest d'. Lastly, we conduct verification using d and d'. The verification should detect that d is not a prefix of d'.

4 EVALUATION

In this section, we use VeriBench to benchmark six state-of-the-art *verifiability-enabled database systems*, namely, QLDB, LedgerDB, SQLLedger, GlassDB, CCF, and Merkle^2. The systems represent different design choices and have varied backgrounds. QLDB [4] and SQLLedger [6] are commercial ledger database systems offered by Amazon Web Services and Microsoft, respectively. LedgerDB [39] is an industrial prototype of a ledger database implemented by Alibaba. GlassDB is a high-performance ledger database system from a recent research project [40]. CCF [31] is a permissioned blockchain framework built by Microsoft for high available and high performance applications. Lastly, Merkle^2 [18] is a transparency log system for low-latency and secure public key and certificate storage. We evaluate the systems with both the macro-benchmarks and micro-benchmarks described in Section 3.5.

4.1 Implementation

Ledger Databases. We use the source code of **QLDB**, **LedgerDB**, **SQLLedger**, and **GlassDB** provided by the GlassDB paper [40]. QLDB, LedgerDB, and SQLLedger are commercial systems with closed-source code. As the performance of these systems offered as cloud services may be affected by factors such as hardware and software configurations and policies, we choose the open-source re-implementation to compare the four systems in the same environment to preclude the performance impact of engineering optimizations, and therefore, facilitate a fair comparison. The data is partitioned according to the hash of a key. 2PC is used to ensure atomicity during the commit. Each partition is replicated to multiple nodes using the Viewstamp Replication Protocol for failure recovery. Each partition maintains a separate ledger to provide verifiability. The systems use different ledger structures according to their designs. The implementation of QLDB and SQLLedger omit the SQL layer, which brings additional overhead. Instead, they provide a key-value store data abstraction. All systems are implemented in C++ and use *protobuf* and *libevent* for serialization and communication, respectively. The systems are integrated with VeriBench using the interactive transactional API. LedgerDB, SQLLedger, and GlassDB implement the deferred verification API, while QLDB implements the online verification API. Though the original QLDB and SQLLedger adopts on-demand verification, we enforce continuous verification for QLDB and SQLLedger by calling the verification function after each transaction.

CCF is an open-source [25] framework provided by Microsoft, allowing users to build applications on it with decentralized trust and centralized computation. To integrate it with VeriBench, we build a key-value store application. CCF allows users to invoke application endpoints through HTTP requests, and a transaction is created for all atomic interactions with the application states within each endpoint invocation. Therefore, CCF is integrated with VeriBench through the stored procedure API, with all workload execution functions defined as endpoints in CCF. CCF adopts on-demand verification. In our experiments, we partition the verification tasks based on the commit sequence and let the clients verify collectively.

Merkle^2 code [35] is provided with the original paper [18]. The system allows clients to search and append public keys for users. In YCSB, the key is a user and the value is the public keys to be appended. Merkle^2 stores the master key of each user in memory for verification purposes. Consequently, each VeriBench client can only deal with a fixed set of users. To run the benchmark with multiple clients, we partition the workload based on key ranges. Merkle^2 is implemented in Go. To integrate it with VeriBench, we build a static library from the Go code to be used by a C++ client. We only implement the $Put(.)$ and $Get(.)$ API, as it exposes the key-value abstraction. The verification is performed asynchronously by a separate verifier server.

Hereinafter, we use the names of these six systems when referring to the performance of the above implementations.

4.2 Experimental Setup

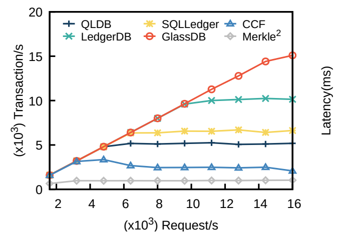
All experiments are conducted on a cluster of 32 nodes. Each node is equipped with a 10-core 3.7GHz Intel Xeon W-1290P CPU, each core supporting HyperThreading, 128GB RAM, and 2TB SSD. The nodes run Ubuntu Server 20.04 operating system and are connected to each other with 1Gbps bandwidth network. Clients and servers are started on different machines to avoid local data contention. For each experiment, we first load the system with initial data. After the system becomes stable, we run each experiment for two minutes. We repeat each experiment five times and present the average.

4.3 Macro Benchmarks

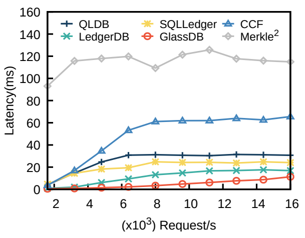
4.3.1 YCSB. We use three YCSB workloads to evaluate the system performance: (1) a *read-heavy* workload, consisting of 80% read operations and 20% write operations, (2) a *balanced* workload, consisting of 50% read operations and 50% write operations, and (3) a *write-heavy* workload, consisting of 20% read operations and 80% write operations.

Performance on a single node. Since Merkle^2 only supports a single-node setup and key-value abstraction, we use YCSB to evaluate the system performance on a single node. For the rest of the systems, to fit into the single-node setup, we disable the replication and set the number of partitions to one. We start 120 VeriBench clients to execute the queries at a request rate ranging from 1,600 to 16,000 requests per second.

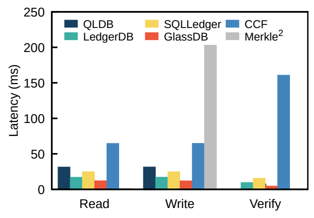
We first run the balanced workload and depict the results in Figure 9a, 9b, and 9c. Figure 9a shows the throughput of the systems. We observe that ledger database systems outperform CCF and Merkle^2 due to the efficiency in verification. GlassDB performs the best among the ledger databases. In particular, it outperforms QLDB, SQLLedger, and LedgerDB by up to 3×, 2.2×, and 1.5×, respectively. GlassDB, LedgerDB, and SQLLedger outperform QLDB since they



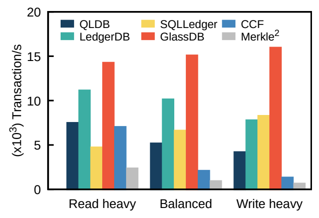
(a) **Throughput for balanced workload**



(b) **Latency for balanced workload**



(c) **Per-operation Latency for balanced workload**



(d) **Throughput for read-write workloads**

Figure 9: Single-node performance for YCSB uniform workloads

asynchronously update their ledgers to reduce the cost on the critical path. Furthermore, they all use batching when updating the ledgers. In particular, LedgerDB appends new transactions to the end of the ledger and periodically updates the Merkle trees once for all new transactions. SQLLedger batches all new transactions in one block, and periodically appends the block to the Merkle tree. GlassDB batches non-overlapping keys from new transactions and creates the blocks by version. The verification process in CCF is less efficient as it requires the client to send an HTTP request for every commit. Merkle^2 suffers from synchronous ledger updates and a significant amount of hashes to be calculated during verification. Hence, it has the lowest throughput. Figure 9b shows the overall latency of the systems. The results are consistent with those for throughput. The latency remains constant after reaching the peak because admission control is applied to the clients.

We further measure the end-to-end latency for each type of query, namely, read, write, and verification. Due to the deferred verification, the latency of the verification query depends on the number of keys verified in one batch. Therefore, we record the average verification latency per key and show the results in Figure 9c. Moreover, QLDB and Merkle^2 run implicit verification, hence, their verification latency cannot be measured from the user's side. The verification overhead is included in the read and write latency for these systems. We observe that the read latency of Merkle^2 is low (0.63ms) because it does not include verification, while its write latency is two orders of magnitude higher because it involves the updating and verification of the Merkle tree. CCF suffers from the verification overhead that is two orders of magnitude higher compared to the batch verification adopted by LedgerDB, and SQLLedger, and GlassDB. This is because verification is done individually for each transaction and the cost of the HTTP requests is very high. While the average number of keys in the verification batch is around 100 for LedgerDB, SQLLedger, and GlassDB.

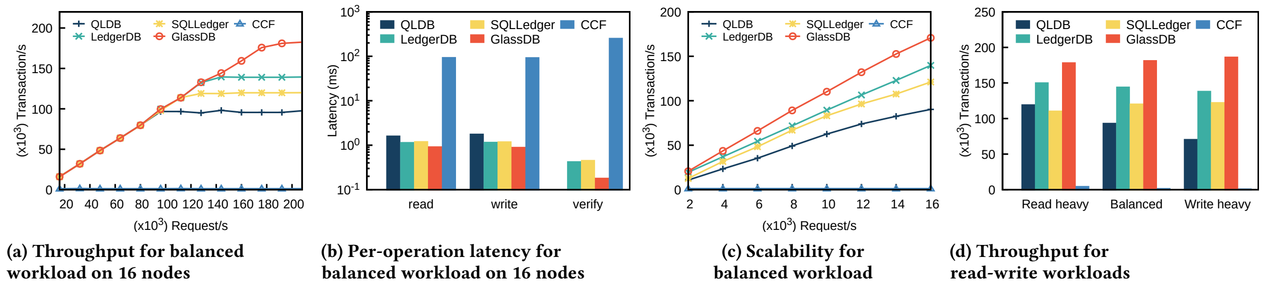
Next, we measure the throughput with read-heavy, balanced, and write-heavy workloads. The results are shown in Figure 9d. For GlassDB and SQLLedger, the throughput increases with workloads containing higher percentage of write operations. This is because the two systems are designed to optimize the write operation by only updating a minimum amount of data during the commit. They defer the more expensive ledger update to an asynchronous thread. Specifically, GlassDB only appends to the write-ahead log and keeps the data temporarily in memory when committing write operations. Similarly, SQLLedger writes the log and updates the indexes during the write operation. The rest of the systems need to update the entire or part of the ledger, and therefore, their throughput decreases with higher percentage of write operations.

Performance on multiple nodes. We then evaluate the performance of QLDB, LedgerDB, SQLLedger, GlassDB, and CCF on multiple-node setups. All systems are deployed with three replicas unless specifically elaborated. For QLDB, LedgerDB, SQLLedger, and GlassDB the data is partitioned across 16 shards in our setup. We start 160 VeriBench clients, where each client has a request rate in the range of 16,000 to 196,000 requests per second.

We first evaluate the systems using the balanced workload with uniform distribution (Zipf factor is 0). Figure 10a shows the throughput of the systems. Compared with the single-node experiments, the performance gap between ledger database systems and CCF becomes larger. This is because the performance of ledger databases improves significantly with data partitioning, while the throughput degrades for CCF with replication enabled. GlassDB outperforms QLDB, SQLLedger, and LedgerDB by up to 1.9×, 1.5×, and 1.3× respectively. It outperforms CCF by two orders of magnitude. Figure 10b shows the average latency for each operation. It has similar trends as the single-node latency. The average verification batch sizes are around 600, 300, and 1000 for LedgerDB, SQLLedger, and GlassDB, respectively. We then run the experiments with read-heavy, balanced, and write-heavy workloads and present the throughput in Figure 10d. Similar to single node results, GlassDB and SQLLedger achieve higher throughput when the percentage of write operations is higher since they are optimized for write operations. The rest of the systems have lower throughput for workloads with higher write percentages.

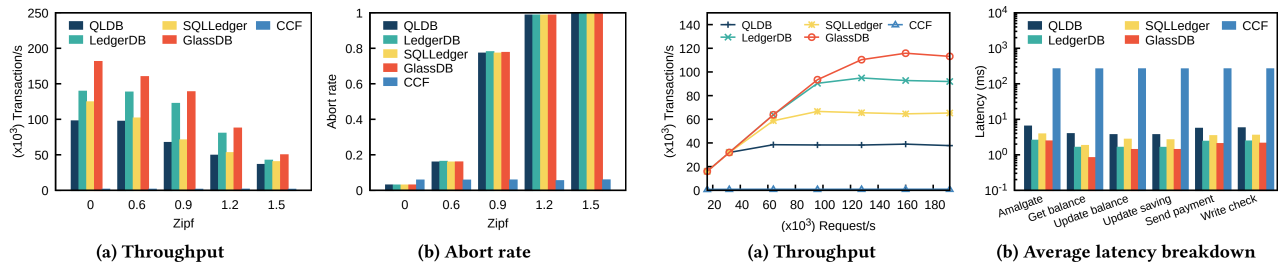
Next, we vary the number of nodes from 2 to 16 to test the scalability of the systems. As shown in Figure 10c, all ledger database systems are able to scale linearly up to 16 nodes, which indicates that the systems incur little overhead when distributing the data and using the atomic commit protocol (2PC). On the contrary, the performance of CCF drops as more nodes are deployed, since the system needs to replicate the data to more nodes.

We measure the impact of data conflicts on the performance by varying the Zipf factor from 0 to 1.5, and present the results in Figure 11. A higher Zipf factor indicates a higher possibility of operating on the same key, and hence, having update conflicts. CCF is not included due to its very low performance. We observe that the throughput of all systems drops and the latency increases as there are more aborts.



(a) Throughput for balanced workload on 16 nodes
(b) Per-operation latency for balanced workload on 16 nodes
(c) Scalability for balanced workload
(d) Throughput for read-write workloads

Figure 10: Multi-node performance for YCSB uniform workloads



(a) Throughput
(b) Abort rate
(a) Throughput
(b) Average latency breakdown

Figure 11: Performance for YCSB zipfian workloads

Figure 12: SmallBank performance on 16 nodes

4.3.2 SmallBank. Next, we use SmallBank, an OLTP workload, to evaluate the performance of the systems. Before the experiments start, we initialize the systems with 200,000 accounts from 100,000 users with a fixed amount of money. The experiments are then conducted by uniformly generating and dispatching six types of transactions to the systems, namely, amalgamate, get balance, update balance account, update saving account, send payment, and write check. We run the experiments with request rates ranging from 16,000 to 196,000 requests per second. The results are shown in Figure 12. We omit the scalability results since they have similar trends with the YCSB experiments.

Figure 12a shows the throughput of the systems. Compared with the YCSB results, the throughput of the systems running SmallBank workloads is up to 1.6× lower due to multiple operations done per transaction. Moreover, the systems reach the peak throughput at a lower request rate, since the server execution time for each transaction includes data access, conflict checking, and other operations. GlassDB achieves the highest throughput, outperforming QLDB, SQLLedger, LedgerDB, and CCF by up to 3×, 1.7×, 1.2×, and two orders of magnitude respectively.

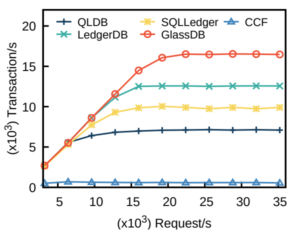
Figure 12b shows the latency of each type of transaction. The transaction types amalgamate and send payment incur higher latency compared to the other four types due to more operations involved, i.e., both transactions have two read and two write operations. CCF latency is relatively constant since the key-value store is kept in memory. Its high latency stems from the expensive replication and verification.

4.3.3 TPC-C. We further evaluate the systems with TPC-C, a more computation-heavy OLTP workload. We generate the TPC-C workloads with five warehouses and let the systems load around 2,550,000 initial records into all the tables. VeriBench runs all five TPC-C transaction types, namely, new order, payment, order status, delivery, and stock level. These five types account for 42%, 42%, 4%, 4%, and 4% of the total number of transactions, respectively. The transaction requests are sent at rates ranging from 3,200 to 36,000 requests per second. The results are shown in Figure 13.

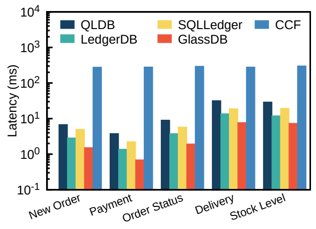
Figure 13a shows the throughput of the systems. The results are in line with previous experiments, but the peak throughput is 7× lower compared to SmallBank. This is expected, as TPC-C transactions are more computation-heavy and incur more conflicts compared to SmallBank. GlassDB achieves the highest performance, with 2.3×, 1.3×, 1.6×, and 30× higher throughput than QLDB, LedgerDB, SQLLedger, and CCF, respectively. CCF, benefiting from using stored procedures, has a smaller performance gap with the ledger databases since the transactions in TPC-C often contain multiple dependencies.

Figure 13b shows the average latency of each type of transaction. The delivery and stock level transactions incur higher latency because of the higher number of dependencies in the transactions. Moreover, the higher number of operations per transaction leads to multiple round trips for systems implementing interactive transactions, such as the four ledger database systems. The latency remains constant for CCF since the transactions are executed as stored procedures with one round trip.

4.3.4 Range Workload. We conduct the experiment with a dataset consisting of 100,000 keys and continuously sending range queries to the server. We measure the performance of queries under small, medium, and large ranges, which have average sizes of 10, 100,

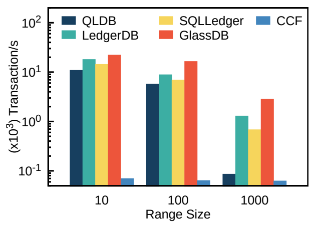


(a) Throughput

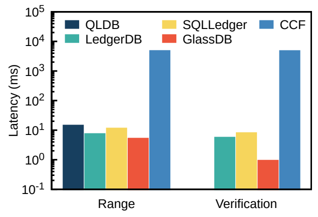


(b) Average latency breakdown

Figure 13: TPC-C performance on 16 nodes



(a) Throughput, 16 nodes



(b) Latency

Figure 14: Range workload performance on 16 nodes

and 1000, respectively. The throughput is presented in Figure 14a. GlassDB has the highest throughput. In particular, it outperforms QLDB, LedgerDB, SQLLedger, and CCF by up to 33×, 2.2×, 4.1×, and more than two orders of magnitude, respectively. The performance of QLDB drops significantly as the range size increases because it only exposes verification API for a single key, which results in overwhelming proofs. Figure 14b shows the latency for each operation under the small range. We observe that the latency of CCF for range operation is 50× higher than its latency for YCSB read operation. It is because the CCF framework does not provide the ordered scan of its built-in key-value store. Hence, users have to scan the entire store.

4.3.5 Provenance Workload. To conduct this experiment, we initialize each key of the dataset with 10 history versions on average. We let the clients continuously send provenance queries to get all history versions of a key. The results are shown in Figure 15. The throughput of GlassDB is 1.4×, 1.1×, and 1.2× higher than that of QLDB, LedgerDB, and SQLLedger, respectively. We observe a larger gap between GlassDB and CCF, i.e. around three orders of magnitude, compared with normal put and get operations. This is because CCF only allows users to fetch one historic state for each endpoint invocation. Consequently, the clients need to send multiple requests to get the entire history of a key.

4.4 Micro Benchmarks

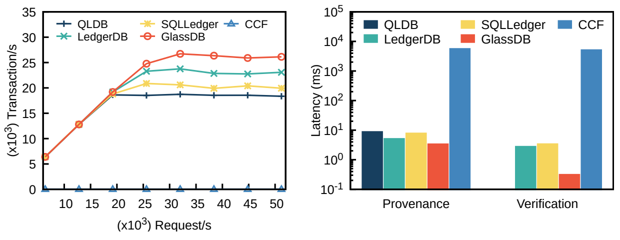
4.4.1 Delay. We first evaluate how deferred verification and asynchronous ADS update help in improving the system performance. We vary the verification delay from 10ms to 1280ms and fix the block time to 10ms. We run the YCSB balanced workload with uniform distribution on the systems that support deferred verification, namely, LedgerDB, SQLLedger, and GlassDB. The results, presented in Figure 16a, show that the throughput of all systems increases as the delay time increases. This is because with larger delay time, the clients will batch more keys for each verification request while sending fewer requests. Consequently, the systems benefit from batching of proof generation and verification.

Next, we evaluate the effect of asynchronous updates by varying the block time from 10ms to 1280ms and fixing the verification delay to 1280ms. From figure 16b, we observe that the systems achieve higher throughput with larger block time, especially in the interval [0, 200]ms. This is due to batch update: with larger batches, the systems are able to append more keys to the ADS and compute the cryptographic functions in one round, therefore, improving the efficiency. Furthermore, since SQLLedger and GlassDB are designed to perform minimum tasks for write operations, they benefit more from batching. The throughput improves by 1.3× and 2.4× for GlassDB and SQLLedger respectively when the block time increases from 10ms to 1280ms. On the contrary, LedgerDB creates the block entries of the ledger during each transaction commit, therefore, a higher block time leads less improvements.

4.4.2 Auditing. For systems that incorporate auditors to help check the consistency of the ADS, we evaluate the cost of the audition process by measuring the verification time per block. We run the experiment with SQLLedger, LedgerDB, and GlassDB because they expose explicit auditor API to users. The experiment is conducted with 16 nodes and 160 clients using YCSB balanced workload. We start an audit client executing audit tasks every second. To fairly compare the systems, we set the same block time of 10ms. The results are shown in Figure 17. LedgerDB incurs the highest audit latency, since the verification of its clue indexes is costly. The clue indexes are a set of skip-list trees with the leaf nodes being the transactions containing specific search keys in chronological order. LedgerDB needs to verify the total number of leaf nodes in each clue index through a Merkle Patricia Trie. After that, each leaf needs to be verified using the Merkle tree built on the ledger. GlassDB exhibits the lowest audit latency due to its efficient ledger structure, which also serves as the search tree. Therefore, it is easy to batch the proof and no additional ADS is required to protect the indexes.

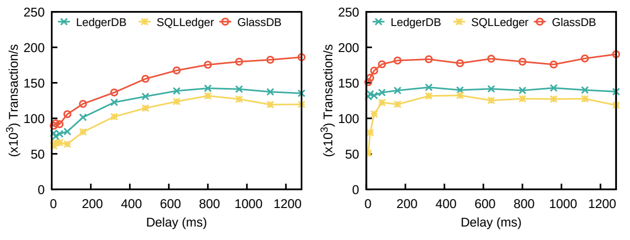
4.4.3 Storage. Lastly, we measure the storage of the systems to evaluate the space efficiency of ADS. We initialize the systems with an increasing number of records from 10,000 to 160,000, and measure the space consumption. As shown in Figure 18, GlassDB is the most space-efficient due to a smaller ADS and aggressive batching algorithm. While QLDB has the highest storage consumption because it updates the ledger for every operation, resulting in overwhelming metadata.

4.4.4 Security. In this section, we assess the security of the systems according to the security levels defined in Section 3.4. QLDB passes the test for inclusion proof but does not provide append-only proof, therefore, has a security level of 1. Merkle^2 and SQL Ledger achieve security levels of 2 by passing the inclusion and append-only proof tests. LedgerDB and GlassDB pass all inclusion proof, current-value proof, and append-only proof tests. Therefore, their security levels are 3. CCF updates the fully replicated ledger in a trusted execution environment, and therefore, has a security level of 4.



(a) Throughput (b) Average latency breakdown

Figure 15: Provenance workload performance on 16 nodes



(a) Impact of delay (b) Impact of block time

Figure 16: Performance influence of delay and block time.

4.5 Discussion

From the experimental results, we observe that the design choices affect the performance and security of *verifiability-enabled database systems* to different degrees.

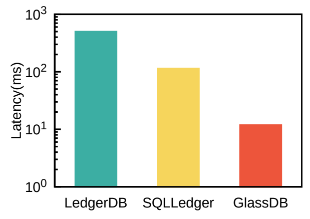
From the performance aspect, partitioning the data into multiple machines has the highest impact and improves performance by more than one order of magnitude. Next, we highlight the impact of batching and deferred verification. As an example, we note the performance gap between QLDB and the other ledger databases. Lastly, the ADS could affect the performance in different degrees depending on its efficiency.

From the security aspect, the threat model is the most important design choice, followed by verification, and ADS. Systems that adopt a threat model with Byzantine actors are the most secure by preventing malicious behavior from happening. They typically do this by employing secure hardware (trusted execution environments) or Byzantine fault-tolerant protocols. Next, there are systems that provide a strong security guarantee by verifying the integrity immediately after each operation. In contrast, systems with deferred verification leave a vulnerability window that may temporarily violate the security guarantee. On-demand verification does not guarantee the updates of data, which has the worst security guarantee. Lastly, there is the security provided by the ADS, which generally protects data from tampering. A more effective ADS also protects the indexes of data. This could further enhance the security guarantee by offering a current-value proof.

5 RELATED WORKS

We observe a research trend in exploring the fusion design between blockchains and traditional databases [2, 15, 22, 27, 33] to obtain both data verifiability and auditability, and effective query processing. Some of the systems try to build databases' query processing engines on top of blockchain layers, however, such systems merely reach a high throughput due to the congenital limitation of blockchains. On the other hand, ledger databases [4, 6, 39] adopt ledger structures from blockchains to build verifiable databases. Such systems usually remove the expensive BFT consensus in blockchains, thus, they are more competent in processing OLTP workloads with high efficiency.

Blockchain benchmarking frameworks. BlockBench [14] is one of the first frameworks to comprehensively evaluate blockchains. Hyperledger Caliper [19] is another widely-used blockchain benchmarking framework, supporting a range of blockchain systems such



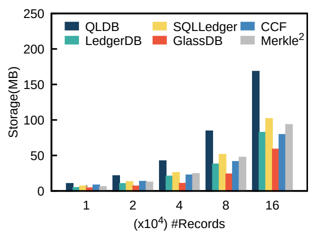


Figure 17: Auditing latency Figure 18: Storage usage

as Hyperledger Fabric [5], Ethereum [37], and FISCO BCOS [28]. However, they are more focused on consensus and smart contracts.
Traditional database benchmarking tools. There is a wide range of performance benchmarks [11–13, 16, 20, 21, 30, 34] for database systems designed for different purposes and handling different types of data. Vieira et al [36] evaluates the security of databases from general aspects. In contrast, our work focuses on the verifiability of databases and evaluates the performance and security impact.

6 CONCLUSIONS

With the increasing digitization of businesses and cloud hosting, there have been increasing demands for transactional systems to be verifiable and auditable. Various commercial *verifiability-enabled database systems* have been designed to meet the demands by supporting the integrity of data, history, and query results. In this paper, we conducted a survey on the design of such state-of-the-art systems. We then proposed VeriBench a framework for benchmarking *verifiability-enabled database systems* with both macro- and micro-benchmarks. Our extensive performance study identified various bottlenecks and their possible causes. We hope that this study and the open-source benchmarking framework will facilitate further development in this area.

ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore under its Emerging Areas Research Projects (EARP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. The work of Meihui Zhang is supported by National Natural Science Foundation of China (62072033).

REFERENCES

[1] AlibabaCloud. 2022. *LedgerDB*. https://www.alibabacloud.com/product/ledgerdb
[2] Lindsey Allen et al. 2019. Veritas: Shared Verifiable Databases and Tables in the Cloud. In *CIDR*. 1–9.
[3] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE*. 576–585.
[4] Amazon. 2019. *Amazon Quantum Ledger Database*. https://aws.amazon.com/qldb/
[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *EuroSys*. 1–15.
[6] Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason Anderson, and Jakub Szymaszek. 2021. SQL Ledger: Cryptographically Verifiable Data in Azure SQL Database. In *SIGMOD*. 2437–2449.
[7] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In *SIGMOD*. 251–266.
[8] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. 2011. Verifiable Delegation of Computation over Large Datasets. In *CRYPTO*. 111–131.
[9] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*.
[10] ConsenSys. 2020. *ConsenSys/quorum: A permissioned implementation of Ethereum supporting data privacy*. https://github.com/ConsenSys/quorum
[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154.
[12] Akon Dey, Alan D. Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *ICDE*. 223–230.
[13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.
[14] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *SIGMOD*. 1085–1100.
[15] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB - A Shared Database on Blockchains. *PVLDB* 12, 11 (2019), 1597–1609.
[16] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. BigBench: towards an industry standard benchmark for big data analytics. In *SIGMOD*. 1197–1208.
[17] Google. 2020. Certificate Transparency. https://www.certificate-transparency.org/.
[18] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. 2021. Mekle^2: a low-latency transparency log system. In *SP*. 285–303.
[19] Hyperledger. 2022. Hyperledger Caliper. https://www.hyperledger.org/use/caliper
[20] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *PVLDB* 9, 13 (2016), 1317–1328.
[21] Scott T. Leutenegger and Daniel Dias. 1993. A Modeling Study of the TPC-C Benchmark. In *SIGMOD*. 22–31.
[22] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. 2018. BigchainDB: A Scalable Blockchain Database. (2018). https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf
[23] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. 2015. CONIKS: Bringing Key Transparency to End Users. In *Usenix Security*. 383–398.
[24] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology*. 369–378.
[25] Microsoft. 2022. *The Confidential Consortium Framework*. https://github.com/microsoft/CCF
[26] Microsoft. 2022. *Ledger overview*. https://archive.ph/mvXCt
[27] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran. 2019. Blockchain Meets Database: Design And Implementation Of A Blockchain Relational Database. *PVLDB* 12, 11 (2019), 1539–1552.
[28] FISCO open-source working group. 2022. *FISCO BCOS*. http://www.fisco-bcos.org/
[29] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *SP*. 238–252.
[30] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. 2011. YCSB++ benchmarking and performance debugging advanced features in scalable table stores. In *SOCC*. 1–14.
[31] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cedric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olga Ohrimenko, Felix Schuster, Roy Schuster, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. 2019. *CCF: A Framework for Building Confidential Verifiable Replicated Services*. https://github.com/microsoft/CCF/blob/main/CCF-TECHNICAL-REPORT.pdf
[32] Mark Ryan. 2014. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*.
[33] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich, and Divya Agrawal. 2019. ChainifyDB: How to Blockchainify any Data Management System. http://arxiv.org/abs/1912.04820
[34] Rebecca Taft, Manasi Vartak, Nadathur Rajagopalan Satish, Narayanan Sundaram, Samuel Madden, and Michael Stonebraker. 2014. GenBase: a complex analytics genomics benchmark. In *SIGMOD*. 177–188.
[35] UCB. 2021. *MerkleSquare*. https://github.com/ucbrise/MerkleSquare
[36] M. Vieira and H. Madeira. 2005. Towards a security benchmark for database management systems. In *DSN*. 592–601.
[37] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[38] Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. 2022. Litmus: Towards a Practical Database Management System with Verifiable ACID Properties and Transaction Correctness. In *SIGMOD*. 1478–1492.
[39] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. 2020. LedgerDB: A Centralized Ledger Database for Universal Audit and Verification. *PVLDB* 13, 12 (2020), 3138–3151.
[40] Cong Yue, Tien Tuan Anh Dinh, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Xiaokui Xiao. 2023. GlassDB: An Efficient Verifiable Ledger Database System Through Transparency. *PVLDB* 16, 6 (2023), 1359–1371.
[41] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. 2020. Analysis of Indexing Structures for Immutable Data. In *SIGMOD*. 925–935.
[42] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *SP*. 863–880.
[43] Zhiwei Zhang, Xiaofeng Chen, Jin Li, Xiaoling Tao, and Jianfeng Ma. 2019. HVDB: a hierarchical verifiable database scheme with scalable updates. *Journal of Ambient Intelligence and Humanized Computing* 10, 8 (2019), 3045–3057.