GlassDB: An Efficient Verifiable Ledger Database System Through Transparency

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

Tien Tuan Anh Dinh
Singapore University of Technology and Design
dinhtta@sutd.edu.sg

Zhongle Xie
National University of Singapore
zhongle@comp.nus.edu.sg

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

Gang Chen
Zhejiang University
cg@zju.edu.cn

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

Xiaokui Xiao
National University of Singapore
xkxiao@nus.edu.sg

ABSTRACT

Verifiable ledger databases protect data history against malicious tampering. Existing systems, such as blockchains and certificate transparency, are based on transparency logs — a simple abstraction allowing users to verify that a log maintained by an untrusted server is append-only. They expose a simple key-value interface without transactions. Building a practical database from transparency logs, on the other hand, remains a challenge.

In this paper, we explore the design space of verifiable ledger databases along three dimensions: abstraction, threat model, and performance. We survey existing systems and identify their two limitations, namely, the lack of transaction support and the inferior efficiency. We then present GlassDB, a distributed database system that addresses these limitations under a practical threat model. GlassDB inherits the verifiability of transparency logs, but supports transactions and offers high performance. It extends a ledger-like key-value store with a data structure for efficient proofs, and adds a concurrency control mechanism for transactions. GlassDB batches independent operations from concurrent transactions when updating the core data structures. In addition, we design a new benchmark for evaluating verifiable ledger databases, by extending YCSB and TPC-C benchmarks. Using this benchmark, we compare GlassDB against four baselines: reimplemented versions of three verifiable databases, and a verifiable map backed by a transparency log. Experimental results demonstrate that GlassDB is an efficient, transactional, and verifiable ledger database system.

1 INTRODUCTION

A verifiable database protects the integrity of user data and query execution on untrusted database providers. Until recently, the focus has been on protecting the integrity of query execution [21, 27, 37]. In this context, users upload the data to an untrusted provider which executes queries and returns proofs that certify the correctness of the results. However, such OLAP-style verifiable databases rely on complex cryptographic primitives that limit the performance or the range of possible queries.

We observe a renewed interest in verifiable databases, with a focus on OLTP-style systems. In particular, there emerges a new class of systems, called *verifiable ledger databases*, whose goal is to protect the integrity of the data history. In particular, the data is maintained by an untrusted provider that executes read and update queries. The provider produces integrity proofs about the data content and its entire evolution history.

An example of verifiable ledger databases is the blockchain [4, 10, 32]. The blockchain maintains a replicated append-only log in a decentralized setting. It protects the integrity of the log against Byzantine attackers, by running a distributed consensus protocol among the participants. The integrity proof (in a permissioned blockchain) consists of signed statements from a number of participants. Another example is a certificate transparency log [15, 24], in which a centralized server maintains a tamper-evident, append-only log of public key certificates. The server regularly publishes summaries of the log which are then checked for consistency by a set of trusted *auditors*. The integrity proof generated by the server can be verified against the published and audited summaries. The third example is Amazon's Quantum Ledger Database (QLDB) service [2], which maintains an append-only log similar to that of certificate transparency. QLDB uses the log to record data operations that are then applied to another backend database.

Our goal is to build a practical verifiable ledger database system. We observe that the three examples above are built from a common abstraction, namely a *transparency log*, which provides two important security properties. First, users can verify that the log is append-only, namely, any successful update operations will not be reverted. Second, users can verify that the log is linear, that is, there is no fork in history. Blockchains enforce these properties by replicating the log and running consensus protocol among the participants. Certificate transparency log and QLDB rely on auditors or users to detect violations of the properties. Despite useful security properties, transparency logs are inadequate as databases. In fact, we identify three challenges in building a practical verifiable ledger database system on top of this abstraction.

The first challenge is the lack of a unified framework for comparing verifiable ledger databases. In particular, we note that the three systems above have roots from three distinct fields of computer science: blockchains are from distributed computing, certificate transparency is from security, and QLDB is from database. As a consequence, there is no framework within which they can be compared fairly. The second challenge is the lack of traditional database abstraction, that is, transactions. The transparency logs used in existing systems expose simple key-value interfaces without transactions. This simplifies the design of the transparency logs, but makes them unsuitable for OLTP workloads. The third

challenge is how to achieve high performance while retaining security. Blockchains, for instance, suffer from poor performance due to the consensus bottleneck. Certificate transparency has low performance because of expensive disk-based operations, while QLDB generates inefficient integrity proofs for verifying the latest data.

We address the first challenge by establishing the design space of verifiable ledger databases. The space consists of three dimensions. The abstraction dimension captures the interface exposed to the users, which can be either key-value or general transactions. The threat model dimension includes different security assumptions. The performance dimension includes design choices that affect the integrity proof sizes and the overall throughput. In addition to the design space, we propose a benchmark for comparing the performance of different verifiable ledger databases. Specifically, we extend traditional database benchmarks, namely YCSB and TPC-C, with additional workloads containing verification requests on the latest or historical data.

We address the second and third challenges by designing and implementing GlassDB, a new distributed verifiable ledger database system that overcomes the limitations of existing systems. GlassDB supports distributed transactions and has efficient proof sizes. It relies on auditing and user gossiping for security. It achieves high throughput by building on top of a novel data structure: a two-level Merkle-like tree. This data structure protects the data indexes, which enables secure and efficient verification. Furthermore, it is built over the states, as opposed to over transactions, which enables efficient lookup and proof generation while reducing the storage overhead. GlassDB partitions data over multiple nodes, where each node maintains a separate ledger, and uses the classic two-phase commit protocol to achieve transaction semantics. Each node of GlassDB has multiple threads for processing transactions and generating proofs in parallel, and a single thread for updating the ledger storage. GlassDB uses optimistic concurrency control to resolve conflicts in transactions, and batching to reduce the cost of updating and persisting the core data structure. We conduct an extensive evaluation of GlassDB, and benchmark it against four baselines: reimplemented versions of three verifiable databases, and a key-value store based on the transparency log.

In summary, we make the following contributions.

- We present the design space of verifiable ledger databases, consisting of three dimensions: abstraction, threat model, and performance. We discuss how existing systems fit into this design space.
- We design and implement GlassDB, a distributed verifiable ledger database system that addresses the limitations of existing works. In particular, GlassDB supports distributed transactions with high performance, under a practical threat model.
- We design new benchmarks for evaluating and comparing verifiable ledger databases. The benchmarks extend YCSB and TPC-C with workloads that stress test the performance of proof generation and historical data access.
- We conduct detailed performance analysis of GlassDB, and compare it against four baselines, namely QLDB [2], LedgerDB [34] and SQL Ledger [5], and a key-value store based on transparency log, Trillian [16]. The results show that GlassDB consistently outperforms the four baselines across all workloads.

2 VERIFIABLE LEDGER DATABASES

2.1 What A Verifiable Ledger Database Is

A verifiable database is a database that ensures the integrity of both the data and query execution. It is useful in outsourced settings, in which the data is managed by an untrusted third party whose misbehavior can be reliably detected. Until recently, verifiable databases have focused on ensuring the integrity of query execution, particularly on the performance and expressiveness of analytical queries that can be verified [21, 27, 37].

A verifiable ledger database is one instance of verifiable database, which focuses on protecting the integrity of both the data content and data history. A user issues a read or update operation (OLTP query) to the database server (or server), which then executes the operation and appends it to a history log H. The database returns integrity proofs showing that (1) the operation is executed correctly on the states derived from H, and (2) the operation is appended to H, and H is append-only. These proofs ensure that malicious tampering such as changing the data content, back-dating operations, forking the history log, are detected. Existing works on authenticated data structure [21], for example, only meet the first condition.

More formally, a verifiable ledger database D consists of four main operations.

- $(\text{digest}_{S',H'}, \pi, R, S', H') \leftarrow \text{Execute}(S, H, op)$: this is run by the database server. It takes as input the current state S and history log H, and the user operation op. It executes op, updates the states S accordingly, and appends op to H. It returns the updated states S', updated history H', a digest value $\text{digest}_{S',H'}$ computed over the new state and history, an execution result R, and a proof π.
- $\pi \leftarrow \text{ProveAppend}(\text{digest}_{S,H}, \text{digest}_{S',H'})$: this is run by the server. It takes as input two digest values corresponding to two different history logs. It returns a proof π.
- $\{0, 1\} \leftarrow \text{VerifyOp}(op, R, \pi, \text{digest}_{S,H}, \text{digest}_{S',H'})$: this is run by the user. It takes as input the user operation op, the execution result R, the proof π, and $\text{digest}_{S,H}, \text{digest}_{S',H'}$ that correspond to the history and state before and after op is executed. It returns 1 if the proof is valid, and 0 otherwise.
- $\{0, 1\} \leftarrow \text{VerifyAppend}(\text{digest}_{S,H}, \text{digest}_{S',H'}, \pi)$: this is run by the user. It takes as input two digest values corresponding to two different history logs, and a proof π. It returns 1 if the proof is valid, and 0 otherwise.

Definition 1. *A verifiable ledger database, which supports the four operations defined above, is secure if it satisfies the following properties.*

- **Integrity:** *the database server cannot tamper with the user operation without being detected. More precisely, given any* $op, \pi, R, S, S', H, H', \text{digest}_{S,H}, \text{digest}_{S',H'}$ *such that* $(\text{digest}_{S',H'}, \pi, R, \_, \_) \leftarrow \text{Execute}(S, H, op)$, $\text{VerifyOp}(op, R, \pi, \text{digest}_{S,H}, \text{digest}_{S',H'}) = 1$, *the server cannot find* π', R' *such that* $R' \neq R$ *and* $\text{VerifyOp}(op, R', \pi', \text{digest}_{S,H}, \text{digest}_{S',H'}) = 1$.
- **Append-only:** *the database server cannot fork the history log without being detected. More precisely, for any* $op, S_1, H_1, S, H, \text{digest}_{S,H}, \text{digest}_{S_1,H_1}, R, \pi$ *such that* $(\text{digest}_{S_1,H_1}, \pi, R, S_1, H_1) \leftarrow \text{Execute}(S, H, op)$ *and* $\text{VerifyOp}(op, R, \pi, \text{digest}_{S,H}, \text{digest}_{S_1,H_1}) = 1$, *the server cannot*

find $op', S_2, H_2, S', H', \text{digest}_{S',H'}, \text{digest}_{S_2,H_2}, R', \pi', \pi_a$ *such that* $(\text{digest}_{S_2,H_2}, \pi', R', S_2, H_2) \leftarrow \text{Execute}(S', H', op')$, $\text{VerifyOp}(op', R', \pi', \text{digest}_{S',H'}, \text{digest}_{S_2,H_2}) = 1$, $\pi_a \leftarrow \text{ProveAppend}(\text{digest}_{S_1,H_1}, \text{digest}_{S_2,H_2})$, $\text{VerifyAppend}(\text{digest}_{S_1,H_1}, \text{digest}_{S_2,H_2}, \pi_a) = 1$, $|H_1| < |H_2|$, *and* H_1 *is not a prefix of* H_2.

Example. Suppose the current history at the database D is H_1, corresponding to the state $S_1 = \{a = 1, b = 2\}$. Starting from the same H_1, one user A issues a sequence of operations $\langle \text{add}(c, 3), \text{remove}(c), \text{add}(d, 4) \rangle$, while another user B issues $\text{add}(\text{d}, 4)$. This scenario can happen due to concurrency, or because the server acts maliciously. For user A's last operation, the server returns $\text{digest}_{S_4,H_4}, \pi_4, R_4$. For user B's operation, it returns $\text{digest}_{S_2',H_2'}, \pi_2', R_2'$. The append-only property means that for $\pi \leftarrow \text{ProveAppend}(\text{digest}_{S_2',H_2'}, \text{digest}_{S_4,H_4})$, $\text{VerifyAppend}(\text{digest}_{S_2',H_2'}, \text{digest}_{S_4,H_4}, \pi) = 0$. In other words, the server cannot fork the history, even if the two branches result in the same state.

2.2 Design Space

Definition 1 admits a simple, naive design in which the proof consists of the query result and complete history H (signed with the provider's cryptographic key). The users replay all operations in H to verify the correctness of H, and they broadcast messages among each other to detect any inconsistent behavior, e.g., the database signed different histories that were not linear. However, this design incurs significant communication and computation costs for the users. A more practical design would need to reduce these costs. To enable a principled comparison of different verifiable ledger databases, we propose to explore the design space along three dimensions: abstraction, threat model, and performance.

2.2.1 Abstraction. This refers to the data model and programming model supported by the database. There are two main data models with different trade-offs. On the one hand, the key-value model exposing simple `Put` and `Get` operations is flexible and scalable. On the other hand, the relational data model supports declarative query languages, and is easier to use. The key-value data model is more suitable for ledger databases running OLTP workloads, since it simplifies the verification logic. In contrast, the relational model entails more complexity, because it uses secondary indexes and supports complex operations such as join and aggregation, which are difficult to verify. Systems such as SQL Ledger and QLDB support the relational model, but they do not guarantee the integrity of indexes and operations.

The two main programming models in ledger databases are transactional and non-transactional. In the former, users can execute multiple operations in one transaction, with serializable properties (ACID). Examples include QLDB, LedgerDB, and SQL Ledger. In the latter, the database performs one operation at a time, without guarantees across multiple operations. Examples include systems such as Trillian, Merkle^2, and Coniks. There are other design choices between non-transactional abstraction and ACID transactional abstraction. In particular, some database systems support serializable transactions over small sets of related keys [9, 12], or keys within the same partitions [20]. Some other databases support transactions with weaker isolation levels, such as snapshot isolation [25]. These design choices can deliver higher performance than the design with serializable transactions, but they suffer from anomalies. We note that in the context of verifiable ledger databases, such anomalies can happen due to the server acting maliciously to cause conflict during execution, instead of due to real concurrency. As a consequence, the application needs to handle a potentially large number of anomalies, which increases complexity and performance overhead. For the rest of the paper, we use the term transactions to refer to serializable transactions.

2.2.2 Threat model. The security of a verifiable ledger database is defined as having integrity proofs that satisfy the two conditions in Definition 1 under a specific threat model. All threat models share common assumptions that the attacker cannot break cryptographic primitives or mount denial of service attacks.

There are three main threat models in the context of verifiable ledger databases. The most common model involves a single untrusted database provider that behaves in a Byzantine manner. It has been shown that in this setting, it is only possible to achieve *fork consistency* [22], i.e., users cannot prevent misbehavior but can only detect it by communicating with each other. As a result, this model assumes that users engage in gossiping, and that the attacker cannot permanently partition the network. To further reduce the cost on the users, the model can be extended by introducing a set of trusted, powerful users called *auditors* that only gossip among themselves. The auditors check for the misbehavior of the database on behalf of the users.

The second threat model assumes that the database is replicated over a set of providers, the majority of which are trusted. Even though there are malicious providers, the system as a whole enforces the correct behavior. In particular, the providers participate in a distributed, Byzantine fault tolerant consensus protocol to ensure consistency of the database [8]. We note that such consensus-based systems provide stronger security guarantees than systems with single malicious providers, that is they can prevent misbehavior as opposed to only detecting it.

The final threat model assumes that the database server is malicious, but it is equipped with some trusted hardware that supports trusted execution environments (TEEs). The TEE protects the computation and data running inside the environment against malicious operating systems and hardware attacks. The entire database can run securely inside the TEE. However, this model assumes that both the computation and the TEE itself are free of vulnerabilities, which does not always hold in practice [7].

2.2.3 Performance. The performance of a verifiable ledger database is evaluated based on two metrics: the user's verification cost, and the database throughput. The former depends on the complexity of the integrity proofs. An efficient proof is short and fast to verify. We further categorize integrity proofs into three types.

- Inclusion proof: given $\text{digest}_{S,H}$ and a value v corresponding to a key k, this proof ensures that this value is included at some point in H.
- Current-value proof: given $\text{digest}_{S,H}$ and a value v of a key k, this proof ensures that v is the latest value of k in H.

Table 1: GlassDB vs. other verifiable ledger databases. N is number of transactions, m is number of keys, and B is number of blocks, where $m \geq N \geq B$. All systems, except Forkbase, support inclusion proof of the same size as append-only proof.

System	Data Model	Transaction	Threat model	Append-Only Proof	Current-Value Proof	Throughput
QLDB [2]	Relational	Transaction	Audit	$O(\log N)$	$O(N)$	Low
LedgerDB [34]	Key-value	Transaction	Audit	$O(\log N)$	$O(N)$	Medium
SQL Ledger [5]	Relational	Transaction	Audit	$O(B)$	$O(N)$	Medium
Forkbase [31]	Key-value	Non-transaction	Audit	$O(N)$	$O(\log m)$	Medium
Blockchain [4]	Key-value	Transaction	Consensus	$O(1)$	$O(1)$	Low
CreDB [23]	Key-value	Transaction	Trusted hardware	$O(1)$	$O(1)$	Low
Trillian [16], ECT [29], Merkle^2[19]	Key-value	Non-transaction	Audit	$O(\log m)$	$O(\log m)$	Low
GlassDB	Key-value	Transaction	Audit	$O(\log B)$	$O(\log B + \log m)$	High

- Append-only proof: given $\text{digest}_{S,H}$ and $\text{digest}_{S',H'}$, the proof ensures that H is a prefix of H' (assuming that $|H| \leq |H'|$).

The database throughput is measured in terms of the number of user queries completed per second, and it depends on the cost of maintaining the security-related data structures for generating the proofs. Designs that exploit parallel execution and avoid contention will have high throughputs.

2.3 Transparency Log

Transparency log is an append-only log protected by a Merkle tree [11]. Each leaf represents an operation, for example updating a key-value tuple. This data structure supports all three types of integrity proofs. The *inclusion proof* consists of the Merkle path from the leaf to the root, which costs $O(\log(N))$ where N is the size of the log. The *append-only proof* includes intermediate nodes between two trees, and has the cost of $O(\log(N))$. The *current-value proof*, however, requires all the leaves of the tree, thus its cost is $O(N)$.

The security of transparency logs depends on auditing. In particulars, users broadcast signed Merkle roots to a number of auditors which check that that there is a single log with no forks. The check is done by requesting and verifying append-only proof from the database provider.

2.4 Review of Existing Systems

Table 1 compares existing verifiable ledger databases according to the design space above. These systems build on top of the transparency logs described above.

Commercial verifiable ledger databases. QLDB [2], LedgerDB [34], and SQL Ledger [5] are two recent services offered by major cloud providers. QLDB uses transparency logs for storing transactions, and executes the operations on indexed tables. However, its throughput is low due to the disk-based communication between the log and the indexed tables. LedgerDB [34] and SQL Ledger [5] improve the performance of QLDB by batching multiple operations when updating the Merkle roots of the log. Since QLDB and LedgerDB build Merkle trees over transactions, the append-only proof costs $O(\log(N))$. However, SQL Ledger store the blocks in a hash chain, requiring $O(B)$ cost for append-only proof. All three systems do not have protection over indexes, which require scanning to the latest transaction for current-value proof. The cost of this is $O(N)$.

Forkbase. Forkbase [31] is a state-of-the-art versioned, key-value storage system. It implements a variant of transparency logs called transparency maps. In particular, Forkbase builds a Merkle tree on top of immutable maps: each update operation results in a new map and a new Merkle root. Each Merkle root also includes a cryptographic pointer to the previous root. Unlike transparency logs, the current-value proof in Forkbase costs $O(\log(m))$ because the latest value is included in the map. However, the append-only proof is $O(N)$, since users have to follow the hash chain to ensure that there are no forks.

Blockchain. Existing blockchain systems assume the majority of trusted providers in a decentralized setting. The providers run a Byzantine fault tolerant consensus protocol to keep the ledger and global states consistent. As the system as a whole is trusted, signed statements from the blockchain can be used as integrity proofs. In particular, in a permissioned blockchain that tolerates f Byzantine failures, the proof contains signatures from $f + 1$ providers. The proof complexity is therefore independent of the history, or $O(1)$. However, the performance of a blockchain is limited by the consensus protocol [13].

CreDB Instead of relying on consensus to protect against Byzantine database servers, CreDB assumes that the server can create trusted execution environments backed by trusted hardware. CreDB processes user transactions inside the TEE and produces signed *witnesses* that capture the history of the states. Both inclusion and current-value proofs in CreDB are efficient, because they are simple messages signed by the TEE. However, the hardware limitation, e.g. the limited memory available for Intel SGX, makes the TEE a performance bottleneck, and results in low throughputs.

Public-key transparency logs. Trillian [16] combines transparency logs and maps to implement new primitives called *verifiable log-based map*. It exposes a key-value interface, and is used for storing public key certificates. When a key is updated, the map is updated and a new Merkle root is computed on the map. It then appends the log with both the operation and the Merkle root. As the result, both the current-value and append-only proofs are efficient, i.e. $O(\log(m))$ complexity. On the other hand, Trillian relies on trusted auditors to regularly verify the consistency between the log and the corresponding map. More specifically, the auditors reconstruct the map from the log and ensure that the Merkle root included in the log is correct. This auditing process is expensive and should be performed by powerful, external entities rather than by the users. Other systems such as CONIKS[24], ECT [29], and Merkle^2 [19] are similar to Trillian in our design space. They improve Trillian by adding support for privacy, revocation (non-inclusion proofs), and reducing the audit cost.

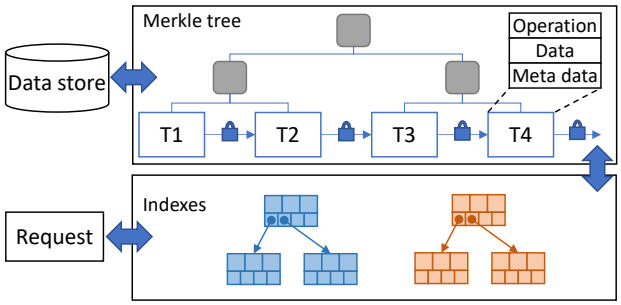


Figure 1: A simple verifiable ledger database. The ledger consists of blocks containing the operation, data, and metadata of the transactions.

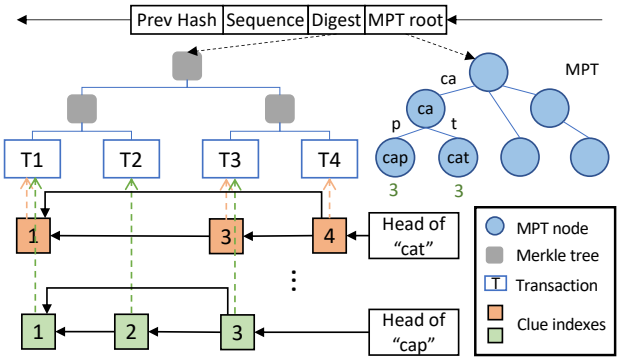


Figure 2: LedgerDB [34] authenticated data structures. There is one clue index for every key, and the size of each clue index is stored in a leaf of the Merkle Patricia Trie.

3 GLASSDB

3.1 Existing Designs

Figure 1 shows a design of verifiable ledger databases used in commercial systems such as QLDB [2]. The key idea is to replace the transaction log in conventional databases with a variant of transparency log called *ledger*. The ledger is a hash-chained sequence of blocks, each of which contains the operation type and parameters, and a Merkle tree is built on top of them to protect their integrity. Updating the ledger requires appending a new transaction block and rebuilding the Merkle tree.

Transaction execution in this design is similar to that in a conventional database. The transaction is first committed to the ledger as a new block, under some concurrency control mechanisms. Then, the data and indexes are updated. The transaction is considered committed once the ledger is updated and the data can be queried via the indexes. The response to the client includes a block sequence number indicating where in the ledger the transaction is committed. During verification, the client requests a digest of the ledger, and then sends a `GetProof` request containing the sequence number and the digest. It receives a Merkle proof showing that the specified block is included in the ledger. After verifying the proof, it checks that the data is included in the block.

The main advantage of this design is that it is easy to extend an existing database system into a verifiable ledger database. In addition, the design is independent of the underlying data abstraction and layout. However, it has two major deficiencies. First, it incurs significant overhead in transaction processing, because updates of the Merkle tree are in the critical path. Second, the indexes are not integrity protected, i.e., the server can respond with stale data. As a result, this design requires the client to scan the ledger to guarantee the returned value is current, which incurs $O(N)$ cost. For example, a key K is committed in both $T2$ and $T4$. The client wants to get the latest value of a key K, however, the server could return the value in $T2$. It will still pass the verification since the server return a true value. To check if it is the latest value, the client has to examine all the blocks from $T2$ to $T4$ for any later commitment.

LedgerDB [34] improves the design above by updating the authenticated data structures asynchronously. This technique is also adopted by SQL Ledger Figure 2 shows the different indexes and Merkle trees used in LedgerDB. In LedgerDB, each transaction is appended to a ledger in the form of a journal entry, and a Merkle tree built on top of the ledger is updated asynchronously in batch, which is called batch accumulated Merkle-tree (bAMT). LedgerDB maintains a skip-list index (called a *clue index*) for each individual data key, with each entry in the skip list pointing to the journal entry corresponding to the transaction that modifies the data key. The size of the index is stored as a leaf of a Merkle Patricia Trie, called clue-counter MPT(ccMPT). The roots of ccMPT and bAMT are stored as a block in a hash chain. LedgerDB also supports data freshness by using another ledger that stores time entries from a timestamp authority.

There are three limitations of LedgerDB's design that result in high verification costs. First, bAMT stores one transaction per leaf, therefore its size can be large when there are many transactions, which leads to larger proofs. Second, it is expensive to verify a value of a key, even in the presence of a trusted auditor. In particular, the ccMPT structure used to protect the clue index is not secure, because each leaf of the ccMPT stores only the size of each clue index, instead of capturing the content of the entire index. As a consequence, to verify the value of a key, the client needs to scan and verify the entire index to ensure that each entry in the clue index points to a correct journal entry. We note that even if a trusted auditor verifies the ccMPT and clue indexes, the client still needs to verify the clue index by itself, because a malicious server can modify the index without changing the ccMPT. For example, a client wants to get the latest value of *cat* committed in $T4$. The server will traverse the clue index of *cat* and return the latest transaction, $T4$. However, a malicious server can have a malformed clue index with missing entries or wrong pointers. In this case, the server can return $T3$ and still pass the verification, since *cat* exists in $T3$. To verify the clue index, it first get the number of leaves from MPT and all leaves in the clue index. If the number matches, it will verify each transaction using the bAMT. Finally, the size of the proof for multiple keys, even when the keys belong to the same transaction, grows linearly with the number of keys because each key requires a separate proof from the ccMPT.

3.2 GlassDB Overview

GlassDB is a new, distributed verifiable ledger database system that overcomes the limitations of the existing designs. It supports general transactions, which makes it easy to use for existing and future applications. It adopts the same threat model as QLDB and

LedgerDB, which assumes that the database server is untrusted, and there exists a set of trusted auditors that gossip among each other. GlassDB achieves high throughputs and small verification costs. Table 1 shows how the system fits in the design space.

There are three novelties in the design of GlassDB that facilitate its high performance. First, GlassDB adopts hash-protected index structures. The key insight we identify from the limitation of existing ledger databases is the lack of comprehensive and efficient protection of the indexes, which leads to either security issues or high verification overhead. Such limitations can be eliminated by adopting hash-protected index structures. Second, GlassDB builds its ledger over the state of data instead of transactions. One advantage of this approach is that the system can retrieve the data and generate current-value proofs more efficiently. Another advantage is that it results in a smaller data structure. The Merkle trees of the existing systems are built over the transactions, which grow quickly and lead to higher storage and computation overhead. In contrast, GlassDB's core data structure grows more slowly as it batches updates from multiple transactions. Third, GlassDB partitions the data over multiple nodes, which enables it to scale to achieve high throughput. Furthermore, it adopts three optimizations that help speed up transaction processing and verification, namely transaction batching, asynchronous persistence, and deferred verification.

Figure 3 shows the design of GlassDB. It partitions the data (modeled as key-value tuples) into different shards based on the hash of the keys, and uses two-phase commit (2PC) protocol to ensure the atomicity of cross-shard transactions. Each shard has three main components: a transaction manager, a verifier, and a ledger storage. A transaction request is forwarded to the transaction manager, which executes the transaction using a thread pool with optimistic concurrency control. A verification request is forwarded to the verifier, which returns the proof. The ledger storage maintains the core data structure that provides efficient data access and proof generation. Each shard maintains an individual ledger based on the records committed. The client keeps track of the key-to-shard mapping, and caches the digests of the shards' ledgers. GlassDB uses write-ahead-log (WAL) to handle application failures. It handles node failures by replicating the nodes.

The life cycle of a transaction at the server can be divided into four phases: prepare, commit, persist, and get-proof. The prepare phase checks for conflicts between concurrent transactions before making commit or abort decisions. The commit phase stores the write set in memory and appends the transaction to a WAL for durability and recovery. The persist phase appends the committed in-memory data to the ledger storage and updates the authenticated data structures for future verification. The get-proof phase generates the requested proofs for the client. In GlassDB, the persist and get-proof phases are executed asynchronously and in parallel with the other two phases. The detail is illustrated in Section 3.3.

3.2.1 APIs. The user (or client) starts by calling `Init(pk, sk)`, which initializes the client's session with the private key *sk* for signing transactions, and sends the corresponding public key *pk* to the auditors for verification. The client invokes `BeginTxn()` to start a transaction, which returns a transaction ID *tid* based on the client ID and timestamp. During the transaction, the client uses `Get(tid, key, (timestamp | block_no))` and `Put(tid, key, value)`. When ready to commit, the client invokes `Commit(tid)`, which signs and sends the transaction, including the buffered writes, to the server. This method returns a promise, which can be passed to `Verify(promise)` to request proof and verify it. The client frequently invokes `Audit(digest, block_no)` to send a digest of a given block to the auditors.

An auditor uses `VerifyBlock(digest, block_no)` to request the server for the block at `block_no`, proof of the block, and the signed block transactions. It verifies that all the keys in the transactions are included in the ledger. It also uses `VerifyDigest(digest, block_no)` to verify that the given digest and the current digest correspond to a linear history, by asking the server to generate append-only proofs. If the given block number is larger than the current block number, it uses *VerifyBlock* to verify all the blocks in between. Finally, the auditor calls `Gossip(digest, block_no)` to broadcast the current digest and block number to other auditors.

3.3 GlassDB Design

3.3.1 Ledger storage. The design goal of GlassDB is to build a storage system that not only offers efficient access to the data, but also supports efficient inclusion, latest, and append-only proofs. To this end, we use a Merkle variant called two-level pattern-oriented split tree (or two-level POS-tree).

A POS-tree is an instance of Structurally Invariant and Reusable Index (SIRI) [31, 35], which combines the Merkle tree and balanced search tree. A parent node in the POS-tree stores the cryptographic hash of its child nodes, such that the root node contains the digest of the entire tree. The user can perform efficient data lookup by traversing the tree. The POS-tree is built from the globally sorted sequence of data. The data is split into leaf nodes using content-defined chunking, in which a new node is created when a pattern is matched. The cryptographic hash values of the nodes in one level form the byte sequence for the layer above. The byte sequence is split into index nodes using similar content-defined chunking approach. POS-tree is optimized for high deduplication rates because of its content-defined chunking. It is immutable, that is, a new tree is created, using copy-on-write, when a node is updated. Finally, the POS-tree is *structurally invariant*, that is, the structure of the tree does not depend on the order of data inserted.

The core data structure of GlassDB is shown in the Figure 4. It consists of an upper level POS-tree and a lower level POS-tree. The lower level POS-tree is built on the database states and serves as the index. The leaf nodes store key-value tuples. For each key, the leaf node stores the pointer to the node containing the previous version of the key. For example, the Node b' stores the key K_9 with value V_9^2, and H_b which is the hash of node b where V_9^1 is. The internal nodes of this tree store the starting key of each child and the hash of the child node. The hash of the root node and other metadata such as block number, timestamp, and transaction IDs, are included in a data block, which is stored as a leaf of the upper level POS-tree. The keys of the upper level POS-tree are block numbers. The hash of this tree's root is the digest of the entire ledger. Retrieving a key from a given block number entails getting the data block with the corresponding block number from the upper level POS-tree, then traversing the lower level POS-tree to locate the data. When updating a key, new nodes are created at both levels using copy-on-write.

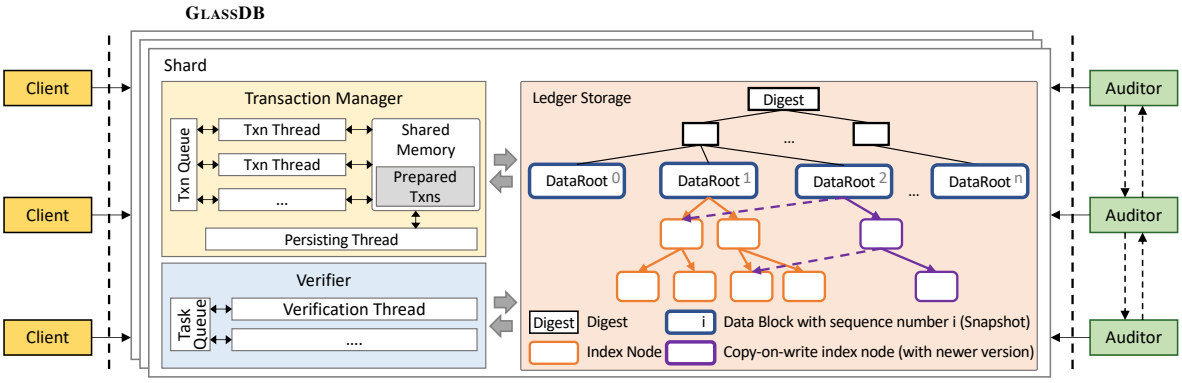


Figure 3: GlassDB design. The transaction manager executes transactions, with a persisting thread asynchronously append committed data to the ledger storage, and the verifier handles proof requests.

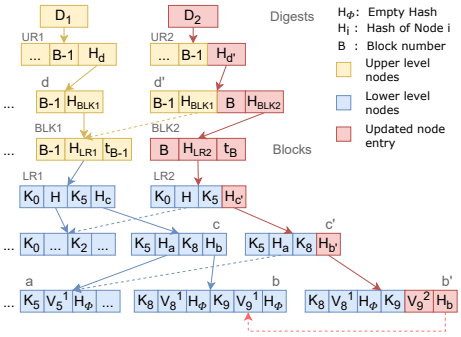


Figure 4: Two-level POS-tree.

One advantage of this authenticated data structure is that it provides efficient current-value proofs, in addition to the inclusion and append-only proofs. Unlike QLDB and LedgerDB described in Section 3.1, GlassDB cannot return stale data and still pass the verification, since each data block represents a snapshot of the database states, the latest values always appear in the last block. Moreover, any disorder of the index will be detected because the content of entire index is hash protected. For example, in Figure 4, the client wants to get latest K_9 with the digest D_2. The server will traverse the last block $BLK2$ and get V_9^2. If the server return with the older version V_9^1 and older proof generated from $BLK1$, it won't pass the verification, since the client will find out that the block fetched is not the latest. Another advantage is that enables efficient failure recovery. In particular, during recovery GlassDB re-executes the transactions to update the two-level POS-tree. If failure happens during updating of the lower level tree, all nodes created before failure can be reused, since the POS-tree structure would be the same given the same input (the order of updates does not matter). However, if failure happens during updating of the upper level tree, the nodes created before failure cannot be reused, because the data blocks include varying content like timestamps. To address this, we keep an additional mapping between block sequences and persisted data blocks. If the block sequence already exists, GlassDB will only re-execute the updates of the upper-level POS-tree based on the persisted blocks.

Example. Consider the two-level POS-tree in Figure 4. When K_9 is updated with the value V_9^2, GlassDB first updates the lower level tree using copy-on-write, creating a new leaf node b' by replacing the V_9^1 and H_ϕ with V_9^2 and H_b respectively. It then updates node c with the hash of node b', creating a new node c'. This is done recursively until a new root node $LR2$ is created. Next, GlassDB uses the hash of $LR2$ to create node $BLK2$ with block number B and timestamp t_B. Finally, it propagates the update towards the root of the upper level tree.

The inclusion proof of K_9 at $B-1$ includes the nodes b, c, $LR1$, $BLK1$, d, $UR1$, and D_1. The verification is done by recursively computing the hash of the child node and comparing it with what is stored in the parent node, and finally checking that $H(UR1)$ is equal to D_1. The current-value proof is generated by computing the inclusion proof based on the last block, e.g. $BLK2$. The append-only proof for showing that the ledger corresponding to D_1 is a prefix of the ledger corresponding to D_2 includes all the common ancestors of $BLK1$ and $BLK1+1$ in the tree whose digest is D_2. In Figure 4, the proof includes d', $UR2$, and D_2. The verification is done by checking that d is the prefix of d', and the path from d' to D_2 is correct.

Discussion. The ledger storage consists of two main components: the ledger structure for storing transactions, and the index for accessing the states. GlassDB has a smaller ledger structure than QLDB and LedgerDB, because its upper POS-tree stores multiple transactions in one block, whereas QLDB and LedgerDB stores one transaction per leaf node of their Merkle trees. SQL Ledger has a smaller ledger structure than GlassDB, because it is based on rows updated within a transaction and transactions committed within a block. However, it uses a hashed chain instead of a Merkle tree, therefore it is less efficient in verification. QLDB and SQL Ledger do not protect the index, thus the server can return stale data, or tamper with the indexes without being detected. LedgerDB constructs clue indexes and protects the size of clue indexes using an additional Merkle Patricia Trie. However, the server can still modify the pointers inside the skip lists to point to stale entries. To detect such tampering, the client needs to verify all entries in the skip lists, thus incurring significant costs. In contrast, the lower

level POS-tree in GlassDB both protects the index and provides efficient access.

3.3.2 Transaction. GlassDB partitions the keys into shards based on their hash values. When a transaction involves multiple shards, GlassDB achieves atomicity using 2PC. Each client is a coordinator. It generates the read set and write set of the transaction, then sends prepare message to the shards. The transaction manager at each shard logs the transaction and responds with a commit or abort based on the concurrency control algorithm. GlassDB uses optimistic concurrency control to achieve serializability. In particular, the read set and write set of concurrent transactions are validated to check for the read-write and write-write conflicts. The shard returns "commit" if there are no conflicts, and returns "abort" otherwise. The client waits for the responses from all shards involved in the transactions, and it resends the messages after a timeout period. If all shards return commits, the client sends the commit messages to the shards, otherwise it sends abort. Each shard then commits or aborts the transaction accordingly, and returns an acknowledgment to the client.

At each shard, the transaction is processed by the transaction manager as follows. All incoming requests are buffered in the transaction queue, waiting to be assigned to available transaction threads. If the queue is full, the transaction is aborted. The transaction threads store the prepared transactions and committed data in the shared memory. The persisting thread persists the committed data asynchronously to the ledger storage.

Asynchronous persistence. Committing transactions to the ledger incurs large overheads due to high contention and long execution time. To address this, GlassDB updates the ledger asynchronously. In particular, when receiving the commit message, the transaction manager stores the transaction data in a multi-version "committed data map" ⟨*key, ver, val*⟩ in memory, and writes to the WAL for durability and recovery. After a timeout, a background thread persists the data in the map to the ledger storage. The persisted data is then removed from the committed data map to keep the memory consumption low. This approach moves the updating of the ledger out of the critical path, thus reducing transaction latency. The trade-off here is that the users cannot retrieve the proofs for data that has not been persisted to the ledger. We explain the verification process in Section 3.3.3.

Transaction batching. The cost of updating and persisting the authenticate data structures is large, even though they are now out of the critical path of transaction execution. It is because both levels of the POS-tree need to be updated and written to disk. To reduce this cost, GlassDB batches multiple committed transactions before updating the ledger. In particular, it uses an aggressive batching strategy that collects independent data from recently committed transactions into a data block. All the blocks created within a time window are appended to the ledger storage. To form a block, the server selects data from the "committed data map" version by version. For a given data version, it can compute the sequence number of the block at which the data will be committed, by adding the current block sequence with the version sequence in the data map. This estimation is used for deferred verification (explained later in Section 3.3.3).

Example. Suppose a server commits three transactions T1, T2, and T3 in timestamp order within a persistence interval. T1 inserts two keys A and B. T2 inserts C and updates A. Lastly, T3 updates B and C. After the commitments, the "committed data map" stores ⟨*A, 1, Va*⟩, ⟨*B, 1, Vb*⟩, ⟨*C, 1, Vc*⟩, ⟨*A, 2, Va'*⟩, ⟨*B, 2, Vb'*⟩, ⟨*C, 2, Vc'*⟩. When the persistence interval expires, the persist thread will divide the data into two batches according to the version number to create two blocks. In particular, it will update the lower POS-tree with ⟨*A, Va*⟩, ⟨*B, Vb*⟩, ⟨*C, Vc*⟩ and ⟨*A, Va'*⟩ ⟨*B, Vb'*⟩ ⟨*C, Vc'*⟩ respectively. The root hash of the updated lower POS-tree together with meta information such as the transaction ID and timestamp are used to create the blocks. Lastly, the two blocks are appended to the upper POS-tree and the digest is updated.

Discussion. Both LedgerDB and SQL Ledger support asynchronous persistence and transaction batching. However, GlassDB maintains the key-value mapping in memory, and only writes WAL to disk during persistence. Therefore, it incurs a lower commitment cost than the other two systems. The batching in GlassDB takes advantage of its index structure to improve performance. In particular, it batches non-overlapping keys from multiple transactions into one block and builds upper level POS-tree on the blocks, which leads to a smaller ledger structure, and consequently lower verification cost, i.e., $O(logB)$. On the other hand, LedgerDB creates a block for each transaction when committing, and batch updates the Merkle tree with multiple blocks periodically. SQL Ledger batches multiple transactions in a Merkle tree and appends a new block created with the Merkle tree root to a hashed chain of blocks. The cost of verification is $O(logT)$ for LedgerDB and $O(logT/B+m)$ for SQL Ledger, where T is the total number of transactions, T/B is the number of transactions for a batch, and m is the number of blocks in the hash-chain scanned. These costs are greater than $O(logB)$.

3.3.3 Verification. Verifying a transaction requires checking both the read set and the write set. To verify the read set, the client checks that the data is correct and is the latest (for example, for the default Get(.) operation). In other words, the server needs to produce current-value proof. To verify the write set, the client checks that the new ledger is append-only, and that the data written to the ledger is correct. In other words, the server needs to produce an append-only proof and an inclusion proof. As an example, consider a client holding a stale digest D_0 commits a transaction that performs read-modify-write on the key K_9. For verification, the client requests four proofs: an append-only proof of current digest D_1 from D_0, a current-value proof of K_9 and V_9^1 with respect to D_1, an inclusion proof of K_9 and V_9^2 with respect to the new digest D_2, and an append-only proof of D_2 from D_1. In GlassDB, the verification requires getting proofs from all participating shards. There is no coordination overhead, because the ledger is immutable with copy-on-write which means read operations can run concurrently with other transactions.

Deferred verification. GlassDB supports *deferred* verification, meaning that transaction verification occurs within a time window, as opposed to immediately. This strategy is suitable for applications that require high performance and can tolerate temporary violations of data integrity. For these applications, the client gets a *promise* from the server containing the future block sequence number where the data will be committed, transaction ID, current

digest, the key and the value. The client can verify the transaction after the block is available by sending a verification request taking the promise as the parameter. The server, on receiving the verification request, will check if the block has been persisted. It generates the proof if the check passes, and returns the proofs and new digest to the client. The client can then verify the integrity of the data as mentioned above. The two-level POS-tree allows the server to batch proofs for multiple keys (especially when they are packed in the same data block). Furthermore, getting the data and the proof can be done at the same time by traversing the tree, which means proof generation can be done with little cost when fetching the data during transaction processing. This is as opposed to LedgerDB requiring the server to traverse one data structure to retrieve the data, and then another data structure to retrieve the proof. To alleviate the burden of deferred verification, in GlassDB, the proof of persisted data is returned immediately during transaction processing, and proof for data to be persisted in future blocks will be generated in deferred verification requests in batch. Deferred verification in GlassDB makes the batching of proof more effective than in LedgerDB and SQL Ledger. This is because the system only needs to access the last block to generate the proof, since the last block covers all current values.

This approach leaves a window of vulnerability during which a malicious database can tamper with the data, but any misbehavior will be detected once the *promised* block number appears in the ledger. GlassDB allows clients to specify customized delay time for verification to find suitable trade-offs between security guarantee and performance according to their needs. Particularly, zero delay time means immediate verification. In this case, the transactions are persisted in the ledger synchronously during the commit phase. This strategy is suitable for applications that cannot afford even a temporary violation of data integrity.

3.3.4 Auditing. While the user verification ensures that the user's own transactions are executed correctly, GlassDB ensures the correct execution of the database server across multiple users. In particular, it relies on a set of auditors, some of which are honest, to ensure that different users see consistent views of the database.

Each auditor performs two important tasks. First, it checks that the server does not fork the history log, by checking that the users receive digests that correspond to a linear history. It maintains a current digest d and block number b corresponding to the longest history that it has seen so far. When it receives a digest d' from a user, it asks the server for an append-only proof showing that d and d' belong to a linear history.

Second, the auditor re-executes the transactions to ensure that the current database states are correct. This is necessary to prevent the server from arbitrarily adding unauthorized transactions that tamper with the states. It also defends against undetected tampering when some users do not perform verification (because they are offline, or due to resource constraints). The auditor starts with the same initial states as the initial states at the server. For each digest d and corresponding block number b, the auditor requests the signed transactions that are included in the block, and the proof of the block and of the transactions. It then verifies the signatures on the transactions, executes them on its local states, computes the new digest, and verifies it against d.

When the auditor receives a digest corresponding to a block number b' which is larger than the current block number b, it first requests and verifies the append-only proof from the server. Next, for each block between b and b', it requests the transactions and verifies that the states are updated correctly. After that, it updates the current digest and block number to d' and b' respectively. Finally, after a pre-defined interval, the auditor broadcasts its current digest and block number to other auditors.

3.3.5 Failure Recovery. GlassDB supports transaction recovery after a node crashes and reboots. In particular, if a node fails before the commit phase, the client aborts the transaction after a timeout. Otherwise, the client proceeds to commit the transaction. When the failed node recovers, it queries the client for the status of transactions, then decides to whether abort or commit. It then checks the WAL for updates that have not been persisted to the ledger storage, and updates the latter accordingly. If the client fails, the nodes have to wait for it to recover, because the 2PC protocol is blocking. We note that this can be mitigated by replacing 2PC with a non-blocking atomic commitment protocol. For example, three-phase commit (3PC) uses an extra phase, allowing participants to communicate among themselves. Another example is non-blocking 2PC [18] that requires participants to forward the vote decisions to every other node. Paxos commit [17] is also non-blocking, in which additional nodes called *acceptors* ensure that the votes are not lost in case of failure. However, these protocols are more complex and incur higher network overheads than 2PC. Integrating them to GlassDB is left as future work.

GlassDB tolerates permanent node failures by replicating the nodes using a crash-fault tolerant protocol, namely Raft. To ensure consistent ledgers across the replicas, GlassDB uses a fixed batch size when creating the blocks during the persistence phase. A timeout is set in case the number of upcoming transactions is insufficient to build a block. When it is expired, a dummy transaction is replicated to all replicas to enforce the block creation. We evaluate the performance impact of node crashes on both schemes in section 5.

3.4 Analysis

3.4.1 Cost analysis. Similar to other verifiable databases, GlassDB incurs additional costs to maintain the authenticated data structure and to generate verification proofs compared to conventional databases. We now analyze the asymptotic computational costs of the main operations in GlassDB.

Persistence. The persistence phase updates the committed data to the two-level POS-tree. The cost of this phase is bounded by the height of the tree, which is the height of upper level plus that of the lower level, i.e., $O(\log B + \log m)$, where B is the number of blocks and m is the number of distinct keys. In contrast, LedgerDB needs to update both the bAMT and ccMPT, which is $O(\log N + \log m)$, where N is the total number of transactions. We note that due to batching, B is much smaller than N. QLDB also updates the Merkle tree over the transactions, thus its cost is $O(\log N)$.

Inclusion proof. To generate an inclusion proof, GlassDB traverses the two-level POS-tree to get the nodes on the path from the leaf to the root. Hence, the cost is $O(\log B + \log m)$. For LedgerDB and QLDB, the proof includes the Merkle proof for a transaction,

and the transaction content. The cost is $O(\log N)$, since the transaction content is small compared to the number of transactions.

Current-value proof. In GlassDB, the lower-level POS-tree captures the entire states, therefore the latest value always appears in the right-most block. The current-value proof is a special case of inclusion proof that includes the right-most block. In other words, the cost is $O(\log B + \log m)$. In contrast, LedgerDB and QLDB do not have protection over indexes. Therefore, their current-value proofs require scanning from one transaction to the latest transaction to check for any new updates on the key. The cost of this is $O(N)$.

Append-only proof. The append-only proof checks if the two digests belong to a linear history. Such a proof contains the nodes created between one digest and another. The cost is $O(\log B)$ for GlassDB, and $O(\log N)$ for LedgerDB and QLDB.

3.4.2 Security analysis. GlassDB is a verifiable ledger database system since it supports the four operations described in Section 2. We now sketch the proof that GlassDB is secure, which involves showing that it satisfies both integrity and append-only property.

For integrity, we first consider Get operation that returns the latest value of a given key (the other Get variants are similar) at a given digest. The user checks that the returned proof π is a valid inclusion proof corresponding to the latest value of the key in the POS-tree whose root is digest. Since POS-tree is a Merkle tree, integrity holds because a proof to a different value will not correspond to the Merkle path to the latest value, which causes the verification to fail. Next, consider the Put operation that updates a key. The user verifies that the new value is included as the latest value of the key in the updated digest. By the property of the POS-tree, it is not possible to change the result (e.g., by updating a different key or updating the given key with a different value) without causing the verification to fail.

For append-only, the auditor keeps track of the latest digest $\text{digest}_{S,H}$ corresponding to the history H. When it receives a digest value $\text{digest}_{S',H'}$ from a user, it asks the server to generate an append-only proof $\pi \leftarrow \text{ProveAppend}(\text{digest}_{S',H'}, \text{digest}_{S,H})$. Since our POS-tree is a Merkle tree whose upper level grows in the append-only fashion, the server cannot generate a valid π if H' is not a prefix of H (assuming $|H'| < |H|$). Therefore, the append-only property is achieved.

In GlassDB, each individual user has a local view of the latest digest digest_{S_l,H_l} from the server. Because of deferred verification, the user sends digest_{S_l,H_l} together with the server's promise during verification. When the latest digest at the server digest_{S_g,H_g} corresponds to a history log H_g such that $|H_g| > |H_l|$, the server also generates and includes the proof $\pi \leftarrow \text{ProveAppend}(\text{digest}_{S_l,H_l}, \text{digest}_{S_g,H_g})$ in the response to the user. This way, the user can detect any *local* forks in its view of the database. After an interval, the user sends its latest digest to the auditor, which uses it to detect *global* forks.

4 BENCHMARK

Even though the verifiable databases expose database-like interface to applications, the existing database benchmarks do not contain verification workloads. To fairly compare GlassDB with other systems, we extend YCSB and TPC-C by including verification workloads.

4.1 YCSB

The existing YCSB workloads include simple put and get operations. We add three more operations, called `VerifiedPut`, `VerifiedGetLatest`, and `VerifiedGetHistory`, and a new parameter `delay`. These operations return integrity proofs that can be verified by the user. The `delay` parameter allows for deferred verification, that is, the database generates the proofs only after the specified duration. When set to 0, the operations return the proof immediately. When greater than 0, the database can improve its performance by batching multiple operations in the same proof.

- `VerifiedPut(k,v,delay)`: returns a promise. The user then invokes `GetProof(promise)` after `delay` seconds to retrieve the proof.
- `VerifiedGetLatest(k,fromDigest,delay)`: returns the latest value of k. The user only sees the history up to `headDigest`, which may be far behind the latest history. For example, the user last interacts with the database, the latter's history digest is `fromDigest`. After a while, the history is updated to another digest `latestDigest`. This query allows the user to specify the last seen history. The integrity proof of this includes an append-only proof showing a linear history from `fromDigest` to `latestDigest`.
- `VerifiedGetHistory(k,atDigest,fromDigest)`: returns the value when the database history is at `atDigest`. `fromDigest` is the last history that user sees. The integrity proof for this query includes an append-only proof from `fromDigest` to `atDigest`.

Based on these operations, we add two new workloads to YCSB. First, *Workload-X* consists of 50% VerifiedPut, 50% VerifiedGetLatest, with 100ms delay. Second, *Workload-Y* consists of 20% VerifiedPut, 40% VerifiedGetLatest, 40% VerifiedGetHistory, with 100ms delay.

4.2 TPC-C

We extend all five types of transactions in TPC-C to the verified versions. Similar to YCSB, each new transaction has a `delay` parameter for specifying deferred verification. When `delay > 0`, each transaction returns a promise which is later used to request the integrity proof. In addition to the five new transactions, we add a new one called `VerifiedWarehouseBalance`, which retrieves the last 10 versions of `w_ytd`. This transaction is possible with verifiable ledger databases because they maintain all historical versions of the data.

5 EVALUATION

5.1 Baselines

We compare GlassDB against four state-of-the-art verifiable ledgers, namely QLDB, LedgerDB, SQL Ledger and Trillian. We do not compare against blockchains due to the different threat models. QLDB, LedgerDB, and SQL Ledger are not open-sourced, thus we implement them based on the documentation available online, or based on the details in the papers. We denote them by QLDB*, LedgerDB*, and SQL Ledger* respectively.

To facilitate *fair performance comparison*, we implement QLDB*, LedgerDB*, SQL Ledger* and GlassDB on top of the same distributed layer, which removes the impact of communication protocols and related implementation details on the overall performance

gaps. In particular, all four systems partition their data over multiple nodes, and they use the same 2PC implementation for distributed transactions. All systems are implemented in C++, using libevent v2.1.12 and Protobuf v3.19.3 for network communication and serialization respectively. We use BLAKE2b as the cryptographic hash function.

*QLDB**. We implement a version of QLDB based on the available documentation. The system consists of a ledger storage and an index storage as shown in Figure 1. The former maintains the transaction log (the WAL) and a Merkle tree built on top of it. The latter maintains a B^+-tree and data materialized from the ledger. When committing a new transaction, the system appends a new log entry containing the type, parameters and other metadata of the transaction, and updates the Merkle tree. After that, the transaction is considered committed and the status is returned to the client. A background thread executes the transaction and updates the B^+-tree with the transaction data and metadata. We implement the inclusion proof, append-only proof, and current-value proof as described in Section 2.3. We do not implement the SQL layer, which is complex and only negatively impacts the overall performance of OLTP workloads.

*LedgerDB**. We implement a version of LedgerDB based on the descriptions in [34]. The system consists of a transaction log, a bAMT, clue indexes, and a ccMPT. We implement bAMT by forcing the update API to take as input a batch of transaction journals. The nodes in bAMT are immutable, that is, each modification results in a new node. We implement the clue index by building a skip-list for each clue, and keeping the mapping of each clue to its skip list's head in memory. To enable queries on keys, we create one clue for each key. As the result, a transaction may update multiple skip lists. The ccMPT is constructed to protect the integrity of the clue index, in which the key is the clue, and the value is the number of leaf nodes in the corresponding skip-list. When committing a new transaction, the system creates a transaction journal containing the type and parameters of the transaction, updates the clue index with it, and returns the status to the client. A background thread periodically updates the bAMT with the committed journals, and updates the ccMPT. The root hash of ccMPT and bAMT are stored in a hash chain. We do not implement the timeserver authority (TSA), which is used for security in LedgerDB, as its complexity adds extra overhead to the system.

SQL Ledger. We implement SQL Ledger based on the original paper [5]. To fairly compare with other systems, we omit the SQL layer and only implement the key-value data model. We use B^+-trees for the indices. When committing data, we first append the log entry to the WAL, then update the current and history B^+-tree. Next, we create a list of data entries for all modified data within the transaction and append them to an in-memory queue. A background thread builds a Merkle tree and creates a transaction entry using the Merkle tree root hash, transaction ID, and timestamp for each data entry list stored in the queue. After that, the Merkle tree for transactions is constructed based on the transaction entries. Finally, a block containing the root hash of the Merkle tree for transactions, block sequence, and previous block hash is appended to a hash chain.

Trillian. We use an implementation of verifiable log-based maps provided by [3, 16]. The system exposes a key-value interface, and consists of two transparency logs and one map. It stores the map of all the keys in a sparse Merkle tree, and asynchronously updates the maps with the new operations in batches. We use the default configurations provided in [3], and use the throughput of the root log as the overall throughput.

5.2 Experiment Setup and Results Summary

All experiments are conducted on 32 machines with Ubuntu 20.04, which are equipped with 10x2 Intel Xeon CPU W-1290P processor (3.7GHz) and 125GB RAM. The machines are on the same rack and connected by 1Gbps network. For each experiment, we collect the measurements after a warm-up of two minutes during which the systems are stable. The results show that GlassDB consistently outperforms QLDB*, LedgerDB*, and SQL Ledger* across all workloads. In particular, compared to LedgerDB* it achieves up to 1.7× higher throughput on the YCSB workload, 1.3× higher throughput for the TPC-C workload, and 1.6× higher throughput for the verification workload. These improvements are due to GlassDB's efficient authenticated data structure, deferred verification, and effective batching.

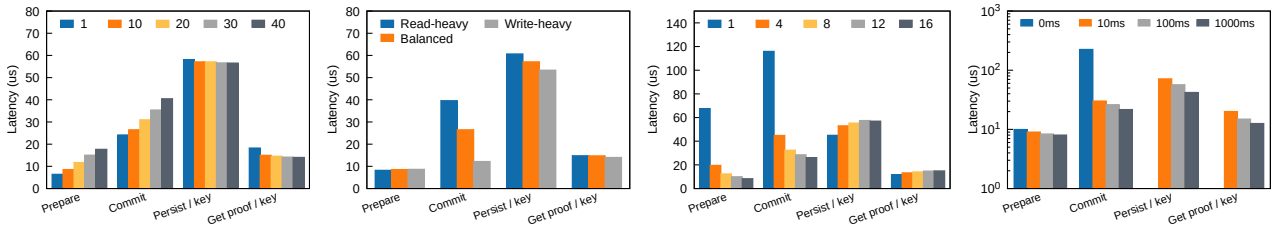
We note that our results are different from what is reported in [34]. In particular, the absolute and relative performances of different systems are not the same, for which we attribute two four reasons. First, the Amazon QLDB service has low performance due to its many limitations such as the maximum transaction size. Moreover, it runs on a serverless platform that is out of the user's control, making it difficult for fair comparisons. Our emulated implementation of QLDB removes these limitations and achieves much higher throughput than the original Amazon QLDB service. Second, [34] lacks sufficient details regarding the experiments. For example, it does not say how it is compared against QLDB (whether the authors used the QLDB service from Amazon, or implemented an emulated one). Third, since the technique details of QLDB are not public, it is not clear from [34] how much the performance gap is due to the differences in hardware, low-level communication protocol, or serialization. Finally, it is not possible to reproduce the results in [34] due to the experiment artifacts being unavailable.

5.3 Micro-Benchmarks

In this section, we evaluate the cost at the server in terms of execution time and storage consumption. We extend the vanilla YCSB benchmark to support transactions, by batching every 10 operations as a transaction. We characterize the workloads as read-heavy (8 reads and 2 writes), balanced (5 reads and 5 writes), and write-heavy(2 reads and 8 writes).

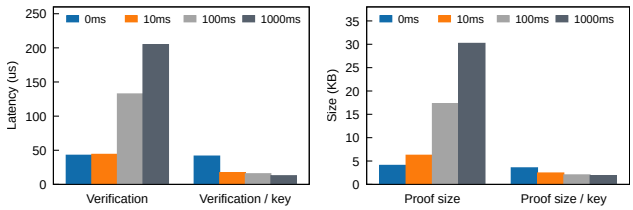
5.3.1 Cost breakdown of GlassDB. We break down the server cost into four phases as described in Section 3.3.2: prepare, commit, persist, and get-proof. For the last two phases, we report the average latency per key, because these phases' costs depend on the number of records in the batch.

Figure 5(a) shows the latency of different phases with varying numbers of operations per transaction (or transaction sizes). We observe that the latency of prepare and commit phase increases as the transactions become larger, which is due to more expensive conflict checking and data operation. Figure 5(b) shows the latency under different workloads. The latency of the prepare phase increases



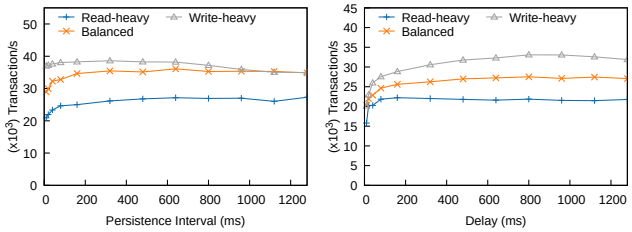
(a) Varying transaction sizes (b) Varying workloads (c) Varying number of nodes (d) Varying persist interval

Figure 5: Latency breakdown at the server.



(a) Verification latency at the client (b) Proof size

Figure 6: Impact of delay time at the client.



(a) Throughput, varying persist interval (b) Throughput, varying verification delay

Figure 7: Impact of delay time on the overall performance.

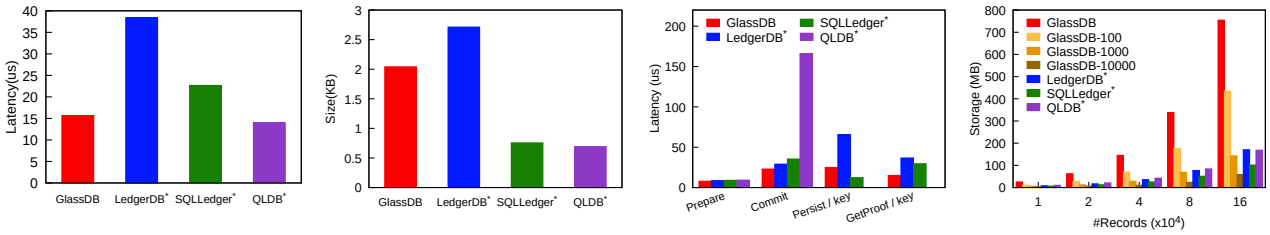
slightly as the workload move from read-heavy to write-heavy because a larger write set leads to more write-write and write-read conflict checking. In contrast, the commit latency of read-heavy workload is much higher than that of write-heavy workload, since read operations are more expensive than the write operations in GlassDB as explained in Section 3.2. Figure 5(c) shows the latency breakdown for varying number of nodes. The latency of the prepare and commit phase decrease as the number of nodes increases, because having more shards means fewer keys to process per node. Figure 5(d) shows the impact of increasing the persist interval. It can be seen that with a longer interval, the persist phase is invoked less frequently, which reduces contention with other phases. As a result, the latency of prepare and commit phases decrease. The persist batch size increases with longer persist intervals, larger transaction sizes, higher write ratio, or with fewer nodes because they lead to more data committed per node. A large persist batch size results in larger data blocks created in the ledger, which in turn increase the batch size of get-proof phases. The results in Figure 5 show that persist and get-proof costs decrease as the persist batch size increases, demonstrating the effectiveness of batching.

We quantify the cost at the client in terms of verification latency and the proof size (which is proportional to the network cost) as shown in Figure 6. We vary the verification delay to show the impact on the costs. The client batches more keys for verification when the delay time is higher, which results in larger proofs as shown in Figure 6(b), and therefore increases the verification latency 6(a). We note that the cost per key decreases with higher delay, demonstrating that batching is effective.

We evaluate the impact of the persistence interval on the overall performance by fixing the client verification delay to 1280ms, while varying the persistence interval from 10ms to 1280ms. Figure 7(a) shows the performance for read-heavy, balanced, and write-heavy workloads. It can be seen that longer intervals lead to higher throughputs for all workloads except for write-heavy workloads. This is because less frequent updates of the core data structure helps reduce contention and increase the effect of batching. For write-heavy workload, however, a long interval causes the update of the core data structure to block transaction execution for longer, which increases the abort rate. In particular, we observe that the abort rate increases to 21.6% at interval of 1280ms for write-heavy workloads, while it remains 1.5% and 3.5% for read-heavy and balanced workloads. Next, we evaluate the impact of verification delay by fixing the persistence interval to 10 ms and varying the delay from 10 ms to 1280 ms. The results are shown in Figure 7(b), in which the throughput increases with larger delays due to proof batching. However, the throughput drops after the peak at 800ms. This is because the batched proof becomes too large that the network cost becomes significant.

5.3.2 Cost breakdown versus other baselines. We compare the latency breakdown of GlassDB with that of three other baselines. We do not compare against Trillian because it does not support transactions.

Figure 8(a) and Figure 8(b) compare the verification latency and per-key proof size of different systems. We measure the proof size per key because each proof is for the entire block containing multiple keys. The per-key proof size of QLDB* and SQL Ledger* are smaller than the rest of the systems since they do not have proofs for the indexes. QLDB* has the smallest per-key proof size of 0.69KB, and therefore lowest verification time, due to its small Merkle tree. The average Merkle tree height for QLDB* is 17, which is higher than that of SQL Ledger* (height of 12). However, the proof of the latter includes additional hashes of blocks between the target block and the latest block. Therefore, its per-key proof size is slighter bigger, i.e., 0.75KB. GlassDB has the smallest tree heights, of 5



(a) Verification time (b) Proof size per key (c) Breakdown latency (d) Storage

Figure 8: Server and client cost versus other baselines.

and 7 on average for the upper level and lower level POS-tree respectively. However, each node of POS-tree is 4× larger than that of a Merkle tree, therefore its proof is large, i.e. 2.1KB. Overall, GlassDB has a comparable verification time as QLDB*, and outperforms SQLLedger*. LedgerDB* has the largest tree heights, which are 17 and 19 for bAMT and ccMPT respectively. Therefore, it has the largest per-key proof size and verification time.

Figure 8(c) shows that GlassDB, LedgerDB*, and SQL Ledger* have lower latency than QLDB* in most phases. The commit latency of QLDB* is especially high because it includes the cost of persisting the authenticated data structure, which explains why Figure 8(c) does not show the cost of the persist phase for QLDB*. In contrast, GlassDB, LedgerDB*, and SQL Ledger* persist the authenticated data structures asynchronously, therefore they have lower latency. GlassDB has the lowest commit latency because it only persists the write-ahead logs and updates an in-memory map when the transaction commits. Both LedgerDB* and SQL Ledger* update the index structures during commit. The skip list update in LedgerDB* incurs less overhead than updating the current and history indexes in SQL Ledger*. GlassDB has lower latency than LedgerDB* in the persist phases because the size of the data committed is smaller. SQL Ledger* commits the least amount of data, therefore it has the lowest persist cost. GlassDB has the lowest per-key latency in the get-proof phase, due to the effective proof batching that reduces the overhead of generating the proof. The batch of get-proof operation for GlassDB is 8 on average, while it is 2 for the others.

5.3.3 Storage consumption. We measure the storage cost of GlassDB with varying batch sizes, and compare them against the two other baselines. Figure 8(d) shows that GlassDB consumes less storage as the batch size increases, because there are fewer saved snapshots. It is most space-efficient when the batch size exceeds 1,000 keys per batch, which can be achieved with 16 nodes and with 100ms delay. LedgerDB* consumes more storage because its authenticated data structure is larger than that of GlassDB.

5.3.4 Impact of design choices. To quantify the impacts of our three novel design choices, we remove these features from GlassDB and compare the resulting system with the baselines. The result is shown in Figure 9. With only the two-level POS-tree, the system (GlassDB-no-DV-no-BA) outperforms QLDB* by 1.2×. By adding deferred verification, the system (GlassDB-no-BA) improves the performance by 2.4×, outperforming LedgerDB* and SQL Ledger*, systems with deferred verification, by 1.3× and 1.4× respectively. By further adding batching, the throughput of the final system (GlassDB) increases by another 1.3×.

5.4 YCSB Workloads

In the experiments, we run read-heavy, balanced, and write-heavy workloads with the number of nodes ranging from 1 to 16, the number of clients ranging from 8 to 80, and delay time ranging from 10ms to 1280ms. We first measure peak throughput by fixing the number of nodes to 16, while increasing the number of clients until the systems are saturated. Figure 10(a) compares the systems under the balanced uniform workload, i.e. Zipf factor is 0. GlassDB outperforms QLDB*, LedgerDB*, and SQL Ledger* by up to 3.7×, 1.7×, and 1.8× respectively. GlassDB, LedgerDB*, and SQL Ledger* are better than QLDB* because they persist the authenticated data structure asynchronously, that is, they avoid updating the Merkle tree in the critical path. Furthermore, they all use batching that helps improve the throughput. GlassDB's batching is more effective at reducing the overall tree heights, therefore the system is more efficient for the get-proof phase and for the verification.

Figure 10(b) shows that all systems scale linearly, and GlassDB achieves the highest throughput. The linear scalability demonstrates that the two-phase commit's overhead is small.

Figure 10(c) displays the throughput comparison under different workloads. It can be seen that GlassDB consistently outperforms the baselines across all workloads. In particular, its throughput increases with a higher write ratio, because write operations are more efficient as data is kept in memory. We note that a higher write ratio leads to more aborts and larger search space for conflicts. For QLDB*, the time to update the Merkle tree is dominant, therefore the abort rate is a key factor that affects the throughput. For LedgerDB* and SQL Ledger*, since the update of the Merkle tree is asynchronous, its reduction of throughput is due to higher disk I/Os and larger conflict search space.

5.5 TPC-C Workloads

We run a mixed workload containing all six types of TPC-C transactions mentioned in Section 4.2, with new order and payment transactions accounting for 42%, and other types accounting for 4% each. We implement the tables in TPC-C on top of the key-value stores in all the systems. In particular, each field in a row is a data unit, and the key-value pair becomes <ColumnName_PrimaryKey, FieldValue>. We make a further optimization to combine fields that are not frequently updated. For example, we combine c_first, c_middle, and c_last to c_name.

Figure 11(a) shows the throughput with an increasing number of clients. GlassDB outperforms QLDB*, LedgerDB*, and SQL Ledger*

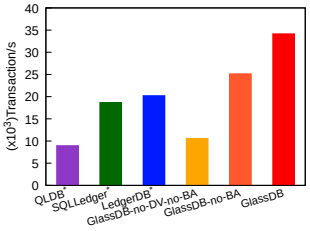
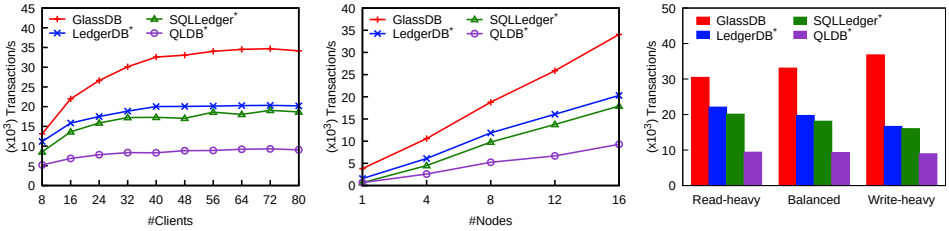
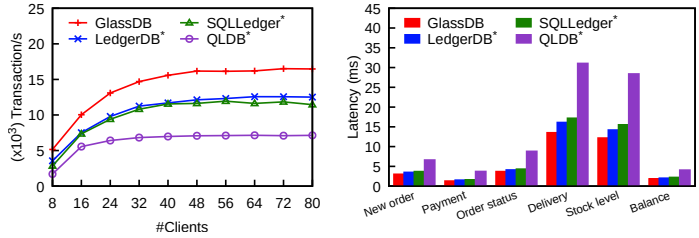


Figure 9: Impact of design choices.



(a) Throughput, 16 nodes (b) Throughput, varying number of nodes (c) Read-write workload

Figure 10: Performance for YCSB uniform workloads.



(a) Throughput, 16 nodes (b) Average latency breakdown

Figure 11: Performance for TPC-C workloads.

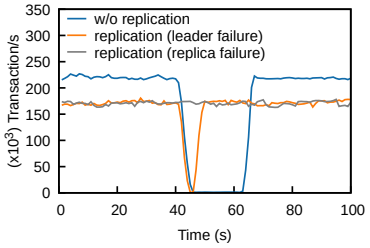
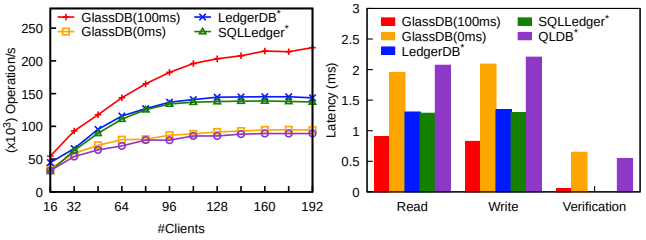


Figure 12: Failure recovery.



(a) Throughput of Workload-X (b) Latency of Workload-X

Figure 13: Workload-X with 16 nodes.

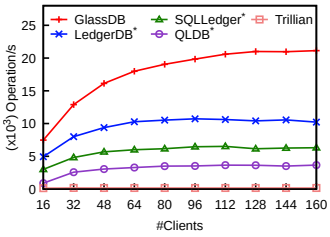


Figure 14: Workload-X on a single node.

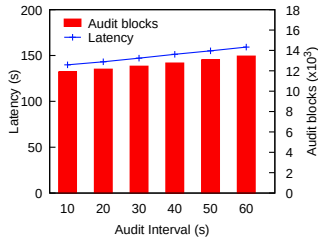


Figure 15: Auditing performance.

by 2.3×, 1.3× and 1.4× respectively. We note that the TPC-C workload has larger and more complex transactions than YCSB workloads, therefore we observe 2.1× lower throughput. In particular, GlassDB achieves peak throughput at 48 clients, as opposed to 64 clients for YCSB. This is because large transaction size involves more nodes and increases the overhead of coordination. Figure 11(b) shows the latency breakdown at the peak throughput for each transaction type; GlassDB consistently has the lowest latency among all types of transactions.

5.6 Verification Workloads

We use workload-X as described in Section 4 to compare GlassDB with QLDB*, LedgerDB*, SQL Ledger*, and Trillian. We omit the results for workload-Y since it displays a similar trend. We run key-value workloads, as Trillian does not support concurrent transactions. For each verified operation, the client performs verification of the proof.

In the distributed settings with 16 nodes, Figure 13(a) shows the throughput for workload-X with an increasing number of clients. GlassDB achieves the highest throughput: 2.5× higher than QLDB*, 1.5× higher than LedgerDB*, and 1.6× higher than SQL Ledger*.

We evaluate the impact of deferred verification by measuring the throughput with 0ms delay, that is, every operation is verified synchronously. Without deferred verification, the throughput is lower than LedgerDB* and SQL Ledger*, and higher than QLDB*. Figure 13(b) shows the latency for each operation. For GlassDB, LedgerDB*, and SQL Ledger*, which use deferred verification, we separate out the cost of verifying one key. GlassDB outperforms the other systems in the read and write latency due to its efficient proofs (smaller proof sizes) and efficient persist phase. Even when combining the cost of transaction execution with that of verification, the total cost of GlassDB, LedgerDB*, and SQL Ledger* are still lower than that of QLDB*. This is because the verification request contains multiple keys, and the two former systems can batch multiple keys in the same proof, whereas QLDB* has one proof per key.

To fairly compare with Trillian, which is a single-node system that only supports key-value abstraction, we use the single-node version of GlassDB, LedgerDB*, SQL Ledger*, and QLDB*. The results are shown in Figure 14. GlassDB outperforms QLDB*, LedgerDB*, SQL Ledger*, and Trillian by up to 5.7×, 2.0×, 3.3× and two orders of magnitude, respectively. The performance gap is due to the cost of the put and get operations in Trillian being

orders of magnitude more expensive. In particular, Trillian stores all data in a separate, local MySQL database instance, thus each operation incurs cross-process overheads.

Finally, we evaluate the cost of the auditing process. We use 16 servers with 64 clients, running the balanced transaction workload. After an interval, an auditor sends `VerifyBlock(.)` requests to the servers and verifies all the new blocks created during the interval. Figure 15 shows the auditing costs with varying intervals from 10s to 60s. Both the latency for verifying the new blocks, and the number of the new blocks grow almost linearly with the audit interval. This is because more blocks are created during a longer interval, and it takes a roughly constant time to verify each block. We remark that the auditing process is expensive, especially when the rate of block creation is high. However, it can be done off the critical path, and is amenable to distributed processing.

5.7 Failure Recovery

In this section, we test the impact of crash failures and the recovery process of GlassDB. We compare the two implementations, i.e., the failure recovery with respect to 2PC and fault tolerance with replication, as described in Section 3.3.5. We set the replica group size to be 3 to tolerate one node failure for the replication setting. The experiment is conducted with 16 nodes and 160 clients. After the performance of the system becomes stable, we let the system continues running for 40 seconds. Next, we kill one node and reboot it after 20 seconds. After that, we let the system run for another 40 seconds. We take measurements of the throughput for every second. The results are shown in Figure 12. In the normal scenario where no crash failure occurs, replication contributes to around 22% overhead. When one node crashes, GlassDB without replication has to abort all transactions accessing keys in the partition hosted by the failed node, therefore, has low throughput until the crashed node is brought back at 60 seconds. For GlassDB with replication, in the case of a leader failure, the system encounters a temporary low throughput because of leader election, syncing of the states, and transaction aborts due to timeouts. It takes around 7 seconds to recover to peak throughput and continues to work as normal. In the case of a replica failure, the system will continue to process the transactions at the peak throughput.

6 RELATED WORK

Verifiable OLAP databases. Zhang et al. [36] propose interactive protocols for verifiable SQL queries. However, their techniques rely on expensive cryptographic primitives Systems that use trusted hardware include EnclaveDB [28], Opaque [38], and ObliDB [14], and they support full-fledged SQL query execution inside trusted enclaves. VeritasDB [30] and Concerto [6] leverage trusted hardware to ensure the integrity of key-value operations. VeriDB [39] extends Concerto to supports general SQL queries. All of these systems make a strong security assumption on the availability and security of the trusted hardware.

Authenticated data structure. Li et al. [21] propose multiple index structures based on Merkle tree and B^+-tree. IntegriDB [37] proposes efficient authenticated data structures that support a wide range of queries such as join and aggregates. We note that these data structures do not guarantee the integrity of the data history.

Blockchain databases. Veritas [1] proposes a verifiable table abstraction, by storing transaction logs on a blockchain. vChain [33] and FalconDB [26] combine authenticated data structures with blockchain, by storing digests of the authenticated index structures in the blockchain. The main disadvantage of blockchain-based systems is that they have poor performance.

7 CONCLUSIONS

In this paper, we described the design space of verifiable ledger databases. We designed and implemented GlassDB that addresses the limitations of existing systems. GlassDB supports transactions, has efficient proofs, and high performance. We evaluated our system against four baselines, using new benchmarks supporting verification workloads. The results show that GlassDB significantly outperforms the baselines.

REFERENCES

[1] Lindsey Allen et al. 2019. Veritas: Shared Verifiable Databases and Tables in the Cloud. In *CIDR*.

[2] Amazon. 2019. *Amazon Quantum Ledger Database.* https://aws.amazon.com/qldb/

[3] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E. Culler, and Raluca Ada Popa. 2019. WAVE: A Decentralized Authorization Framework with Transitive Delegation. In *USENIX Security*.

[4] 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*. 30.

[5] 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.

[6] 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.

[7] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.

[8] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: a distributed storage system for structured data. In *OSDI*.

[10] ConsenSys. 2020. *ConsenSys/quorum: A permissioned implementation of Ethereum supporting data privacy.* https://github.com/ConsenSys/quorum

[11] Scott A. Crosby and Dan S. Wallach. 2009. Efficient Data Structures for Tamper-Evident Logging. In *USENIX Security*.

[12] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-Store: a scalable data store for transactional multi key access in the cloud. In *SoCC*.

[13] 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*.

[14] Saba Eskandarian and Matei Zaharia. 2020. ObliDB: Oblivious Query Processing for Secure Databases. In *VLDB*.

[15] Google. 2020. Certificate Transparency. https://www.certificate-transparency.org/.

[16] Google. 2020. Trillian: general transparency. https://github.com/google/trillian.

[17] Jim Gray and Leslie Lamport. 2006. Consensus on Transaction Commit. *ACM Trans. Database Syst.* 31, 1 (2006), 133–160.

[18] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-blocking Two-phase Commit Protocol.. In *EDBT*. 157–168.

[19] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Reluca Ada Popa. 2021. Mekle2: a low-latency transparency log system. In *IEEE Symposium on Security and Privacy*.

[20] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*.

[21] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*.

[22] Jinyuan Li, Maxwell Krohn, David Mazieres, and Dennis Shasha. 2014. Secure Untrusted Data Repository (SUNDR). In *OSDI*.

[23] Kai Mast, Lequn Chen, and Emin Gün Sirer. 2018. Enabling strong database integrity using trusted execution environments. *arXiv preprint arXiv:1801.01618* (2018).

[24] 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*.

[25] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*.

[26] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. 2020. FalconDB: Blockchain-based Collaborative Database. In *SIGMOD*.

[27] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: A Strongly Encrypted Database System. In *VLDB*.

[28] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database using SGX. In *Security and Privacy*.

[29] Mark Ryan. 2014. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*.

[30] Rohit Sinha and Mihai Christodorescu. 2018. VeritasDB: High Throughput Key-Value Store with Integrity. *IACR* 2018 (2018), 251.

[31] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi , and Pingcheng Ruan. 2018. ForkBase: An Efficient Storage Engine for Blockchain and Forkable Applications. In *VLDB*.

[32] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[33] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases. In *SIGMOD*. 141–158.

[34] 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. In *VLDB*.

[35] 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.

[36] 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.

[37] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for Outsourced Databases. In *CCS*.

[38] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI*.

[39] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. VeriDB: An SGX-based Verifiable Database. In *Proceedings of the 2021 International Conference on Management of Data*. 2182–2194.