

Distributed Verifiable Ledger Database

Technical Field

The present invention relates, in general terms, a distributed verifiable ledger database.

Background

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. 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 online analytical processing (OLAP)-style verifiable databases rely on complex cryptographic primitives that limit the performance or the range of possible queries.

With renewed interest in verifiable databases, recent focus has been placed on online transactional processing (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. Another example is a certificate transparency log, in which a centralized server maintains a tamper-evident, append-only log of public key certificates. The third example is Amazon's Quantum Ledger Database (QLDB) service, 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.

In these systems, 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 technology. As a

consequence, there is no framework within which they can be compared fairly. The second challenge is the lack of database abstraction, that is, transactions. The transparency logs used in existing systems expose a key-value interface. For example, blockchains maintain the states in a key-value storage, certificate transparency and QLDB store the hashes of the certificates and the operations in the log, respectively. This key-value model is too low-level to support general OLTP database 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.

It would be desirable to overcome all or at least one of the above-described problems, or at least provide a useful alternative.

Summary

Disclosed herein is a verifiable ledger database configured to manage transactions of a ledger, comprising a plurality of shards formed by partitioning transaction data and comprising:

- a ledger storage configured to provide access to the transaction data and support a plurality of proofs;
- a transaction manager configured to execute each transaction according to a respective transaction request; and
- a verifier configured to verify the transactions by returning the proofs according to verification requests.

In some embodiments, the ledger storage comprises an upper level POS-tree and a lower level POS-tree, the upper level POS-tree and the lower level POS-tree each comprises a plurality of nodes including at least one root node and at least one leaf node, and wherein the ledger storage comprises a hash chained sequence of blocks and each leaf node of the upper level POS-tree comprises a block number for a respective block.

In some embodiments, the lower level POS-tree is built on states of the verifiable ledger database, and the lower level POS-tree comprises one or more root nodes each containing a respective root hash and a respective said block number.

In some embodiments, the root nodes of the lower level POS-tree are stored as leaf nodes of the upper level POS-tree.

In some embodiments, the verifiable ledger database is configured to retrieve the transaction data from a given block number by locating a corresponding leaf node of the upper level POS-tree with the given block number and then traversing the lower level POS-tree to locate the transaction data.

In some embodiments, the verifiable ledger database is configured to update the transaction data by creating new nodes at the lower level POS-tree and the upper level POS-tree using copy-on-write.

In some embodiments, the transaction data is modelled as a plurality of keys, the verifiable ledger database is configured to partition the keys into the shards based on the hash of the keys.

In some embodiments, the verifiable ledger database is configured to use a two-phase commit (2PC) protocol to ensure atomicity of the transactions.

In some embodiments, the transaction manager is configured to log each transaction and respond with a commit or abort based on a concurrency control algorithm.

In some embodiments, the transaction manager comprises:

- a transaction queue for buffering the transaction requests;
- a plurality of transaction threads for receiving the transaction requests buffered in the transaction queue;
- a shared memory for storing prepared transactions and committed transaction data;
- and
- a persisting thread for persisting the committed transaction data asynchronously to the ledger storage.

The verb form of "persist" as used herein refers to building or updating the authenticated data structure (two-level POS-tree) in the ledger storage (on disk). The word "persist" has been selected to distinguish from that process from the "commit" action that refers to inserting/updating committed data in memory in the commit phase.

In some embodiments, the transaction manager is configured to execute the transactions by:

- assigning the transaction requests buffered in the transaction queue sequentially to the transaction threads if the transaction queue is not full; and
- aborting the transactions if the transaction queue is full.

In some embodiments, the transaction manager is configured to update ledger stored in the verifiable ledger database asynchronously by:

- allowing the shared memory to store the transaction data in a committed data map when the transaction manager receives a commit message;
- writing to a write-ahead-log (WAL);
- allowing the persisting thread to persist the transaction data in the committed data map to the ledger storage to generate persisted data; and
- removing the persisted data from the committed data map.

In some embodiments, the transaction manager is configured to batch the transactions to be committed before updating the ledger by:

- collecting respective data from each transaction to be committed into a data block;
- and
- appending the data blocks created within a time window to the ledger storage.

In some embodiments, collecting the respective data from each transaction to be committed comprises selecting the respective data from the committed data map version by version.

In some embodiments, wherein the proofs comprise one or more of an inclusion proof, a current-value proof, and an append-only proof.

In some embodiments, the verifier is configured to batch the proofs for the transaction data.

In some embodiments, the verifier is configured to return the proofs of the persisted data immediately during transaction processing.

In some embodiments, the verifier is configured to verify the transactions within a time window after the transaction is processed.

In some embodiments, the verifier is configured to verify read set and write set of each transaction.

In some embodiments, the verifiable ledger database comprises one or more auditors for ensuring correct execution of the verifiable ledger database server, wherein each auditor is configured to:

- check whether the verifiable ledger database server forks the history log by checking whether users of the verifiable ledger database receive digests that correspond to a linear history; and

- re-execute the transactions to ensure that current states of the verifiable ledger database are correct.

In some embodiments, the verifiable ledger database, after one or more nodes crash and reboot, is configured to recover said one or more nodes.

Brief description of the drawings

Embodiments of the present invention will now be described, by way of non-limiting example, with reference to the drawings in which:

Figure 1 illustrates the proposed design (hereinafter referred to interchangeably as "LedgeBase" or "GlassDB", each referring to the same system);

Figure 2 illustrates Merkle-tree based transparency logs;

Figure 3 illustrates latency breakdown at the server;

Figure 4 illustrates latency breakdown with different persistence interval;

Figure 5 illustrates impact of delay time at the client;

Figure 6 illustrates impact of delay time on the overall performance;

Figure 7 illustrates server and client cost versus other baselines;

Figure 8 illustrates Performance for TPC-C workloads;

Figure 9 illustrates Workload-X with 16 nodes;

Figure 10 illustrates Workload-X on a single node; and

Figure 11 illustrates auditing performance.

Detailed description

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. Building a practical database from transparency logs, on the other hand, remains a challenge.

Presently disclosed is a distributed database that addresses these limitations under a practical threat model, herein referred to as "LEDGEBASE" or "GlassDB", while also addressing the lack of transaction support and inferior efficiency with existing verifiable ledger databases.

Figure 1 shows the design of GlassDB 100. The distributed database 100 is configured to manage transactions of a ledger by partitioning transaction data into a plurality of shards 102. The transaction data is modelled as a plurality of keys, or key-value tuples.

Partitioning may be based on the hash of the keys, and use two-phase commit (2PC) protocol to ensure the atomicity of cross-shard transactions.

Each shard 102 has three main components: a transaction manager 104, a verifier 106, and a ledger storage 108. A transaction request is forwarded to the transaction manager 104 that is configured to execute each transaction according to a respective transaction request – execution may be performed using a thread pool with optimistic concurrency control to achieve serializability. A verification request is forwarded to the verifier 106 which is configured to verify the transactions by returning proofs according to the verification request. The ledger storage 108 is configured to provide access to the transaction data and support a plurality of proofs. The ledger storage 108 thus maintains the core data structure that provides efficient data access and proof generation.

Each shard 102 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 100 may use write-ahead-log (WAL) to handle application failures. In some embodiments, a replication layer is added on top of the ledger storage 108. This replication layer enables GlassDB to handle node failures. In some such embodiments, GlassDB tolerates permanent node failures by replicating the nodes using a crash-fault tolerant protocol, such as Raft.

GlassDB 100 overcomes some limitations of existing distributed database designs. It supports general transactions, may it versatile in support current and developing 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 one or more auditors that gossip among each other and are trusted, for ensuring correct execution of the verifiable ledger database server. GlassDB achieves high throughputs and small verification costs. Table 1 shows how the system fits in the design space.

System	Abstraction	Threat model	Append-Only Proof	Current-Value Proof	Throughput
QLDB	Transaction	Audit	$O(\log N)$	$O(N)$	Low
LedgerDB	Transaction	Audit	$O(\log N)$	$O(N)$	Medium
Forkbase	Key-value	Audit	$O(N)$	$O(\log m)$	Medium
Blockchain	Transaction	Consensus	$O(1)$	$O(1)$	Low

CreDC	Transaction	Trusted hardware	$O(1)$	$O(1)$	Low
Traillian, ECT, Merkle	Key-value	Audit	$O(\log m)$	$O(\log m)$	Low
GlassDB	Transaction	Audit	$O(\log B)$	$O(\log B + \log m)$	High

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.

The ledger storage 108 adopts a hash-protected index structures. This enables GlassDB 100 to perform comprehensive and efficient protection of the indexes, thereby avoiding one or both of the security issues and high verification overhead or previous distributed ledgers. The ledger storage 108 is also built over the state of data instead of transactions. This structure enables more efficient retrieval and current-value proof generation, while resulting in a smaller data, structure than, for example, Merkle trees of existing systems. The structure also grows more slowly by using batch updates from multiple transactions. Also, by partitioning the data across multiple nodes as discussed below, high throughput is achieved. Lastly, GlassDB 100 uses transaction batching, asynchronous persistence, and deferred verification to speed up transaction processing and verification.

The ledger storage 108 may be structured as a two-level pattern-oriented split tree (or two-level POS-tree) – a Merkle variant. A POS-tree is an instance of Structurally Invariant and Reusable Index (SIRI), combining the Merkle tree and balanced search tree and formed by nodes. The tree is built from a globally ordered sequence of data with a parent node storing the cryptographic hash of its child nodes, the root node thereby containing the digest of the entire tree. Data lookup is performed by traversing the tree. 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 a similar content-defined chunking approach. POS-trees are optimized for high deduplication rates because of its content-defined chunking. Finally, it is immutable, that is, a new tree is created, using copy-on-write, when a node is updated.

With further reference to Figure 1, the ledger storage 108 may comprise an upper level POS-tree 110 and a lower level POS-tree 112. Each tree 110, 112 comprises a plurality of nodes

including at least one root node and at least one leaf node. The lower level POS-tree 112 is built on the database states, and comprises one or more root nodes each called a respective 'data block' (one of which is indicated by numeral 124) that contains a corresponding root hash and other block meta info (e.g. block number where the previous version of data resides) such as block number. The data blocks 124 are stored as the leaves of the upper level POS-tree 110. This POS-tree serves as an index over the data blocks 124, and its root is the digest of the entire ledger. To retrieve a key from a given block number, the block number is acquired from the upper level POS-tree 110, and the lower level POS-tree 112 is then traversed to locate the data.

When updating a key, new nodes are created at both levels using copy-on-write. This authenticated data structure thus provides efficient current-value proofs, in addition to the inclusion and append-only proofs. Since each data block represents a snapshot of the database states, the latest values always appear in the last block, thus current-value proof can be verified with only one block.

QLDB and LedgerDB use Merkle trees to ensure the integrity of the data. However, they offer weak security and incur significant verification overhead. Ledger storage 108 protects the integrity of the indices – database indices and clue indices are not protected in QLDB and LedgerDB. Embodiments of the two-level POS-tree protect both the data, the indices, and the history. In particular, the upper level protects the lineage of the states, while the lower level protects the data and serves as the index. A proof in GlassDB 100 thus includes relevant nodes from the leaf of lower level POS-tree 112 to the root of upper level POS-tree 110. Since both levels are tamper-evident, it is not possible for the server to modify the data or skip some versions without being detected. In addition, the client can check the data block 124 from which the data is fetched, thereby verifying that the data is the latest.

GlassDB 100 partitions the keys into shards 102 based on their hash values. Each client 105 is a coordinator. It generates the read set and write set of the transaction, then sends a "prepare" message to the shards 102. The transaction manager 104 at each shard 102 logs the transaction and responds with a commit or abort based on the concurrency control algorithm. Regarding concurrency control, the read set and write set of concurrent transactions are validated to check for the read-write and write-write conflicts. The shard 102 returns "commit" if there are no conflicts, and returns "abort" otherwise. The client 105

waits for the responses from all shards 102 involved in the transactions, and it resends the messages after a timeout period. If all shards 102 return commits, the client 105 sends the commit messages to the shards 102, otherwise it sends abort. Each shard 102 then commits or aborts the transaction accordingly, and returns an acknowledgment to the client 105.

At each shard 102, the transaction is processed by the transaction manager 104. Incoming requests are buffered in the transaction queue 114. The requests wait in the queue 114 to be assigned to available transaction threads 116. Transaction requests buffered in the transaction queue 114 are sequentially assigned to the transaction threads 116 if the transaction queue 114 is not full. If the queue 114 is full, the transaction is aborted. The transaction threads 116 store the prepared transactions 118 and committed data in the shared memory 120. The persisting thread 122 persists the committed data asynchronously to the ledger storage 108.

To reduce overheads due to contention and execution time, GlassDB 100 updates the ledger asynchronously. When receiving the commit message, the transaction manager 104 stores the transaction data in a "committed data map" $\langle key, ver, val \rangle$ in memory. The *ver* parameter refers to the committed data map potentially being a multi-version committed data map. The transaction manager 104 then writes to the WAL for durability and recovery. After a timeout, a background thread persists the data in the map to the ledger storage 108. The persisted data is then removed from the committed data map to keep the memory consumption low.

Transaction latency is reduced by moving ledger updating out of the critical path for transaction execution, though the cost of updating and persisting the authenticated data structures remains large – both levels of the POS-tree need to be updated and written to disk. GlassDB 100 may reduce the cost by batching multiple committed transactions before updating the ledger. Batching may be performed by collecting independent data from recently committed transactions into a data block 124. At least some, and preferably all, the blocks 124 created within a time window are appended to the ledger storage. To form a block 124, the GlassDB 100 server selects data from the committed data map version by version. For a given data version, GlassDB 100 computes the sequence number of the block 124 at which the data will be committed, by adding the current block sequence with the version sequence in the data map.

By batching multiple transactions, GlassDB affords a smaller Merkle tree than LedgerDB and QLDB, and is therefore more efficient.

Verifying a transaction requires checking both the read set and the write set, with the verifier 106. The proofs involved in verification may include one or more of an inclusion proof, a current-value proof, and an append-only proof. To verify the read set, the client 105 checks that the data is correct and is the latest (for example, for the default Get(.) operation – discussed below). The server thus produces the current-value proof. To verify the write set, the client 105 checks that the new ledger is append-only, and that the data written to the ledger is correct. The server thus produces the append-only proof and the inclusion proof.

The inclusion and current-value proofs in GlassDB 100 may contain the hashes of the nodes in the two-level POS-tree 110 along the path from the leaf to the root, through the specific block and the latest block, respectively. The append-only proof may contain the hashes of nodes along the path where the old root resides. If the old root node does not exist in the new Merkle tree (i.e., because the old Merkle tree is not a complete tree), a proof for its left child node may be generated. To verify the proof, the client 105 computes the digest and compares it with the digest saved locally. The verification requires getting proofs from all participating shards 102. There is no coordination overhead, because the ledger 100 is immutable with copy-on-write which means read operations can run concurrently with other transactions.

Transaction verification can occur within a time window, as opposed to immediately – i.e. deferred verification, in which proofs for the transaction data are batched. This strategy is suitable for applications that require high performance and can tolerate temporary violations of data integrity. For these applications, a promise is sent to the client 105 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 105 can verify the transaction after the block is available by sending a verification request taking the promise as parameter. The server, on receiving the verification request, will check if the block has been persisted. It generates the inclusion proof and append-only proof if the check passes, and returns the proofs and new digest to the client 105. The client 105 can then verify the integrity of the data as mentioned above. The two-level POS-tree 110 allows the server (presently, the verifier 106) to batch proofs for multiple keys. This is especially the case when the keys are

packed in the same data block 124. Furthermore, getting the data and the proof can be done at the same time by traversing the tree 110, which means proof generation can be done with little cost when fetching the data during transaction processing. 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.

Any malicious interaction with the ledger 100 will be detected once the promised block number appears in the ledger 100. GlassDB 100 allows clients 105 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 to the ledger storage 110 synchronously during the commit phase. This strategy is suitable for applications that cannot afford even a temporary violation of data integrity.

GlassDB 100 thus inherits the verifiability of transparency logs while supporting transactions and offering high performance. The transparency log 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.

In comparing verifiable ledger databases, a unified framework is required. Presently, A design space we established comprising three dimensions: an abstraction dimension for capturing the interface exposed to the users, which can be either key-value or general transactions; a threat model dimension including different security assumptions; and a performance dimension including design choices that affect the integrity proof sizes and the overall throughput. The benchmark for comparison extends traditional database benchmarks, namely YCSB and TPC-C, with additional workloads containing verification requests on the latest or historical data.

GlassDB 100 supports distributed transactions and is designed to enable database abstraction and achieve high performance while retaining security. 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. Each node of GlassDB has multiple threads for processing transactions and generating proofs in parallel, and a single thread for updating the ledger storage.

As mentioned above, the transparency log is an append-only log accompanied by a Merkle tree. Figure 2 shows an example of a transparency log 100, 102 at two different times: one transparency log 100 with 3 elements and another (102) with 6 elements. Each leaf 104 represents an operation, for example updating a key-value tuple. The proof for an append operation is the new Merkle root tree. Regarding running the proofs: the inclusion proof is the Merkle path from the corresponding leaf to the root. The cost of this proof is $O(\log(N))$ where N is the size of the log. In the example, the proof that element 2 exists in the log consists of the hashes of node 1 and b . Regarding the append-only proof, the proof consists of nodes for reconstructing both trees, which has the complexity of $O(\log(N))$. In the example, to prove that c is included in l , the proof includes the hashes of node $e, 3, 4, k$. The first three are sufficient to compute c , and all four are sufficient to compute l . Regarding the current-value proof, the proof includes all the leaves of the tree, which has the complexity of $O(N)$. In our example, suppose the latest value for key k is set at node 3. Given l and the tuple, the user has to fetch all 6 elements and check that nodes 4,5,6 do not update the tuple.

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

Table 1 above compares existing verifiable ledger databases, according to the present design space a proof cost, that are built on top of transparency logs. QLDB and LedgerDB are mentioned above. Forkbase is a versioned, key-value storage system implementing transparency maps. Blockchain systems assume the majority of trusted providers in a decentralized setting. CreDB assumes that the server can create trusted execution environments backed by trusted hardware. Regarding public-key transparency logs: Trillian combines transparency logs and maps to implement new primitives called verifiable log-based map. ECT and Merkle are similar to Trillian in the present design space but improve Trillian by adding support for privacy, revocation (non-inclusion proofs), and reducing the audit cost.

Having described the structure and behaviour of the GlassDB database, now the life cycle of a transaction at the server will be described. That life cycle is 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.

Interaction with the server and the one or more auditors of GlassDB may use a plurality of APIs. The APIs may include:

- Init (sk): initializes the client with private key *sk* for signing the transactions, and sends public key *pk* to the auditors for verification.
- BeginTxn(): starts a transaction. It returns a unique transaction ID *tid* based on client ID and timestamp.
- Get(*tid*, *key*, (*timestamp* | *block_no*)): returns the latest value of the given key (default option), or the value of the key before the given timestamp or block number.
- Put (*tid*, *key*, *value*): buffers the write for the ongoing transaction.
- Commit(*tid*): signs and sends the transaction to the server for commit. It returns a promise.
- Verify (promise): requests the server for a proof corresponding to the given promise, then verifies the proof.
- Audit(digest, *block_no*): sends a digest of a given block to the auditors.

The one or more auditors use the following APIs to ensure that the database server is working correctly.

- VerifyBlock (digest, *block_no*): requests 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.
- VerifyDigest (digest, *block_no*): verifies 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 Verify-Block to verify all the blocks in between.

- Gossip(digest, block_no): broadcasts the current digest and block number to other auditors.

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, LaedgeBase relies on a set of auditors, some of which are honest, to ensure different users see consistent views of the database.

Each auditor performs two tasks. The first task involves checking that the server does not fork the history log. This may be achieved in a variety of ways. In one embodiment, the first task involves checking that the users receive digests that correspond to a linear history. The auditor 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.

The second task performed by an auditor is the re-execution of transactions to ensure that the current database states are correct. This prevents 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 – e.g. 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. The auditor 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 (i.e. the auditor gossips).

Lastly, some embodiments of GlassDB do not tolerate permanent node failures but instead support recovery after a node crashes and reboots. In particular, if a node fails before the commit phase, the client 105 aborts the transaction after a timeout. Otherwise, the client 105 proceeds to commit the transaction. When the failed node recovers, the failed node:

- queries the client 105 for the status of transactions, then decides to whether abort or commit.
- checks the WAL for updates that have not been persisted to the ledger storage 108, and updates the latter accordingly.

If the client 105 fails, the nodes have to wait for it to recover, because the 2PC protocol is blocking – the delay resulting from this wait can be mitigated by replacing 2PC with a non-blocking atomic commitment protocol, such as three-phase commit (3PC).

In other embodiments, GlassDB is extended to tolerate permanent node failures by replicating the nodes using a crash-fault tolerant protocol such as Paxos and Raft.

Similar to other verifiable databases, GlassDB incurs additional costs to maintain the authenticated data structure and to generate verification proofs compared to conventional databases. For integrity, first considering the 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 the 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, $digest_{s,H}$, corresponding to the history log H . When it receives a digest value $digest_{s',H'}$ from a user, it asks the server to generate an append-only proof $\pi \leftarrow \text{ProveAppend}(digest_{s',H'} \text{ digest}_{s,H})$. 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 S_{S_l, H_l} from the server. Because of deferred verification, the user sends digest $digest_{S_l, H_l}$ 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 $\pi \leftarrow \text{ProveAppend}(digest_{S_l, H_l}, digest_{S_g, H_g})$ 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.

In some embodiments, a delay parameter can be used to specify a delay for performing a proof. Where the delay parameter is 0, the proof is performed immediately. Where the delay parameter is greater than 0, multiple operations can be batched in the same proof, thereby improving performance. Performing the operation $\text{VerifiedPut}(k, v, \text{delay})$: returns a promise. The user then invokes $\text{GetProof}(\text{promise})$ after delay seconds to retrieve the proof. $\text{VerifiedGetLatest}(k, \text{fromDigest}, \text{delay})$: 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. Relatedly, $\text{VerifiedGetAt}(k, \text{atDigest}, \text{fromDigest})$: 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.

Figure 3(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 3(b) shows the latency under different workloads. The latency of the prepare phase increases 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 . Figure 3(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.

Notably, the persist and get-proof latency remain almost constant for the three experiments in Figure 3. This is because the latency of get-proof and persist only depend on the number of batched keys. The results are measured at peak throughputs, and it is observed that the number of keys included for persistence and proof generation per node is similar for different number of nodes. Therefore, the latency of the two operations are constant.

Figure 4 shows the impact of varying the delay time on the server costs. As the delay time increases, the server handles the verification request and ledger persistence less frequently. As a result, there is less resource contention for other phases, which leads to lower latency for prepare and commit phases. For the persist phase, the higher delay means a larger batch, therefore lower persist latency per key.

The cost at the client can also be quantified in terms of per-key verification latency and the proof size (which is proportional to the network cost) as shown in Figure 5. The client batches more keys for verification when the delay time is higher, which results in larger proofs as shown in Figure 5(b), and therefore increases the verification latency 5(a). We note that the cost per key decreases with higher delay, demonstrating that batching is effective.

The impact of the persistence interval on the overall performance can be compared by fixing the client verification delay – delay specified before verification takes place, to benefit from batching – while varying the persistence interval (time period). Figure 6(a) shows the performance for read-heavy, balanced, and write-heavy workloads. It can be seen that longer intervals lead to higher throughputs across all workloads. This is because less frequent updates of the core data structure helps reduce contention and increase the effect of batching.

The impact of verification delay can also be assessed, by fixing the persistence interval and varying the delay. The results are shown in Figure 6(b), in which the throughput increases with larger delays. However, for the write-heavy and balanced workloads, the throughput

drops after the peak at 160 ms and 320 ms respectively. This is because a larger delay results in more keys to be verified, which increases contention with the transaction execution. For write-heavy workload, the number of keys to be verified increases the fastest as there are more updates.

GlassDB was benchmarked against Emulated LedgerDB and Emulated QLDB for experimental purposes. GlassDB has comparable latency to Emulated LedgerDB and lower latency than Emulated QLDB in most phases. This is due to asynchronous persistence of authenticated data structures. GlassDB has the lowest commit latency because it only persists the write-ahead logs when the transaction commits. Furthermore, it has lower latency in the persist and getproof phases because the size of the data structure is smaller.

Figure 7(b) and Figure 7(c) compare the verification latency and per-key proof size of different systems. Emulated QLDB has the smallest proof size, with GlassDB having smaller latency than that of Emulated LedgerDB since the authenticated data structure in GlassDB is smaller, leading to smaller verification time shown in Figure 8(b), and GlassDB has more keys per block, thereby reducing the per-key proof.

Figure 7(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 100 ms delay. Emulated LedgerDB consumes more storage because its authenticated data structure is larger than that of GlassDB.

Figure 8(a) and 8(c) show the throughput with an increasing number of clients and server nodes respectively, under a mixed workload containing various transaction types, including new order and payment transactions. GlassDB outperforms Emulated LedgerDB and Emulated QLDB. The average latency of the systems are shown in Figure 8(b). The results are consistent with the throughput performance. Figure 8(d) shows the latency breakdown at the peak throughput for each transaction type; GlassDB consistently has the lowest latency among all types of transactions.

Figure 9(a) shows the throughput for a particular workload with 16 nodes and an increasing number of clients. GlassDB achieves the higher throughput than Emulated QLDB and Emulated LedgerDB. Without deferred verification, the throughput is lower than Emulated

LedgerDB. Figure 9(b) shows the latency for each operation. GlassDB outperforms the other systems in the read and write latency due to its efficient proofs (smaller proof sizes) and efficient persist phase.

Under the same workload in a single node system, GlassDB outperformed Emulated QLDB and Trillian as reflected in Figure 10.

Finally, the cost of the auditing process is evaluated. Using 8 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 11 shows the auditing costs with varying intervals from 20 ms to 100 ms. Both the latency for verifying the new blocks, and the number of new blocks grows 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.

In summary, embodiments of GlassDB address many of the limitations of existing systems. GlassDB supports transactions, has efficient proofs, and high performance. GlassDB was evaluated against three baselines, using new benchmarks supporting verification workloads. The results show GlassDB significantly outperforms the baselines.

It will be appreciated that many further modifications and permutations of various aspects of the described embodiments are possible. Accordingly, the described aspects are intended to embrace all such alterations, modifications, and variations that fall within the spirit and scope of the appended claims.

Throughout this specification and the claims which follow, unless the context requires otherwise, the word “comprise”, and variations such as “comprises” and “comprising”, will be understood to imply the inclusion of a stated integer or step or group of integers or steps but not the exclusion of any other integer or step or group of integers or steps.

The reference in this specification to any prior publication (or information derived from it), or to any matter which is known, is not, and should not be taken as an acknowledgment or admission or any form of suggestion that that prior publication (or information derived from

it) or known matter forms part of the common general knowledge in the field of endeavour to which this specification relates.

Claims

1. A verifiable ledger database configured to manage transactions of a ledger, comprising a plurality of shards, each shard being formed by partitioning transaction data and comprising:
 - a ledger storage configured to provide access to the transaction data and support a plurality of proofs;
 - a transaction manager configured to execute each transaction according to a respective transaction request; and
 - a verifier configured to verify the transactions by returning the proofs according to verification requests.
2. The verifiable ledger database of claim 1, wherein the ledger storage comprises an upper level POS-tree and a lower level POS-tree, the upper level POS-tree and the lower level POS-tree each comprises a plurality of nodes including at least one root node and at least one leaf node, and wherein the ledger storage comprises a hash chained sequence of blocks and each leaf node of the upper level POS-tree comprises a block number for a respective block.
3. The verifiable ledger database of claim 2, wherein the lower level POS-tree is built on states of the verifiable ledger database, and the lower level POS-tree comprises one or more root nodes each containing a respective root hash and a respective said block number.
4. The verifiable ledger database of claim 3, wherein the root nodes of the lower level POS-tree are stored as leaf nodes of the upper level POS-tree.
5. The verifiable ledger database of claim 4 being configured to retrieve the transaction data from a given block number by locating a corresponding leaf node of the upper level POS-tree with the given block number and then traversing the lower level POS-tree to locate the transaction data.

6. The verifiable ledger database of claim 4, wherein the verifiable ledger database is configured to update the transaction data by creating new nodes at the lower level POS-tree and the upper level POS-tree using copy-on-write.
7. The verifiable ledger database of any one of claims 1 to 6, wherein the transaction data is modelled as a plurality of keys, the verifiable ledger database is configured to partition the keys into the shards based on the hash of the keys.
8. The verifiable ledger database of claim 7 being configured to use a two-phase commit (2PC) protocol to ensure atomicity of the transactions.
9. The verifiable ledger database of any one of claims 1 to 8, wherein the transaction manager is configured to log each transaction and respond with a commit or abort based on a concurrency control algorithm.
10. The verifiable ledger database of any one of claims 1 to 9, wherein the transaction manager comprises:
 - a transaction queue for buffering the transaction requests;
 - a plurality of transaction threads for receiving the transaction requests buffered in the transaction queue;
 - a shared memory for storing prepared transactions and committed transaction data; and
 - a persisting thread for persisting the committed transaction data asynchronously to the ledger storage.
11. The verifiable ledger database of claim 10, wherein the transaction manager is configured to execute the transactions by:
 - assigning the transaction requests buffered in the transaction queue sequentially to the transaction threads if the transaction queue is not full; and
 - aborting the transactions if the transaction queue is full.
12. The verifiable ledger database of claim 10 or 11, wherein the transaction manager is configured to update ledger stored in the verifiable ledger database asynchronously by:

allowing the shared memory to store the transaction data in a committed data map when the transaction manager receives a commit message;
writing to a write-ahead-log (WAL);
allowing the persisting thread to persist the transaction data in the committed data map to the ledger storage to generate persisted data; and
removing the persisted data from the committed data map.

13. The verifiable ledger database of claim 12, wherein the transaction manager is configured to batch the transactions to be committed before updating the ledger by:
collecting respective data from each transaction to be committed into a data block; and
appending the data block that are created within a time window to the ledger storage.
14. The verifiable ledger database of claim 13, wherein collecting the respective data from each transaction to be committed comprises selecting the respective data from the committed data map version by version.
15. The verifiable ledger database of any one of claims 1 to 14, wherein the proofs comprise one or more of an inclusion proof, a current-value proof, and an append-only proof.
16. The verifiable ledger database of any one of claims 1 to 15, wherein the verifier is configured to batch the proofs for the transaction data.
17. The verifiable ledger database of any one of claims 12 to 16, wherein the verifier is configured to return the proofs of the persisted data immediately during transaction processing.
18. The verifiable ledger database of any one of claims 1 to 17, wherein the verifier is configured to verify the transactions within a time window after the transaction is processed.
19. The verifiable ledger database of any one of claims 1 to 17, wherein the verifier is configured to verify read set and write set of each transaction.

20. The verifiable ledger database of any one of claims 1 to 19 comprising one or more auditors for ensuring correct execution of the verifiable ledger database server, wherein each auditor is configured to:
- check whether the verifiable ledger database server forks the history log by checking whether users of the verifiable ledger database receive digests that correspond to a linear history; and
 - re-execute the transactions to ensure that current states of the verifiable ledger database are correct.
21. The verifiable ledger database of any one of claims 2 to 20, after one or more nodes crash and reboot, being configured to recover said one or more nodes.