ViDB: Cost-Efficient Ledger Database At Scale

Paper ID 587

Abstract

Ledger databases combine traditional database functionality with tamper-evident guarantees, enabling applications to maintain auditable records of transactions and data modifications. While existing systems provide essential integrity properties, they face significant scalability challenges as data volumes grow, suffering from escalating storage costs, I/O amplification, and computational overhead that limit their practical deployment at scale.

We present ViDB, a cost-efficient ledger database that maintains consistent performance regardless of data volume. ViDB introduces three key innovations: (1) a Time-Partitioned B+-tree Forest (TPBF) that bounds computational complexity by limiting tree heights through configurable capacity partitioning; (2) Location-based Versioned Page Storage (LVPS) that eliminates write amplification by storing pages directly at physical locations without background compaction; and (3) in-memory garbage collection that enables efficient version operations without intensive disk I/O. Our evaluation against five state-of-the-art ledger databases shows ViDB achieves 2× to 20× improvements in transaction throughput while reducing storage consumption by over 70%. When integrated into Hyperchain, a commercial blockchain platform, ViDB significantly outperforms existing solutions for both payment and smart contract workloads, demonstrating its practical effectiveness at scale.

CCS Concepts

• **Information systems** → **Data management systems**; • **Security and privacy** → *Database and storage security*.

Keywords

Ledger database, cost efficiency, verifiable query, data versioning

ACM Reference Format:

Paper ID 587 . 2026. ViDB: Cost-Efficient Ledger Database At Scale. In *Proceedings of International Conference on Management of Data (SIGMOD '26).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Ledger databases are increasingly adopted in diverse applications, such as healthcare [34], e-sports [13], advertising [16], and federated learning [22], where data immutability and verifiability are critical. While conventional blockchain-based systems [33] offer these guarantees, they often suffer from performance bottlenecks due to decentralization and runtime verification overheads—mechanisms that might be unnecessary in practical deployments [4, 39]. In contrast, ledger databases aim to strike a balance between integrity guarantees and high performance, providing flexible query support that better serves production workloads.

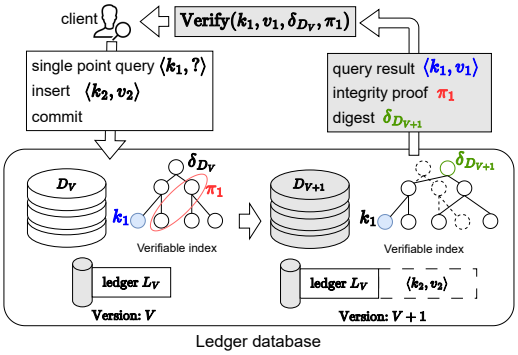


Figure 1: Ledger database functionality.

To achieve efficiency, ledger databases integrate traditional database functionality with verifiable storage. They enable rudimentary operations such as record insertion, point lookups, and range scans, while ensuring tamper evidence through proof generation and versioning. An example is shown in Figure 1, where the client first queries the current version of data D_V using keys k_1 with verification, then inserts two records with keys k_2, creating a new version of data D_{V+1}. These properties are enabled by *verifiable index*, a key component that unifies data indexing and authentication [31, 40].

The verifiable index combines data indexing with an authenticated data structure (ADS), typically realized as Merkle tree variants [23, 41]. It supports both multi-version data management and verifiability with cryptographic proofs. By co-designing data access paths and integrity guarantees, the verifiable index enables performance-aware verifiable storage, at the cost of importing significant I/O and computation overheads. Additionally, these costs grow with data volume and hinder broader adoption, especially at scale. Reducing them without compromising verifiability or database functionality remains a pressing problem. As summarized in Table 1, three key challenges hinder the applicability of ledger databases:

(1) Large storage consumption: Growing data volume leads to a larger verifiable index, and keeping multiple versions further exacerbates the storage space cost. Prior studies have shown that the storage usage grows significantly with the number of records and versions [41, 42]. Under strict storage constraints, aggressive version pruning may be necessary. However, existing systems treat version pruning as a low-frequency operation [31], causing new versions to stall when storage space is limited. Furthermore, the existing verifiable index stores each node with its identifier on disk to enable retrieval by identifier, incurring additional storage overhead for the identifiers [1].

SIGMOD '26, May 31–June 05, 2026, Bengaluru, India

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Table 1: Compare VIDB with existing ledger databases (the shaded background and bold font indicate a better design).

Challenge		LedgerDB [39]	SQLLedger [4]	QLDB [1]	QMDB [43]	LETUS [31]	**VIDB**(Ours)
Large storage consumption	version pruning	infrequent	not support	not support	infrequent	infrequent	**frequent**
	additional storage for identifier	✓	✓	✓	✓	✓	×
Intensive disk I/O	background compaction	✓	✓	✓	×	×	×
	garbage collection	disk I/O					**in-memory**
Heavy computational cost	index tree height	increase with data volume					**bounded**
	version backtracking cost	linear			not supported	**constant**	**constant**

(2) Intensive disk I/O: Current ledger databases suffer from write amplification when storing verifiable index nodes. The amplification effect becomes more significant when the data volume increases. The randomness of node identifiers requires background compaction for ordering, causing write amplification that incurs intensive disk I/O for repeatedly rewriting nodes [1, 39]. Additionally, the garbage collection for version operations, including pruning and rollback, consumes significant I/O bandwidth, because it scans all the files to find pages to remove [43].

(3) Heavy computational cost: With the increasing complexity of data operations, the ledger database consumes more computational resources to maintain the processing performance. Current designs of verifiable index use a single tree for all data records [41], and when data volume increases, the tree height grows, and so does the computational complexity of data operations and the proof size. In addition, some ledger databases store multiple versions as linked lists, and trace back from the latest version to reach a historical version [4, 39]. As a result, querying an older version requires a longer path of backtracking and costs more time.

In this work, we present **VIDB**, a ledger database that can scale with data volume while maintaining low hardware cost. Specifically, VIDB addresses the aforementioned challenges with the following design principles. (1) VIDB designs the verifiable index structure based on the B^+-tree to utilize its range query efficiency and balanced structure, and reduce the complexity of data operations and proof size. The VIDB verifiable index is a forest instead of a single tree, where the index tree is partitioned according to the creation time. As data volume increases, the index tree height is bounded, and the disk I/O and computational costs do not increase monotonically. The verifiable index also supports data versioning and eliminates the backtracking cost for historical version queries. (2) VIDB stores the verifiable index structure with a location-based page indexing scheme, which writes each node only once to the storage space, avoiding background compaction and minimizing write amplification. Our analysis shows that VIDB achieves lower write amplification compared to state-of-the-art ledger databases. Nodes are retrieved directly by their disk location, eliminating the storage consumption for information such as the identifiers. (3) VIDB designs compact memory data structures to track the lifecycle of pages, facilitating the garbage collection. VIDB performs garbage collection entirely through in-memory operations, and avoiding intensive disk I/O.

Our experiments show that, compared with existing verifiable ledger databases, VIDB achieves 2× to 20× improvement in transaction performance while reducing over 70% storage cost. We deploy VIDB on Hyperchain [15] to support high-performance, large-scale blockchain data. In summary, we make the following contributions.

- We present a cost-efficient ledger database that co-designs index and storage layers to achieve bounded complexity and reduced write amplification, addressing fundamental scalability limitations of existing systems.
- We design a verifiable index that partitions data across multiple trees with the integrity proof, keeping the tree height under a configurable bound to reduce the write amplification.
- We introduce a storage layer using direct physical addressing that eliminates write amplification and enables in-memory garbage collection without intensive disk I/O.
- We conduct extensive experiments against five state-of-the-art systems, demonstrating 2× to 20× throughput improvements and up to 70% storage reduction. Integration with a commercial system, Hyperchain, further validates practical effectiveness in production blockchain environments.

Syllabus. § 2 provides the preliminary. § 3 describes the architecture of VIDB and § 4 discusses VIDB detailed design in achieving cost-efficient query processing, data versioning and garbage collection. We present our experimental results in § 5 to show the efficiency of VIDB. § 6 presents related works and § 7 concludes the paper.

2 Preliminary

In this section, we revisit the ledger database and its functionality. We conclude the section with a discussion and rethink on the implementation choices in the design space.

2.1 Ledger database functionality

Ledger databases are required to protect the integrity of both the data content and data history [4, 39, 40]. The ledger database maintains a key/value dataset D and an append-only log, called the *ledger* L. The ledger database processes transactions through data operations, including insert, update, and query. On receiving commitments from a client, the ledger database consolidates the current D and stores it as a new version. As a result, there are multiple versions of D, and we mark version V of the state as D_V. Figure 1 shows that the ledger database executes a client transaction on the data of the latest version D_V. The ledger database retrieves the query results from D_V and generates an *integrity proof* π to show that the result has not been tampered with. Then the ledger database creates a dataset of the new version D_{V+1} and appends the inserted records to the ledger L. The ledger database generates a *digest* $\delta_{D_{V+1}}$ that identifies the new version $V+1$, and publishes the digest with certification for clients. To verify a query result (k, v) is from D_V, the client relies on a Verify function, the integrity proof π returned with the query result, and the certified digest δ_{D_V}. The

Table 2: Processing costs of varieties of Merkle trees.

OPs / Index	insert complexity	query processing complexity		proof size	
		point query	range query	point query	range query
MPT	$O(\|k\|)$	$O(\|k\|)$	$O(L_R\|k\|)$	$O(\|k\|)$	$O(N_R\|k\|)$
MBT	$O(\log_m N_B + \frac{N_{\text{rec}}}{N_B})$	$O(\log_m N_B + \log_2 \frac{N_{\text{rec}}}{N_B})$	$O(L_R(\log_m N_B + \log_2 \frac{N_{\text{rec}}}{N_B}))$	$O(\log_m N_B + \frac{N_{\text{rec}}}{N_B})$	$O(N_R(\log_m N_B + \frac{N_{\text{rec}}}{N_B}))$
Merkle B$^+$-tree	$O(\log_m N_{\text{rec}})$	$O(\log_m N_{\text{rec}})$	$O(N_R + \log_m N_{\text{rec}})$	$O(\log_m N_{\text{rec}})$	$O(\log_m N_{\text{rec}})$
TPBF (Ours)	$O(\log_m N_{\text{cap}})$	$O(\log_m N_{\text{cap}})$	$O(N_R + \log_m N_{\text{cap}})$	$O(\log_m N_{\text{cap}})$	$O\left(N_{RS} \log_m N_{\text{cap}}\right)$

Verify function $\text{Verify}(k, v, \delta_{D_V}, \pi)$ returns 1 if (k, v) is proved to be from D_V, and 0 otherwise.

2.2 Verifiable index

The ledger database relies on the verifiable index to generate integrity proofs. A verifiable index typically takes the form of a Merkle tree-like data structure. In a Merkle tree [23], a leaf is a data record, and an internal node contains the hash of the concatenation of all its children. Proof is constructed using nodes on the path from the root to the record. When creating a new version, the root is recalculated and published as the digest.

There are three representative Merkle tree varieties [41]. Merkle Patricia trie (MPT) [33] and Merkle Bucket (MBT) [3] are respectively adopted by the widely used systems Ethereum and Fabric Hyperledger, while Merkle B$^+$-tree [21] is seen in academic [28, 40].

MPT splits the key k into a hexadecimal string, which corresponds to the path from the MPT root to the leaf storing the record $< k, v >$. MBT hashes the records into N_B buckets, then uses the buckets as leaf nodes and constructs a Merkle tree. The Merkle B$^+$-tree introduces Merkle hashes for integrity protection into the B$^+$-tree. We use m to denote the Merkle tree fanout.

To quantify the performance of Merkle tree varieties, we analyze the complexities of *data insertion*, *point query* for a key k, and *range query* that scans key range $[k_{\text{lb}}, k_{\text{ub}}]$. Table 2 summarizes the analysis result. $|k|$ is the key length, and N_{rec} is the number of records. For a range query, $L_R = (k_{\text{ub}} - k_{\text{lb}})$ is the range size, and N_R is the number of records in the query result set.

Discussion: Costs of Merkle tree varieties. Higher insert and query complexities require more computational resources to deploy the ledger database. Larger proof size requires more I/O bandwidth to transfer the proof for data validation. From Table 2, we observe that, for all the verifiable indices, the complexity and proof size are all workload-driven parameters, i.e., the key length $|k|$ and number of records N_{rec}. Even worse, for MPT and MBT, the complexity of the range query is also related to the range size L_R. Only the Merkle B$^+$-tree can perform slightly better in the range query since the complexity and proof size are not influenced by L_R. In conclusion, the computational and I/O bandwidth costs of current Merkle tree structures are determined by the workload and keep increasing as data volume scales up.

For comparison, we list the cost of ViDB in Table 2. As we will formally introduce in § 3, N_{cap} is the capacity of a tree, which is configurable and far less than N_{rec}, i.e., $N_{\text{cap}} << N_{\text{rec}}$. N_{RS} is the number of non-empty result sets for a range query, and $1 \leq N_{RS} \leq N_R$. The complexity of ViDB is the lowest compared with the Merkle tree varieties.

2.3 Version storage

A ledger database is a multi-version storage where every commitment produces a new version V. Ledger database should allow clients to operate on the data according to their versions, including version pruning that deletes the versions older than a specified version V and version rollback that deletes versions newer than a specified version V. For both pruning and rollback, the ledger database has to find records of the versions to be deleted and reclaim their storage space for reuse.

Version rollback. Version rollback restores the data to a prior version V, by removing all the data of versions newer than V. When multiple databases detect inconsistencies, they must efficiently roll back to a consistent version. Version rollback depends on garbage collection to identify data to be removed.

Version pruning. Version pruning removes data that is older than a specific version to save storage space [25]. Version pruning is triggered frequently when the storage space is constrained, but the database receives a large number of versions that can not be known upfront [4]. The version pruning is done with a garbage collection process that traverses all the pages to identify those storing obsolete versions [31, 43].

Discussion: Costs of version operations. Both version rollback and version pruning require garbage collection to free the storage space for new data records. However, existing garbage collection methods are I/O-intensive. For example, LETUS [31], a state-of-the-art ledger database, performs disk scans to identify pages for deletion and rewrites all valid data into new files during garbage collection. As a result, LETUS takes hours to process a version pruning, which fails to free space in time for high-frequency data ingestion when the disk space is constrained. Therefore, it is critical to eliminate intensive disk I/O for version operations.

2.4 Limitation in current solutions

Existing ledger databases can handle large-scale data deployments but face significant cost-efficiency challenges in cloud environments, particularly regarding storage consumption, disk I/O, and computational overhead.

Lack of support for frequent version pruning. Ledger databases maintain multiple data versions to ensure immutability, leading to prohibitive storage overhead. While version pruning removes data older than a specified version [31, 39], existing systems are not optimized for frequent pruning operations required in storage-constrained environments.

Storage overhead from index node identifiers. Ledger databases retrieve verifiable index nodes using node identifiers stored alongside node content [31]. This co-location of identifiers and content

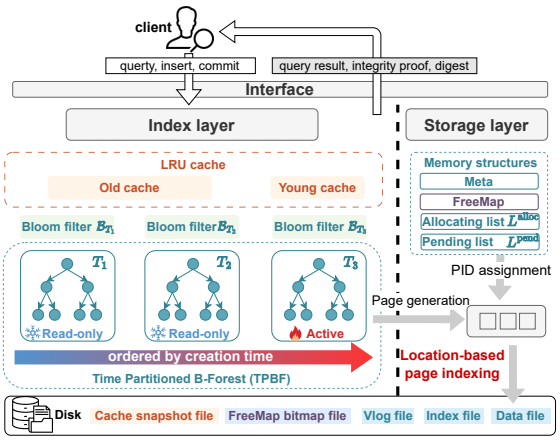


Figure 2: VIDB overview.

enables node lookup but significantly amplifies disk space consumption, as each node requires additional storage for its identifier metadata.

Write amplification from background compaction. Verifiable index storage introduces substantial write amplification. Previous ledger databases employ LSM-based storage where node content serves as values and content-derived hashes serve as keys [4, 39]. The inherent randomness of these hash-based keys requires continuous background compaction to maintain key ordering, consuming significant I/O bandwidth.

Intensive disk I/O from garbage collection. Garbage collection for reclaiming storage from removed versions involves three I/O-intensive steps [30]: (1) scanning all pages to identify those belonging to the target version, (2) updating pages that reference the removed version, and (3) recording freed pages scattered across storage for reuse. All steps require substantial disk operations in existing designs.

Escalating complexity from growing tree height. Data operations in ledger databases require traversing the Merkle tree from root to leaf, making insertion, point queries, and range queries dependent on tree height. As shown in § 2.2, tree height—and thus operational complexity—grows logarithmically with data volume in existing Merkle tree structures.

High latency for historical version queries. Existing systems [4, 39] store record versions as linked lists with the latest version as the head. Historical version retrieval requires traversing this chain from head to target version, resulting in query latency proportional to version age.

3 System Design

VIDB is a ledger database aiming at achieving cost efficiency in storage space, disk I/O, and computational cost, with data scaling.

3.1 Overview

VIDB employs a two-tier architecture comprising an index layer and a storage layer, as illustrated in Figure 2. The system implements a key-value data model augmented with Merkle proofs to ensure data integrity and support authenticated queries. The index layer

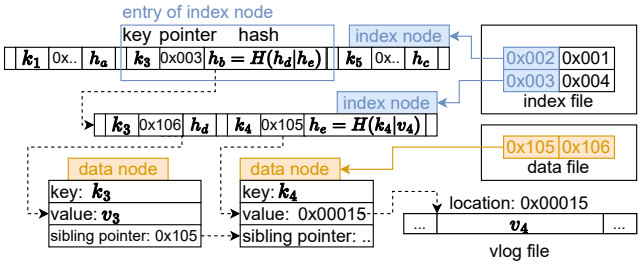


Figure 3: Tree structure.

Table 3: API interface.

category	No.	API	category	No.	API
Key-value query	F1	Get(k)	verification	F6	Proof(V, k)
	F2	RangeGet(k_{lb}, k_{ub})	version management	F7	Commit(V)
	F3	HistGet(V,k)		F8	Rollback(V)
	F4	Put(k, v)		F9	Prune(V)
	F5	Delete(k)			

functions as the index and the ADS for the key-value data records, supporting efficient data retrieval and proof generation. The storage layer manages the storage space, providing page abstraction for the index layer.

The system exposes a series of APIs (detailed in Table 3) that support the key-value query, verification, and version control. Key-value APIs encompass point queries (F1), range query with specified bounds (F2), historical queries for specific versions (F3), record insertions (F4), and deletions (F5). The verification APIs provide cryptographic proof generation for any version and key combination (F6) Version management APIs support transaction commits that create new data versions (F7), rollbacks to previous states (F8), and pruning of outdated versions (F9) for storage optimization.

We use the client in Figure 1 as an example. To pose a query, the client calls Get(k_1) for a single point query, receive the query result $\langle k_1, v_1 \rangle$ for VIDB For data verification, the client can use Proof(V, k) to request a proof π_1 from VIDB. For updating data, the client calls Put(k_2, v_3) to update the data. The client calls Commit(V) to commit the updates as a new version $V + 1$. VIDB create the new version $V + 1$ and return the digest $\delta_{D_{V+1}}$ to the client. The client can use the digest and proof to check the tamper evidence of the storage and the query process, respectively.

3.2 Index layer

The index layer serves as the core component responsible for both data indexing and integrity proof generation for query results. This dual functionality ensures that VIDB can provide verifiable query responses while maintaining efficient data access patterns. As discussed in § 1, the design of the index layer prioritizes the optimization across computational cost and disk I/O. Rather than employing conventional monolithic indexing approaches, VIDB introduces a novel temporal partitioning strategy that addresses the issue of unbounded tree height growth as data volume increases.

Temporal Partitioned B$^+$-tree Forest (TPBF). The TPBF structure implements a temporal-partitioned indexing mechanism defined as a forest $F = \{T_1, T_2, ..., T_n\}$, where each tree $T_i \in F$ functions as a Merkle tree responsible for indexing a distinct record subset $C_{T_i} \subset D$ from the complete dataset D. This partitioning strategy

enforces two fundamental properties that ensure the scalability: (1) *Disjointness* guarantees that all subsets remain mutually exclusive, formally expressed as $C_{T_i} \cap C_{T_j} = \emptyset$ for $i \neq j$, thereby eliminating data duplication across trees. (2) *Bounded capacity* limits each subset size to a configurable threshold, ensuring $|C_{T_i}| \leq N_{\text{cap}}$ for all $T_i \in F$, where $N_{\text{cap}} \in \mathbb{N}$ represents the maximum capacity parameter. Database administrators can configure the maximum tree height $h_{\max}$ using the relationship $h_{\max} = \log_m N_{\text{cap}}$.

The forest-based design fundamentally differs from traditional approaches that maintain a single monolithic Merkle tree spanning the entire dataset. TPBF achieves improved performance through temporal partitioning, where each tree T_i independently manages query processing, update operations, and verification tasks for its assigned record subset C_i. The system maintains exactly one active tree capable of receiving updates while all remaining trees operate in read-only mode, as demonstrated in Figure 2. When the active tree reaches the capacity threshold N_{cap}, it transitions to read-only status, and TPBF instantiates a new active tree to handle subsequent updates. This approach effectively decouples write operations from historical data, enabling concurrent read access to frozen trees while maintaining write performance through the dedicated active tree. Note that the TPBF implementation employs Copy-on-Write semantics for multi-version support, ensuring node immutability and generating new nodes for all update operations to maintain consistency across different data versions.

Figure 3 demonstrates the structure of a Merkle B$^+$-tree in TPBF whose fanout is m. The tree consists of two node types serving distinct functions in indexing and verification. Index nodes contain m entries, each with three fields: a pointer field referencing child nodes through page identifiers (PID), a key field storing the minimum key value of the child subtree, and a hash field containing the cryptographic digest $\mathcal{H}(h_1|...|h_m)$ computed from child node hash values. The root node includes an additional digital signature for verification. Data nodes store actual key-value records with three fields: the key field for record identification, the value field for associated data (or positional references to a separate value log for large values), and sibling pointers linking nodes in key-sorted order for efficient range queries.

Discussion. The TPBF design provides several key advantages for authenticated data management. First, the temporal partitioning approach ensures bounded computational complexity by maintaining stable tree heights as data volume increases, unlike monolithic tree structures, where height grows with dataset size. Second, restricting updates to a single active tree minimizes the number of modified nodes during write operations, thereby reducing write amplification overhead as analyzed in §4.4. Finally, the temporal ordering inherent in TPBF enables efficient tiered storage management, as older trees created earlier and accessed less frequently can be readily identified and migrated to lower-cost storage tiers without requiring additional metadata or scanning operations.

Auxiliary structures. In addition to TPBF, ViDB incorporates auxiliary structures to enhance query performance. In particular, LRU caches accelerate queries for frequently accessed keys, while Bloom filters reduce the overhead for queries with empty results.

To exploit query locality, ViDB maintains two least recently used (LRU) caches: a young cache for the active tree and an old cache for all the read-only trees. An LRU cache stores recently accessed records in memory, enabling fast retrieval without disk I/O. When a queried record is retrieved from the active tree, it is added to the young cache; otherwise, it is added to the old cache. The two caches do not contain records with the same key. If a record in the old cache is removed if it is later accessed from the active tree.

The system employs Bloom filters to optimize queries for non-existent keys, which would otherwise require traversing all trees. Each tree T is equipped with a Bloom filter $\mathcal{B}_T$ summarizing its key membership. Before traversing T for key k, the system queries $\mathcal{B}_T$. If $\mathcal{B}_T(k)$ returns negative, the traversal is bypassed since the Bloom filter guarantees $k \notin C_T$.

During query processing, ViDB utilizes LRU caches and Bloom filters before traversing trees in TPBF to accelerate the execution. The details of the processing are explained in later sections § 4.1.

3.3 Storage layer

The storage layer provides compact node storage for both data records and index nodes while enabling efficient page retrieval on disk. Our Location-based versioned page storage (LVPS) design controls write and space amplification, supports efficient version queries and frequent version pruning, and eliminates extensive disk I/O during garbage collection through a novel location-based page indexing scheme.

ViDB employs a page-oriented storage architecture where each node (index or data) resides in a fixed-size page (4KB). The system assigns Page IDs (PIDs) based on physical disk locations, enabling direct page access without storage space searches, as shown in Figure 3. This location-based approach eliminates the need to store PIDs explicitly, reducing storage overhead. Figure 2 shows that LVPS supports PID assignment for the pages by the index layer. When the index layer requests a page via PID, LVPS directly locates it using the physical address, while new pages receive PIDs before being asynchronously flushed to their designated locations.

Since PIDs represent physical locations, LVPS must determine them before disk flushing to prevent index layer stalls during tree construction. The system employs two key memory structures, as illustrated in Figure 2: The *Meta table* maintains version-specific entries containing root PID, max index PID, and max data PID for each version, while the *FreeMap* organizes freed pages into continuous page spans. Each FreeMap entry maps span sizes to lists of starting PIDs, enabling efficient free page location without disk overhead.

For new page allocation, LVPS first queries the FreeMap for available free pages, falling back to incrementing the maximum PID from the Meta table when no free pages exist. The system maintains two auxiliary lists to support garbage collection: an allocating list L^{alloc} tracking pages allocated from FreeMap during version rollbacks, and a pending list L^{pend} recording pages updated in subsequent versions that become eligible for pruning. These structures enable efficient garbage collection without excessive disk access, where details can be referred to § 4.3. In the implementation, ViDB persists the FreeMap as a space-efficient bitmap file, while storing the Meta table, allocating list, and pending list within the index file.

The location-based page indexing approach provides three distinct advantages over existing ledger storage solutions. First, the

design achieves constant write amplification by writing each page exactly once to its designated location, contrasting with LSM-based storage systems [1, 4, 33] that perform multiple writes during layered compaction processes. Second, the approach significantly reduces update complexity compared to content-based indexing schemes [4, 31] that derive PIDs from page content such as hash values or version numbers. While content-based systems require expensive storage space searching, and page sorting operations to locate requested pages, location-based indexing enables direct page access through physical addressing, eliminating both search overhead and sorting requirements. At last, the design reduces space amplification by eliminating the need to store explicit PID mappings, whereas content-based indexing incurs storage overhead by maintaining PID-to-location mappings to enable page searches. These combined advantages position LVPS as a fundamentally more efficient approach to versioned page storage in large-scale ledger databases.

4 Query Processing in VIDB

In this section, we elaborate on how VIDB utilizes TPBF and LVPS to achieve verifiable query processing, data versioning, and garbage collection with cost efficiency. We close this section with a theoretical analysis of data operation complexity and write amplification.

4.1 Verifiable query processing

We elaborate on how VIDB processes point queries and range queries verifiably.

4.1.1 ***Point query.*** Given a key k, a point query retrieves the record $\langle k, v\rangle$ along with its integrity proof. VIDB first consults the cache and returns the cached record and proof if available. When a cache miss occurs, VIDB queries the Bloom filters to identify candidate trees that may contain the target record, forming the set $\mathcal{T} = \{T | \mathcal{B}_T(k) \text{ is positive}\}$. The system then traverses all trees in $\mathcal{T}$ concurrently to locate the target record, collecting hashes from sibling nodes along each traversal path to construct the integrity proof. If the target record exists in multiple trees, VIDB selects the result from the tree created later to ensure freshness. Finally, the query result and its integrity proof are inserted into the young cache for future access if it is from the active tree; otherwise, it is cached in the old cache.

*Example 4.1 (**Point query**).* Figure 4 illustrates a point query for key k_5, where the processing flow is marked green. The processing starts from cache lookup(①), and then queries the bloom filter(②). Assuming k_5 is not cached but exists, VIDB concurrently searches both T_2 and T_3(③). During the traversal, nodes are directly located on disk based on the location-based page indexing(④). In T_2, the system traverses nodes 1 and 3 to locate v_{5,T_2} at node 9. Simultaneously, in T_3, the system visits nodes 5, 7, and 10 to retrieve v_{5,T_3} from the value log file. Since T_3 is created later than T_2, it contains the fresh data. VIDB returns $\langle k_5, v_{5,T_3}\rangle$ as the final result along with the corresponding integrity proof. The query result and proof are then cached in the young cache for subsequent queries.

4.1.2 ***Range query.*** VIDB traverses each tree T to find a lower bound $k_{\text{lb},T}$ and an upper bound $k_{\text{ub},T}$ satisfying $k_{\text{lb},T} \geq k_{\text{lb}}$ and $k_{\text{ub},T} \leq k_{\text{lb}}$. Similar to point query processing, trees are concurrently traversed, and the integrity proofs are collected during the traversal. The records between $k_{\text{lb},T}$ and $k_{\text{ub},T}$ are retrieved as a result set based on the pointer among data nodes. After traversal, non-empty result sets are returned with the integrity proof. If a key appears in multiple result sets, VIDB uses the one from the more recently created tree.

*Example 4.2 (**Range query**).* Figure 4 describes a range query invoked by RangeGet(k_2, k_9), and the processing flow is marked blue. VIDB traverses the trees concurrently, and finds that the bounds on T_2 are k_3 and k_9 while the bounds on T_3 are k_2 and k_7. Note that k_2 is not found on T_1, and k_3 is the upper bound instead. Similarly, k_9 is not found on T_2, so k_7 is the lower bound. After the traversal, two result sets is retrieved: $\{k_1, k_3, k_9\}$ from T_2 and $\{k_2, k_3, k_4, k_5, k_6, k_7\}$ from T_3. Although k_5, k_6 appear in both trees, the results from the more recently created tree, i.e., T_3, are used.

4.2 Versioning query processing

In this section, we describe how VIDB updates existing data or inserts new data while maintaining multiple data versions, and how VIDB process query that designates a specific version.

4.2.1 ***Update or insert data.*** VIDB updates or inserts data only on the active tree. When the client invokes the function Put(k, v) and immediately commits, VIDB first traverses the active tree with k until it reaches the last layer of index nodes. During the traversal, all the visited nodes are pushed into a first-in-last-out (FILO) stack. VIDB update the data node containing k if it exists, otherwise create a data node to store record $\langle k, v\rangle$. Then VIDB recursively updates all the ancestors of the data node containing k, which are buffered in the FILO stack. When the client invokes Delete(k), VIDB inserts a record $\langle k, \text{null}\rangle$ as a tombstone to indicate the record is deleted. If the client commits a batch of put operations, VIDB updates the nodes in the active tree in a depth-first manner, based on the FILO stack. It ensures that each node will be updated only once for a batch of put operations. Note that data versioning is achieved by copy-on-write, so VIDB copies the original node and applies the updates on the copied node, without modifying the original nodes. After the client's commitment, VIDB creates a new version, and updates Meta, L^{alloc}, L^{pend}, and FreeMap according to the generated pages.

*Example 4.3 (**Insertion**).* Figure 4 provides an example to show how VIDB inserts a record $\langle k_8, v_8\rangle$. New nodes created during the insertion are distinguished by a dashed border. The processing flow is marked red. VIDB traverses the active tree T_3 and pushes node 5 and 8 into the FILO stack(①). VIDB creates a data node 11 to store k_8 and the location of v_8 in the vlog file. VIDB then pops the ancestors of node 11, i.e., node 8 and 5, from the FILO stack, copies them into node 8' and 5', and applies the updates on the copied nodes.

The insertion requires three pages for node 11, 8', and 5'(②). LVPS assigns PID 0x002 to node 8' based on the FreeMap, and assigns PID 0x009 and 0x103 to node 5' and 11 using the Meta table (③). The pages are asynchronously flushed to the disk based on the PIDs (④). All the memory structures update accordingly. Page 0x002 is removed from the FreeMap and added into L^{alloc}. Meta

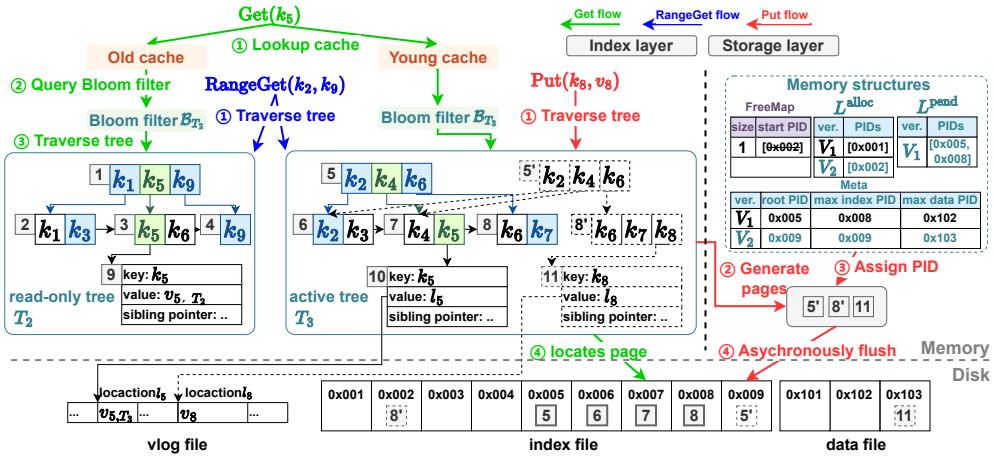


Figure 4: Running example of ViDB.

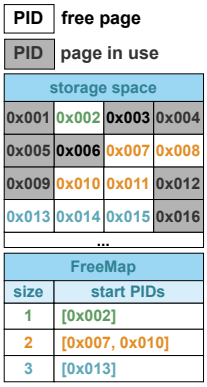


Figure 5: An example for FreeMap.

create an entry for the new version to record page 0x009 and 0x103 are the new max index/data PID, and 0x009 is also the root of the new version. L^{pend} records page 0x005 and 0x008 become pending pages since node 5 and 8 are updated.

4.2.2 ***Historical version query.*** A historical version query request invoked by HistGet(V, k) retrieves the record with a key k and a specific version V. The processing of a version query is similar to the point query, but the traversal starts from the root of the requested version V. Since the Meta structure records the root PID of all the versions, ViDB retrieves any given version with the same cost. For comparison, previous works maintain multiple versions as a linked list, and the I/O path is longer to retrieve an older version.

4.3 Garbage collection with version operations

We explain how ViDB processes garbage collection during version pruning and rollback without intensive I/O utilizing the LVPS. For version pruning, we consider ViDB prunes all versions before V_{prune}. For version rollback, we consider ViDB rollback the data from the current latest version V_{roll}. Both operations will remove data and require garbage collection to reclaim the disk space. The garbage collection can be divided into three steps.

Identify the free pages. Garbage collection first identifies the free pages whose content is removed after version pruning or rollback. Previous work [43] scans all the pages to find free pages, requiring intensive disk I/O. ViDB relies on the allocating list L^{alloc} and pending list L^{pend} to find free pages without accessing the disk.

During version pruning, ViDB identifies all the pages recorded in L^{pend} associated with the version before V_{prune} as free pages. During version rollback, ViDB identifies pages recorded in L^{alloc} associated with the version newer than V_{roll} as free pages. In both cases, ViDB only reads from memory structures and does not need to scan the pages as previous works.

Remove reference to the free pages. In conventional systems, garbage collection updates all pages referencing freed pages to maintain pointer consistency and avoid dangling references. ViDB avoids this step, as each page is only referenced by its parent. Moreover, in ViDB, parent nodes are always from newer versions than their children. As a result, when a page becomes free during version pruning or rollback, its parent is also guaranteed to be free.

Track the free pages. At last, garbage collection has to record all the free pages for future reuse. The number of free pages can be large, and it is prohibitive to record each one. ViDB uses the FreeMap structure to record free pages, which compactly organizes a large number of free pages into spans. Figure 5 shows a FreeMap maintaining five page spans. For example, the second entry records two spans, each of size 2. The first span starts from page 0x007, so it indicates pages 0x007 and 0x008.

We provide two examples to illustrate how garbage collection works for version pruning and rollback.

*Example 4.4 (**Version pruning**).* Based on Figure 4, we describe how ViDB prunes the version V_1. ViDB first identifies free pages from the entry of V_1 in the pending list L^{pend}, i.e. page 0x005 and 0x008, and insert them into the FreeMap. Then, ViDB remove all the entries associated with V_1 in the memory structure, that is, the entry V_1 in Meta, L^{alloc}, and L^{pend}.

*Example 4.5 (**Version rollback**).* Based on Figure 4, we describe how ViDB rolls back to version V_1. Since only V_2 is newer than V_1, the rollback process is to remove V_2. ViDB first identifies free pages from the entry of V_2 in the allocating list L^{alloc}, i.e. page 0x002. ViDB insert page 0x002 into the FreeMap. At last, ViDB deletes the entries in Meta, L^{alloc}, and L^{pend} associated with V_2.

Discussion. Compared with previous works, ViDB significantly reduces the disk I/O for version operations. For version pruning, previous work, such as QMDB [43], LETUS [31] require scanning the disk to identify pages of the pruned versioned. For rollback, Ethereum [33] traverses the tree to find pages to be removed, which is I/O intensive. ViDB uses L^{alloc} and L^{pend} to identify the free pages without accessing the disk. To reuse the free pages, previous solutions [31] merge the free pages by reorganizing all the valid data into new files, consuming significant disk I/O. With FreeMap, ViDB efficiently tracks all the FreeMap and reuses them without intensive data reorganization. Therefore, ViDB can achieve efficient garbage collection for version pruning, avoiding intensive disk I/O.

4.4 Analysis

We quantify the query complexity and write amplification of ViDB.

4.4.1 ***Operation complexity.*** We discuss the complexity of point queries, range queries, and inserting data. Table 2 summarizes the operation complexities of ViDB. Compared to other Merkle trees listed in Table 2, ViDB removes the number of data records N_{rec} and the length of key $|k|$ from the complexities. The complexities are mainly determined by the configurable N_{cap}, i.e., the maximum tree capacity. It means that ViDB can bound the computation cost, and prevent it from growing with data volume.

The processing of a point query includes LRU cache lookup, bloom filter query, and tree traversal. The complexities of the first two steps are constant. Since ViDB traverses the trees concurrently, the time of tree traversal is the time to traverse one tree. Besides, thanks to the bloom filter, ViDB does not traverse all the trees most of the time. With sufficient parallelism, the cost of a query is determined by the tree height $\log_m N_{\text{cap}}$, where m is the fanout. Similar to the point query, the tree traversal cost of the range query is determined by the tree height. Note that a range query requires additional N_R steps to retrieve records between the upper and lower bounds, where N_R is the number of records in the query range. The complexity of updating or inserting a record is determined by the number of nodes updated in the active tree. Since all the ancestor nodes are updated, the complexity of updating one record is also bounded by the tree height.

4.4.2 ***Proof size.*** The proof size is critical for the efficiency of data verification. ViDB needs to transfer the proof to the client for data verification, occupying the network bandwidth. Therefore, a small proof size can reduce the network bandwidth requirement. The proof sizes in ViDB are summarized in Table 2. We can observe that the proof sizes also do not grow with the data volume.

For a point query, only the result of one tree is returned, so only the integrity proof collected from one tree is used, whose size is bounded by the tree height. For a range query, the number of non-empty result sets $N_{RS} \in [1, N_R]$, where N_R is the number of records in the query range. For each result set, the proof is the sibling's hash of the path to the upper and lower boundary, and the path length is the tree height.

4.4.3 ***Write amplification.*** The write amplification A_{write} is the physical writes on the backend storage for each record update. The write amplification of a ledger database A_{write} can be decomposed into the amplification from the index layer A_{index} and the amplification from the storage layer A_{storage}, and $A_{\text{write}} = A_{\text{index}} \times A_{\text{storage}}$.

A_{index} counts the number of Merkle tree nodes ViDB generates for updating a record. Thanks to the design of TPBF, ViDB only generates nodes on the active tree. A_{index} counts the nodes on the path from the root to the data node, which is equal to the tree height. Thus, $A_{\text{index}} = \log_m N_{\text{cap}}$. Note that it is the worst case discussed above. For the best case, a batch of N_B new records share all the nodes, and the amplification is amortized, $A_{\text{index}} = \frac{1}{N_B} \log_m N_{\text{cap}}$.

A_{storage} is the backend storage write for a Merkle tree node. With location-based page indexing, ViDB writes a page to disk only once, without any further compaction. Thus, $A_{\text{storage}} = |\text{Page}|$, where $|\text{Page}|$ is the page size.

In conclusion, the overall write amplification of ViDB is calculated as Eq. 1 for the worst case, and Eq. 2 for the best case.

$$A_{\text{write}}^{\text{ViDB-w}} = O(\log_m N_{\text{cap}}) \times |\text{Page}| \tag{1}$$

$$A_{\text{write}}^{\text{ViDB-b}} = O(\log_m N_{\text{cap}}/N_B) \times |\text{Page}| \tag{2}$$

For comparison, we also calculate the write amplification of Ethereum's MPT+LevelDB architecture (ETH) [33] and the state-of-the-art LETUS engine (LETUS) [31] in Eq. 3 and Eq. 4, respectively.

$$A_{\text{write}}^{\text{ETH}} = O(N_{\text{rec}} \log_m N_{\text{rec}}) \times |\text{Page}| \tag{3}$$

$$A_{\text{write}}^{\text{LETUS}} = O(\log_m N_{\text{rec}}/T_b) \times |\text{Page}| \tag{4}$$

Note that both write amplifications grow with the number of data records N_{rec}. The write amplification of ViDB is lower since $N_{\text{rec}} >> N_{\text{cap}}$. Although LETUS can reduce the amplification with a configurable parameter T_b, this value is usually small (e.g., 2) in practice.

5 Evaluation

In this section, we conduct several experiments to analyze ViDB by comparing it with existing ledger database designs regarding micro-benchmarks and real-world system evaluation.

5.1 Setup

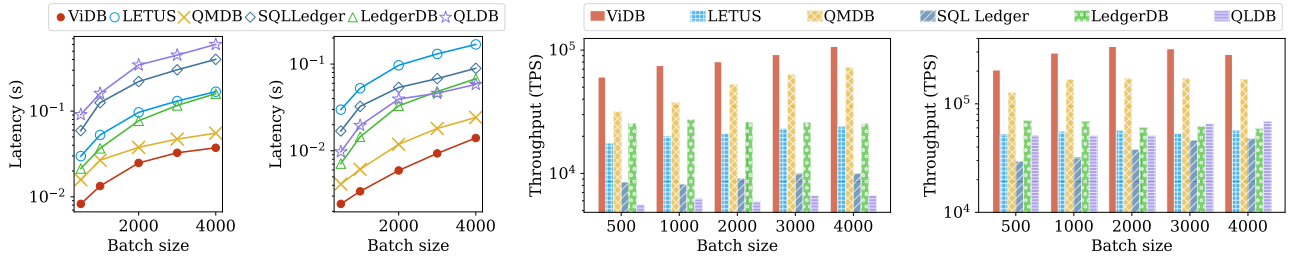
Environment. All experiments were conducted on a server with Ubuntu 22.04 LTS, equipped with an 18-core Intel(R) Xeon(R) Gold 6240C 2.60GHz CPU, 128 GB RAM, and 1.5TB SSD disk. We evaluate the ledger databases on a single-node setup without sharding.

Baselines. We compare ViDB against five state-of-the-art ledger databases, namely QLDB [1], LedgerDB [39], SQL Ledger [4], QMDB [43], and LETUS [31]. We use the open-source implementation[1] of QLDB, LedgerDB, SQL Ledger, and QMDB. LETUS is not open-sourced, and we implement it ourselves based on its original paper [31]. In addition, we integrate ViDB and LETUS into a commercial system called Hyperchain [15] to evaluate the performance under real blockchain workloads.

Workloads. We generate workloads to evaluate the performance of data insertion, point queries, range queries, historical queries, and garbage collection. For data insertion and point queries, we use a *micro-benchmark* based on YCSB, and set the Zipf factor to 0 for a balanced access distribution. For range queries and historical queries, we generate query workloads under different data volumes. Version pruning and rollback workloads are used to evaluate garbage collection performance. Each workload operates on preloaded key-value datasets generated using sequential keys paired with random string values. For real-world system evaluation, we leverage Hyperchain to evaluate blockchain-specific performance through two transaction types: (1) *Simple payment* transactions randomly select sender-receiver pairs, transferring funds when sender balance permits; (2) *EVM transfer* transactions invoke the transfer function of a deployed ERC-20 smart contract with randomly selected accounts. The test environment initializes 30K accounts with 3% active transaction participation.

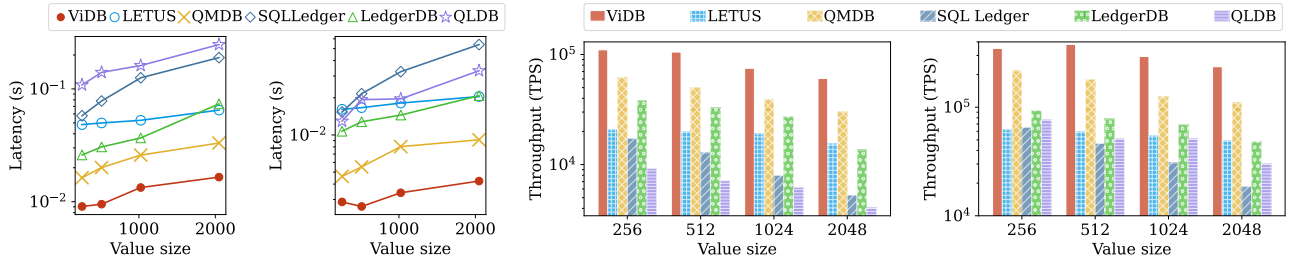
Metrics. We evaluate system performance using three key metrics: throughput (transactions per second), latency (response time in seconds), and disk usage (storage consumption in kilobytes).

[1] https://anonymous.4open.science/r/ViDB-Cost-Efficient-Ledger-Database-At-Scale-FD73/README.md



(a) Put latency vs. batch size (b) Get latency vs. batch size (c) Put throughput vs. batch size (d) Get throughput vs. batch size

Figure 6: Micro-benchmark results (vs. batch size).



(a) Put latency vs. value size (b) Get latency vs. value size (c) Put throughput vs. value size (d) Get throughput vs. value size

Figure 7: Micro-benchmark results (vs. value size).

These metrics collectively assess computational efficiency, execution speed, and storage cost-effectiveness.

5.2 Micro-benchmark

5.2.1 Point query. We generate batches of operations using the system's get and put APIs to measure transaction processing latency upon preloaded 1 million data records for each ledger database. Each put API call randomly selects a key and updates its associated value, corresponding to an insertion operation in the ledger. Each get API call retrieves the value for a randomly chosen key, corresponding to a point query operation. To systematically analyze performance characteristics, we fix the key size at 32 bytes and vary both value size and batch size parameters, which is similar to the micro-benchmark in LETUS [31].

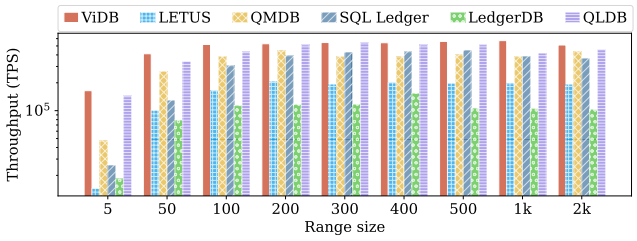
Figure 6 presents results for get/put workloads with batch sizes ranging from 500 to 4000 operations, while maintaining a fixed value size of 1024 bytes. Complementary results in Figure 7 depict performance across value sizes from 256 bytes to 2048 bytes with a fixed batch size of 1000 operations.

The experimental results demonstrate the superior cost efficiency of ViDB. For put workloads, ViDB achieves throughput improvements of 4.5×, 2.0×, 12.4×, 4.4×, and 28.1× compared to LETUS, QMDB, SQL Ledger, LedgerDB, and QLDB, respectively. This performance advance stems from LVPS's elimination of background compaction, which significantly reduces write amplification. Additionally, the bounded tree height property of TPBF substantially reduces insertion computational overhead, enabling ViDB to maintain the lowest put latency across all tested conditions (Figure 6(a) and Figure 7(a)). Point queries in get workloads similarly benefit from the bounded tree height design. ViDB achieves query throughput improvements of up to 6.0×, 2.0×, 10.6×, 6.2×, and 428.4× over LETUS, QMDB, SQL Ledger, LedgerDB, and QLDB, respectively, while maintaining the lowest latencies (Figure 6(b) and Figure 7(b)).

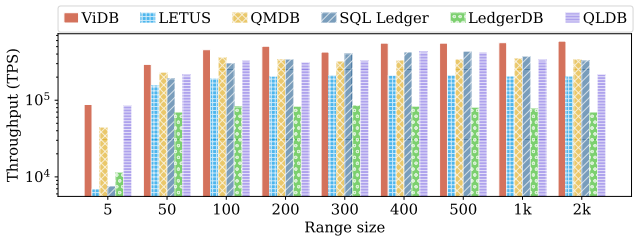
To understand the performance differences among ledger databases, we analyze the detailed results using batch size 4000 from Figure 6(d). The performance variations can be attributed to fundamental architectural differences in how each system manages Merkle-based indexes when processing put operations. QLDB, SQL Ledger, and QMDB all maintain separate data index and Merkle tree as individual structures, requiring updates to both components for each put operation. QLDB exhibits the lowest throughput due to its maintenance of dual B-tree indexes for current and historical versions, creating complex update procedures. SQL Ledger achieves higher throughput than QLDB by performing asynchronous Merkle tree updates. QMDB also updates both structures but truncates keys to nine bytes for the B-tree, significantly reducing update complexity. However, this optimization introduces potential key collision risks that can cause system crashes [43].

LedgerDB outperforms QLDB and SQL Ledger by eliminating the B-tree index requirement, but its performance remains limited by the need to update skip lists for historical data. LETUS reduces update complexity by unifying the index and Merkle tree into a single verifiable index, but suffers from performance degradation due to excessive tree height. In contrast, ViDB achieves the highest performance despite utilizing B-tree structures to enhance get performance. This results from two aforementioned key innovations : (1) unifying the B-tree and Merkle tree structures, and (2) reducing update complexity through time-based tree partitioning (§ 3.2).

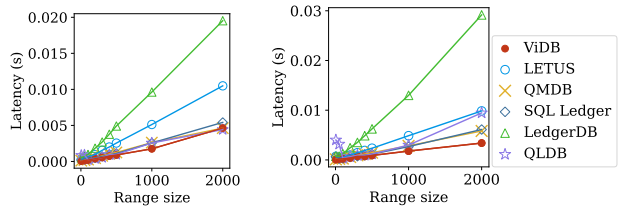
The batch size analysis in Figure 6(c) reveals that ViDB's put performance scales positively with batch size, while LedgerDB,



(a) Throughput (1M records)



(b) Throughput (10M records)



(c) Latency (1M records) (d) Latency (10M records)

Figure 8: Range query performance.

SQL Ledger, QLDB, and LETUS show diminishing or flat performance trends. QLDB's synchronous persistence prevents batched updates from amortizing I/O costs. Although LETUS, SQL Ledger, and LedgerDB employ asynchronous persistence, their performance improvement is limited because the number of nodes requiring updates increases proportionally with batch size. VIDB's scalability advantage derives from its compact active tree design. Larger batch sizes increase the probability of sharing node updates across multiple put operations, leading to improved throughput efficiency.

The value size analysis reveals an inverse relationship between value size and throughput for both get and put operations (Figure 7(d) and Figure 7(c)). This performance degradation occurs because larger values require more time to read from disk, given limited I/O bandwidth constraints. The same I/O bottleneck affects put operations, as they must write larger values to storage. Across all tested value sizes, VIDB consistently achieves the highest throughput, demonstrating the effectiveness of its storage layer's constant write amplification optimization.

5.2.2 Range query. Beyond single-point operations, we evaluate the performance of ledger databases on range queries with varying selectivity. We preload datasets of 1 million and 10 million records into each tested system, then execute range queries with result set sizes ranging from 5 to 2,000 records. Each data point represents the average of 100 query executions to ensure statistical significance. We measure both throughput and latency, as shown in Figures 8(a)–8(d). All experiments use 32-byte keys and 1024-byte values to keep consistency with our point query evaluations.

As can be observed in Figure 6(b) and Figure 7(d)), the presence of B-tree indices in QLDB, SQL Ledger, and QMDB enables efficient range query processing, resulting in substantially higher throughput compared to their random get performance. In contrast, LedgerDB lacks indexing support, causing its range query performance to degrade to that of random gets. LETUS employs a verifiable DMM-Trie index that provides moderate range query acceleration, though performance remains limited by the large tree height. VIDB leverages TPBF with range query efficiency from the B^+-tree structure, achieving the highest range query throughput across all experimental conditions while maintaining cryptographic verifiability guarantees.

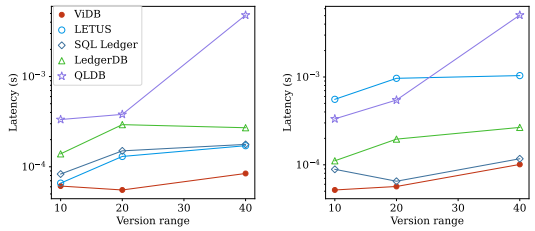
To assess scalability, we evaluate range query performance on larger datasets by increasing the record count from 1M to 10M. As demonstrated in Figure 8(b), VIDB's performance advantage is preserved and even amplified with increased data volume. The system maintains consistently low latency while achieving higher throughput as range sizes increase, demonstrating effective scaling properties of our indexing approach.

The latency analysis in Figure 8(c) and Figure 8(d) confirms VIDB's superior performance, exhibiting the lowest query latency among all evaluated ledger databases. Importantly, VIDB demonstrates small latency growth rates with respect to range size, ensuring scalable performance even for large-range queries.

5.2.3 Historical Query. This evaluation examines the performance of ledger databases when querying historical data versions, a critical capability for audit and compliance scenarios. We exclude QMDB from this analysis as it does not provide historical version query functionality, focusing our comparison on systems that support multiple version access.

We preload datasets containing 1M and 10M records into each system, then randomly select keys from the loaded dataset for historical queries. For each selected key, we query versions ranging from 10 to 40 versions prior to the current state, simulating realistic audit scenarios where users need to access recent historical data. Each measurement represents the average latency over 100 query executions to ensure statistical reliability.

Figure 9(a) demonstrates that VIDB achieves superior performance for historical version queries across all tested version depths. While baseline systems (LETUS and SQL Ledger) exhibit comparable latency for recent historical versions (e.g., 10th version prior), their performance degrades significantly when accessing older versions. This degradation stems from their architectural design: SQL Ledger, LedgerDB, and QLDB employ backward traversal strategies that require long sequential navigation from the current version to the target historical state. In contrast, VIDB leverages its Meta table (detailed in § 3.3) to provide direct access to any historical version through efficient metadata lookups, achieving nearly constant $O(1)$ query latency regardless of version age. This architectural advantage becomes increasingly pronounced as the version distance increases, with VIDB maintaining sub-millisecond latency even for versions 40 steps removed from the current state.



(a) 1M records (b) 10M records

Figure 9: Query historical version.

5.2.4 Storage cost. To show ViDB's effectiveness in reducing storage space consumption, we measure the disk usage of the ledger database with data volumes ranging from 10 thousand to 10 million. In this evaluation, we use 32-byte keys and 256-byte values.

Compared to the baselines, ViDB substantially reduces disk usage as Figure 10 shows. When the data volume is relatively small, ledger databases with RocksDB backend, including QLDB, LedgerDB, and SQL Ledger, have a constant storage overhead (e.g., 10K and 100K records in Figure 10). ViDB, LETUS, and QMDB manage the page storage themselves, so the disk usage is lower than those utilizing RocksDB. Among them, ViDB achieves the lowest storage consumption thanks to the elimination of storage for identifiers.

For large data volumes, ViDB still maintains the efficiency in disk usage (e.g., 1M and 10M records in Figure 10). SQL Ledger, LedgerDB, and QLDB keep a relatively low storage cost for a large data volume, using the compaction of RocksDB. However, the compaction sacrifices the get and put performance (in § 5.2), since it introduces write amplification and overhead for query processing. ViDB achieves the same level and even lower disk usage without the cost of compaction, thanks to the compact storage of LVPS.

Storage tiering is a common technique for saving storage cost [31]. We evaluate ViDB under a two-tier storage configuration consisting of a hot tier and a cold tier, where the cold tier is low-cost and low-performance, suited for infrequently accessed records. We measure the query latency on both tiers while ViDB migrates data to the cold tier. As shown in Figure 11, query latency on the hot tier remains stable during the migration. This trend holds as the dataset grows from 1 million to 10 million records, indicating that cold data migration does not affect the query processing on the hot tier. This behavior, as discussed in § 3.2, is enabled by the temporal partitioned structure of TPBF, which efficiently identifies cold data based on tree creation time, without intensive disk I/O.

5.2.5 Version operations. This section evaluates ViDB's performance on critical version management operations: version pruning and version rollback. These operations are essential for maintaining system efficiency and supporting operational requirements in production ledger databases.

Version Pruning. We assess the impact of version pruning on system performance by monitoring update latency as version history accumulates. Our experimental setup uses datasets ranging from 500 to 4,000 records, with each record updated up to 100 times to create a substantial version history. We configure the system to trigger version pruning automatically when the version count reaches 40, then maintain this threshold throughout subsequent updates. Each data point represents the average latency over multiple update operations to ensure reliability. We use 32-byte keys and 1024-byte values to maintain consistency with previous evaluations.

Figure 12 presents update latencies for each version (lower part) alongside corresponding index sizes (upper part). Once version pruning activates at the 40th version, the index size stabilizes, remaining unchanged across different data volumes. Notably, ViDB maintains consistent update performance throughout the pruning process. This stability results from our LVPS layer design, which eliminates the intensive I/O operations typically associated with garbage collection in traditional systems. Consequently, version pruning operates transparently without introducing significant performance degradation to update latency.

Figure 13 compares disk utilization across different systems as data volume increases. ViDB demonstrates significantly lower storage overhead compared to baseline systems lacking version pruning capabilities (SQL Ledger, LedgerDB, QLDB). QMDB exhibits similarly low storage consumption, but this comes at the cost of eliminating historical query capabilities entirely—a fundamental limitation for ledger database applications. ViDB achieves a balance by providing both historical data access and efficient storage management through intelligent version pruning.

Rollback. We evaluate rollback efficiency across varying data volumes and rollback distances (5 to 50 versions). As shown in Figure 14, rollback latency remains consistently low and exhibits minimal sensitivity to both dataset size and rollback depth. This performance characteristic stems from ViDB's metadata-driven rollback mechanism, which performs rollback operations entirely in memory without requiring disk I/O or data reconstruction, enabling rapid recovery operations essential for production environments.

5.3 Real-world System Evaluation

In this evaluation, we demonstrate ViDB's effectiveness when deployed as the storage backend for blockchain systems, representing a real-world deployment scenario for ledger databases.

We integrate ViDB into Hyperchain, a production blockchain platform, replacing its default storage layer. To provide a solid comparison, we evaluate three configurations: the original Hyperchain with its native LSM-tree-based storage, Hyperchain with LETUS, and Hyperchain with ViDB. All systems process identical workloads using the Simple Payment and EVM Transfer settings described in Section 5.1. This end-to-end evaluation captures the complete performance impact of different storage backends under realistic blockchain transaction processing patterns.

Figure 15 demonstrates that ViDB's systematic design for cost-efficiency translates to superior end-to-end performance across both blockchain workloads. Under workloads with random access patterns over large-scale datasets, LETUS suffers from significant performance degradation due to its replay-based page access mechanism, which requires reading multiple physical pages to reconstruct a single logical page. This overhead causes LETUS to perform worse than even the original Hyperchain configuration for both Simple Payment and EVM Transfer workloads.

In contrast, ViDB enables direct page access through location-based indexing using physical identifiers (PIDs), eliminating the

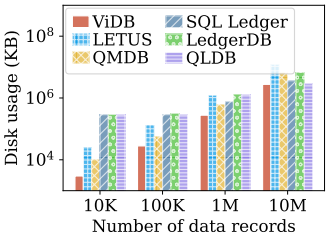


Figure 10: Disk usage.

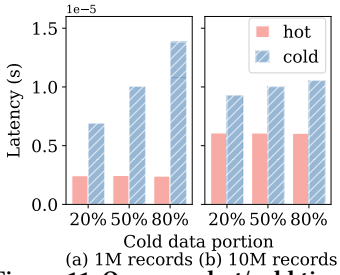


Figure 11: Query on hot/cold tier.

Figure 12: Update latency in each version.

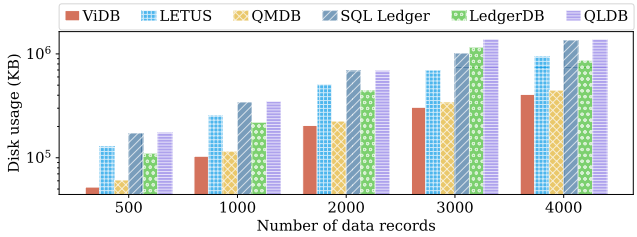
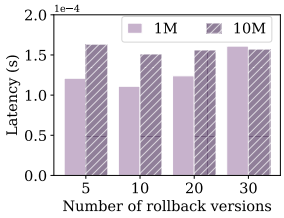


Figure 13: Disk usage for repeated updating.



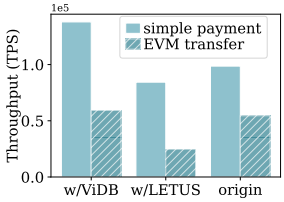


Figure 14: Version rollback. Figure 15: Real System Test.

time-consuming replay process entirely. This architectural advantage allows ViDB to maintain low disk I/O overhead while improving computational efficiency for both get and put operations at scale. For Simple Payment workload, ViDB significantly outperforms the original Hyperchain, whose LSM-tree backend suffers from background compaction overhead that degrades performance under large data volumes. Similarly, for EVM Transfer workload, ViDB achieves the highest throughput among all configurations, demonstrating its effectiveness in supporting complex smart contract executions that require frequent state access and modification.

6 Related Works

Verifiable storage. While blockchain systems provide fundamental verifiable storage capabilities, they suffer from significant limitations in query functionality and performance [39]. Even with optimized backend storage designs [2, 3, 8, 20, 26, 27, 29, 32], inherent characteristics such as decentralization overhead and per-transaction verification requirements continue to hinder blockchain storage performance. Ledger databases [1, 4, 10, 12, 14, 18, 35, 37–39, 44], in contrast, provide data immutability and tamper evidence guarantees while delivering better performance compared to traditional blockchain systems. However, as data volumes scale, existing ledger databases struggle to maintain cost efficiency across storage consumption, I/O bandwidth, and computational resources.

Verifiable index structure. The verifiable index structure is critical for verifiable storage in data indexing and verification. As the data volume grows, ledger databases introduce an index to speed up query processing [1, 4]. However, a large index also incurs high disk access costs. RainBlock [27] and LMPT [8, 9] utilize a layered tree as an index and keep a small-sized top layer in memory to reduce disk access. But they introduce the cost of periodic tree merging when processing updates.

Page Management in ledger database. Early ledger databases utilize LSM-based storage, such as levelDB [11] and RocksDB [24], to store the verifiable index nodes [1, 33, 39]. The randomness of node identifiers omits the locality of nodes, leading to write amplification due to heavy background compaction. ChainKV [6] and Block-LSM [7] restore locality by using tree level or version as the prefix of the node identifier, which poses a strong assumption of the key format. LETUS [31] adopts delta tree encoding to reduce write amplification, at the cost of the replay process when fetching pages.

Garbage collection. Generic garbage collection [19, 36, 45] used by programming languages and system runtimes has to track the liveness of objects continuously, due to the lack of assumption about the life-cycle of objects. In databases, the garbage collection can be optimized since the life-cycle of objects follows the semantics [5, 17, 30]. However, the garbage collection process is rarely discussed in ledger databases, even though it is necessary for version operations. Therefore, we propose ViDB, enabling garbage collection process in-memory and optimizing the efficiency of version operations.

7 Conclusion

In this paper, we present ViDB, a ledger database maintaining low hardware costs with scaling data volume. ViDB mainly aims at reducing costs of storage space, I/O bandwidth and computation resources. ViDB comprises a interface layer supporting rich database functionality, an index layer and a storage layer. Base on time-based tree partition, the index layer bounds computational complexity for inserting, point lookup and range scan. The storage layer is designed to achieve low I/O and space amplification as well as in-memory garbage collection process. Experimental evaluation shows that ViDB achieves 2× ~ 20× performance improvement and over 70% storage reduction comparing with existing ledger databases.

References

[1] Amazon. 2025. *Amazon Quantum Ledger Database.* Amazon Web Services, Inc. https://aws.amazon.com/qldb

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event, China, 76–88.

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, Porto,Portugal, 1–15.

[4] 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 *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event, China, 2437–2449.

[5] Rodrigo Bruno and Paulo Ferreira. 2018. A Study on Garbage Collection Algorithms for Big Data Environments. *Comput. Surveys* 51, 1 (2018), 20:1–20:35.

[6] Zehao Chen, Bingzhe Li, Xiaojun Cai, Zhiping Jia, Lei Ju, Zili Shao, and Zhaoyan Shen. 2023. ChainKV: A Semantics-Aware Key-Value Store for Ethereum System. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–23.

[7] Zehao Chen, Bingzhe Li, Xiaojun Cai, Zhiping Jia, Zhaoyan Shen, Yi Wang, and Zili Shao. 2021. Block-LSM: An Ether-aware Block-ordered LSM-tree Based Key-Value Storage Engine. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, Storrs, CT, USA, 25–32.

[8] Jemin Andrew Choi, Sidi Mohamed Beillahi, Peilun Li, Andreas Veneris, and Fan Long. 2022. LMPTs: Eliminating Storage Bottlenecks for Processing Blockchain Transactions. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, Shanghai, China, 1–9.

[9] Jemin Andrew Choi, Sidi Mohamed Beillahi, Srisht Fateh Singh, Panagiotis Michalopoulos, Peilun Li, Andreas Veneris, and Fan Long. 2024. LMPT: A Novel Authenticated Data Structure to Eliminate Storage Bottlenecks for High Performance Blockchains. *IEEE Transactions on Network and Service Management* 21, 2 (2024), 1333–1343.

[10] Johannes Gehrke, Lindsay Allen, Panagiotis Antonopoulos, Arvind Arasu, Joachim Hammer, Jim Hunter, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Srinath T. V. Setty, Jakub Szymaszek, Alexander van Renen, Jonathan Lee, and Ramarathnam Venkatesan. 2019. Veritas: Shared Verifiable Databases and Tables in the Cloud. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019*. www.cidrdb.org, Online, 1–9.

[11] Google. 2025. *Google/Leveldb*. Google. https://github.com/google/leveldb

[12] Google. 2025. *google/trillian: A transparent, highly scalable and cryptographically verifiable data store*. Google. https://github.com/google/trillian

[13] Hedera. 2022. *EA Licensed Esports Platform Realm Launches in Apex Legends*. Hedera. https://www.prnewswire.com/news-releases/ea-licensed-esports-platform-realm-launches-in-apex-legends-301668505.html

[14] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. 2021. Merkle2: A Low-Latency Transparency Log System. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 285–303.

[15] Hyperchain. 2025. *World Leading Enterprise-level Blockchain Platform*. Hangzhou Hyperchain Technology Co., Ltd. https://www.hyperchain.cn/en/

[16] Ledger Insights. 2023. *Telefonica Blockchain Tech Used to Fight Advertising Fraud*. Ledger Insights - blockchain for enterprise. https://www.ledgerinsights.com/telefonica-blockchain-advertising-fraud/

[17] Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-Lived Transactions Made Less Harmful. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (Sigmod '20)*. ACM, online conference [Portland, OR, USA], 495–510.

[18] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. 2020. CALYPSO: Private Data Management for Decentralized Ledgers. *Proceedings of the VLDB Endowment* 14, 4 (2020), 586–599.

[19] Yossi Levanoni and Erez Petrank. 2001. An On-the-Fly Reference Counting Garbage Collector for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, Tampa, Florida, USA, 367–380.

[20] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. 2024. LVMT: An Efficient Authenticated Storage for Blockchain. *ACM Transactions on Storage* 20, 3 (2024), 1–34.

[21] Feifei Li, Marios Hadjieftheriou, George Kollios, and Leonid Reyzin. 2008. Authenticated Index Structures for Outsourced Databases. In *Handbook of Database Security: Applications and Trends*. Springer, New York, NY, USA, 115–136.

[22] Hong Lin, Ke Chen, Dawei Jiang, Lidan Shou, and Gang Chen. 2024. Refiner: A Reliable and Efficient Incentive-Driven Federated Learning System Powered by Blockchain. *The VLDB Journal* 33, 3 (2024), 807–831.

[23] Ralph C. Merkle. 1982. *Method of providing digital signatures*. Leland Stanford Junior University. https://patents.google.com/patent/US4309569A/en

[24] Meta. 2025. *Facebook/Rocksdb*. Meta. https://github.com/facebook/rocksdb

[25] Satoshi Nakamoto. 2008. *Bitcoin: A Peer-to-Peer Electronic Cash System*. bitcoin. https://bitcoin.org/bitcoin.pdf

[26] Zhe Peng, Haotian Wu, Bin Xiao, and Songtao Guo. 2019. VQL: Providing Query Efficiency and Data Authenticity in Blockchain Systems. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, Macao, China, 1–6.

[27] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. 2021. {RainBlock}: Faster Transaction Processing in Public Blockchains. In *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX Association, Virtual Event, 333–347.

[28] Xiaodong Qi. 2022. S-Store: A Scalable Data Store towards Permissioned Blockchain Sharding. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. IEEE, London, United Kingdom, 1978–1987.

[29] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. mLSM: Making Authenticated Storage Faster in Ethereum. In *USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX Association, Boston, MA, USA, 1–6.

[30] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2023. One-Shot Garbage Collection for In-memory OLTP through Temporality-aware Version Storage. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[31] Shikun Tian, Zhonghao Lu, Haizhen Zhuo, Xiaojing Tang, Peiyi Hong, Shenglong Chen, Dayi Yang, Ying Yan, Zhiyong Jiang, Hui Zhang, and Guofei Jiang. 2024. LETUS: A Log-Structured Efficient Trusted Universal BlockChain Storage. In *Companion of the 2024 International Conference on Management of Data*. ACM, Santiago, Chile, 161–174.

[32] 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. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1137–1150.

[33] Gavin Wood. 2022. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* BERLIN VERSION, 934279c (2022), 1–41. https://ethereum.org/content/developers/tutorials/yellow-paper-evm/yellow-paper-berlin.pdf

[34] Chenyuan Wu, Mohammad Javad Amiri, Haoyun Qin, Bhavana Mehta, Ryan Marcus, and Boon Thau Loo. 2024. Towards Full Stack Adaptivity in Permissioned Blockchains. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1073–1080.

[35] Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. 2022. Litmus: Towards a Practical Database Management System with Verifiable ACID Properties and Transaction Correctness. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia, PA, USA, 1478–1492.

[36] Hirotaka Yamamoto, Kenjiro Taura, and Akinori Yonezawa. 1998. Comparing Reference Counting and Global Mark-and-Sweep on Parallel Computers. In *Languages, Compilers, and Run-Time Systems for Scalable Computers, 4th International Workshop, LCR '98*. Vol. 1511. Springer, Pittsburgh, PA, USA, 205–218.

[37] Xinying Yang, Sheng Wang, Feifei Li, Yuan Zhang, Wenyuan Yan, Fangyu Gai, Benquan Yu, Likai Feng, Qun Gao, and Yize Li. 2022. Ubiquitous Verification in Centralized Ledger Database. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, Kuala Lumpur, Malaysia, 1808–1821.

[38] Xinying Yang, Ruide Zhang, Cong Yue, Yang Liu, Beng Chin Ooi, Qun Gao, Yuan Zhang, and Hao Yang. 2023. VeDB: A Software and Hardware Enabled Trusted Relational Database. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[39] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. 2020. LedgerDB: A Centralized Ledger Database for Universal Audit and Verification. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3138–3151.

[40] Cong Yue, Tien Tuan Anh Dinh, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Xiaokui Xiao. 2023. GlassDB: An Efficient Verifiable Ledger Database System through Transparency. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1359–1371.

[41] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. 2020. Analysis of Indexing Structures for Immutable Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, online conference [Portland, OR, USA], 925–935.

[42] Cong Yue, Meihui Zhang, Changhao Zhu, Gang Chen, Dumitrel Loghin, and Beng Chin Ooi. 2023. VeriBench: Analyzing the Performance of Database Systems with Verifiability. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2145–2157.

[43] Isaac Zhang, Ryan Zarick, Daniel Wong, Thomas Kim, Bryan Pellegrino, Mignon Li, and Kelvin Wong. 2025. *QMDB: Quick Merkle Database*. LayerZero Labs. arXiv:2501.05262

[44] Meihui Zhang, Zhongle Xie, Cong Yue, and Ziyue Zhong. 2020. Spitz: A Verifiable Database System. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3449–3460.

[45] Benjamin Zorn. 1990. Comparing Mark-and Sweep and Stop-and-Copy Garbage Collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. ACM, Nice, France, 87–98.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009