ViDB: Cost-Efficient Ledger Database At Scale (Extended version)

Xinyu Chen[1] Hao Duan[2] Yang Sun[2] Cuihua Yang[2] Meihui Zhang[3]
Zhongle Xie[1] Ke Chen[1] Lidan Shou[1] Gang Chen[1]
[1] Zhejiang University, [2] Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
[3] Beijing Institute of Technology
{chenxy325,xiezl,chenk,should,cg}@zju.edu.cn,{duanhao,sunyang,yangcuihua}@bcds.org.cn,meihui_zhang@bit.edu.cn

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 consumption, disk I/O, and computational complexity 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 Temporal-Partitioned B$^+$-tree Forest (TPBF) that bounds storage consumption, disk I/O, and computational complexity by limiting tree heights through partitioning with configurable capacity, (2) Location-based Versioned Page Storage (LVPS) that eliminates write amplification by storing pages directly at physical locations without background compaction, as well as reduces the storage consumption of the page identifiers, and (3) in-memory garbage collection that enables efficient version operations without intensive disk I/O. Our evaluation against six 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.

PVLDB Reference Format:
Xinyu Chen[1] Hao Duan[2] Yang Sun[2] Cuihua Yang[2] Meihui Zhang[3] Zhongle Xie[1] Ke Chen[1] Lidan Shou[1] Gang Chen[1] . ViDB: Cost-Efficient Ledger Database At Scale (Extended version). PVLDB, 19(1): XXX-XXX, 2026.
doi:XX.XX/XXX.XX

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

1 Introduction

Ledger databases are increasingly adopted in diverse applications, such as healthcare [38], e-sports [14], advertising [17], and federated learning [24], where data immutability and verifiability are critical. While conventional blockchain-based systems [37] offer these guarantees, they often suffer from performance bottlenecks due to decentralization and runtime verification overheads, mechanisms that might be unnecessary in practical deployments [5, 43]. In contrast, ledger databases aim to strike a balance between integrity guarantees and high performance, providing flexible query support that better serves production workloads.

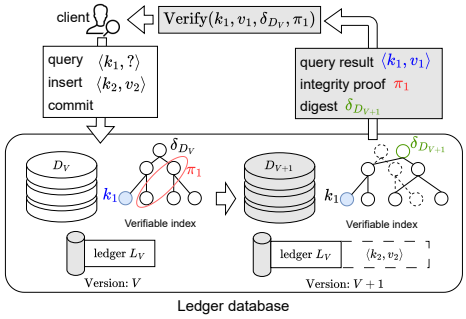


Figure 1: Ledger database functionality.

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. As illustrated in Figure 1, a client queries the current version dataset D_V for key k_1 with an integrity proof π_1, then inserts a new record with key k_2, resulting in an dataset of an updated version D_{V+1}, where V and $V+1$ are contiguous versions. Integrity guarantees and high performance rely on the *verifiable index*, a key component that unifies data indexing and authentication [35, 44].

The verifiable index combines data indexing with an authenticated data structure (ADS), typically realized as Merkle tree varieties [25]. Most existing verifiable indices employ a single-tree design, which maintains a single tree managing the entire dataset [45]. 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 verifiable storage, at the cost of 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: As data volume grows, the size of the verifiable index expands accordingly, and maintaining multiple versions further amplifies storage costs. In existing single-tree verifiable indices, scaling up the data volume leads to higher storage amplification. Prior studies have confirmed that storage usage rises sharply with the number of records and versions [45, 46].

Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Table 1: VIDB vs. existing ledger databases (shaded cells are designs with better efficiency in storage, disk I/O, and computation).

Challenge		LedgerDB [43]	SQLLedger [5]	QLDB [2]	QMDB [47]	LETUS [35]	**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**

Under strict storage constraints, aggressive version pruning may become necessary. However, current systems regard pruning as a low-frequency operation [35], causing new versions to stall with limited storage space. Moreover, storing tree nodes together with identifiers for lookup incurs additional storage overhead [2].

(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 from hashing requires background compaction for ordering, causing write amplification that incurs intensive disk I/O for repeatedly rewriting nodes [2, 43]. 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 [47].

(3) Heavy computational cost: With the increasing complexity of data operations, the ledger database consumes more computational resources to maintain the processing performance. In single-tree verifiable indices, the tree height increases with data volume, leading to the growth in computational complexity and 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 [5, 43]. 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) The VIDB designs the verifiable index structure as 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 reducing 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 life-cycle of pages, facilitating the garbage collection. VIDB performs garbage collection entirely through in-memory operations, and avoids intensive disk I/O.

The source code of VIDB is publicly available on github[1]. Our experiments show that, compared with existing ledger databases, VIDB achieves 2× to 20× improvement in transaction performance while reducing over 70% storage cost. We deploy VIDB on Hyperchain [16] to support high-performance, large-scale blockchain applications. 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 and space amplification, addressing fundamental scalability limitations of existing systems.
- We design a novel verifiable index that partitions data, and uses a forest with multiple trees to verifiably index data. Compared to the common single-tree design, the use of the forest makes the tree height under a configurable bound, improving the efficiency of the index.
- We introduce a location-based page indexing scheme that reduces write amplification and enables in-memory garbage collection without intensive disk I/O. In contrast to conventional hash-based page indexing, location-based page indexing avoids background compaction and the additional storage overhead for page identifiers.
- We conduct extensive experiments against six state-of-the-art systems, demonstrating 2× to 20× throughput improvements and up to 70% storage reduction. Integration with a commercial system, Hyperchain [16], further validates practical effectiveness in production blockchain environments.

2 Preliminary

Ledger databases are required to protect the integrity of data and its history [5, 43, 44]. The ledger database maintains a key-value dataset D and an append-only log L, called the *ledger*. The ledger database supports queries (point, range, and historical) and writes (insert, update, and delete). Upon receiving client commits, it consolidates D into a new version. As a result, there are multiple versions of D. We denote the dataset of a specific version V as D_V. Figure 1 illustrates how the system processes a client transaction on the latest version D_V. The ledger database retrieves the query results from D_V and generates an *integrity proof* π_1 to show that the result has not been tampered with. Then the ledger database appends the write operations to the ledger L and creates a dataset with a new version dataset D_{V+1}. For dataset D_V of each version V, the ledger database publishes a digest δ_{D_V} certified with cryptographic signature. To verify a query result $\langle k, v\rangle$ is from D_V, the client calls the function $\text{Verify}(k, v, \delta_{D_V}, \pi)$, which returns true if $\langle k, v\rangle$ is proved to be from D_V, and false otherwise. The ledger database relies on the verifiable index to generate the proof π and the digest δ_{D_V}.

Proof and digest generation. The verifiable index supports the generation of the integrity proof and digest. A verifiable index typically uses a Merkle tree [25], where each leaf stores a data record and each internal node stores the hash of its children. The integrity proof uses the hashes along the path from the root to the record, and the digest is simply the Merkle root hash.

[1] https://github.com/fishfishfishfishfish/ViDB

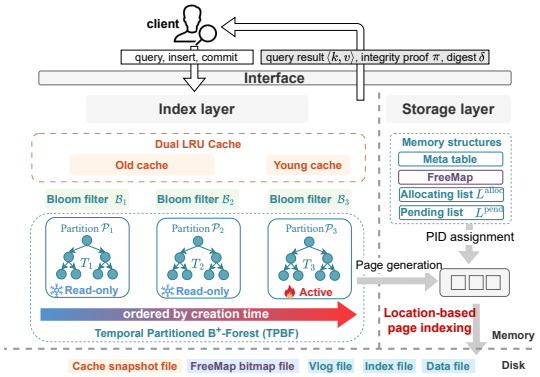


Figure 2: VIDB overview.

Three representative Merkle tree varieties have been widely applied [45]. MPT [37] and MBT [4] are used by Ethereum and Hyperledger Fabric, respectively, while Merkle B$^+$-tree [23] appears in academic systems [31, 44]. MPT splits the key k into a hexadecimal string that represents the path from the root to the leaf storing $\langle k, v\rangle$. MBT hashes records into N_B buckets and builds a Merkle tree over them. Merkle B$^+$-tree integrates Merkle hashes into the B$^+$-tree for integrity protection. For more details on Merkle trees, please refer to [45].

Version operations. A ledger database should allow clients to operate on data by version, including pruning versions older than a specified V and rolling back versions newer than V. *Version rollback* restores data to a prior version by removing all newer versions. It is particularly useful in distributed database systems [20, 34], where inconsistencies may arise from network partitions, concurrent updates, or system failures. In such scenarios, multiple databases must coordinate to roll back to a mutually consistent version, thereby ensuring correctness. *Version pruning* removes old data versions that are no longer needed for future access, thereby saving storage space. Version pruning is frequently triggered under storage constraints, especially when the database accumulates many versions that cannot be known in advance [5]. For example, in Bitcoin [27], pruning is triggered when the blockchain size exceeds a preconfigured storage threshold for normal nodes. Both pruning and rollback rely on garbage collection to locate target-version records and reclaim their storage space. However, garbage collection is I/O-intensive in existing designs.

3 VIDB Architecture

VIDB is a ledger database aiming at achieving cost efficiency in storage space, disk I/O, and computational cost, with data scaling. It implements a key-value data model augmented with Merkle proofs to ensure data integrity and support verifiable queries. VIDB employs a two-tier architecture comprising an index and a storage layer, as illustrated in Figure 2. The index layer 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.

3.1 Index layer

The index layer is the core component for data indexing and integrity proof generation, providing efficient and verifiable query processing, containing a verifiable index named TPBF, Bloom filters,

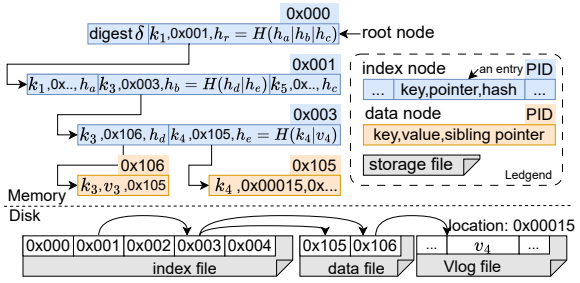


Figure 3: Tree structure.

and LRU caches. As introduced in §1, the index layer is optimized for storage, disk I/O, and computational complexity.

Temporal Partitioned B$^+$-tree Forest (TPBF). Instead of the conventional single-tree design, the index layer employs a novel temporal partitioned forest that addresses the issue of unbounded tree height growth as data volume increases. The core idea is to divide data into multiple fixed-size partitions, with each tree managing one partition. In this way, the tree height is bounded by the partition size rather than by the total data volume. Additionally, there is only a single active partition that accepts writes at any given time to avoid congestion. When the active partition reaches the capacity threshold N_{cap}, it becomes read-only, and a new partition is created for subsequent writes. Naturally, the partitions are temporally ordered according to their creation time. We denote a partition by $\mathcal{P}_i$, where a larger i indicates a newer partition. Each partition is managed by one Merkle B$^+$-tree. The TPBF is defined as a forest $F = \{T_1, T_2, ..., T_n\}$. Each tree $T_i \in F$ serves as both an index and a Merkle tree, independently managing query processing, write operations, and verification for $\mathcal{P}_i$. The tree of the active partition is called the active tree. To enable multi-versioning, TPBF employs Copy-on-Write semantics. Nodes are immutable once created, and any update generates new nodes, thereby maintaining consistency across versions and preserving historical data.

Tree structures. VIDB adopts Merkle B$^+$-tree to utilize its balanced structure and range query efficiency in TPBF, as depicted in Figure 3. 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 of the child subtree, and a hash field containing $\mathcal{H}(h_1|...|h_m)$ computed from child node hash values. The root node is a special index node that includes an additional digital signature as the digest. 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.

Bloom filters. The system employs Bloom filters to optimize queries for non-existent keys, which would otherwise require traversing the entire tree. Each $\mathcal{P}_i$ is equipped with a Bloom filter $\mathcal{B}_i$ that summarizes its key membership. Before traversing T_i for a key k, the system queries $\mathcal{B}_i$. If $\mathcal{B}_i(k)$ returns false, the traversal is skipped, since the Bloom filter guarantees $k \notin \mathcal{P}_i$. Given the partition size N_{cap}, the Bloom filter could be parameterized to achieve a low false-positive rate. We adopt standard Bloom filters [1, 6] instead of more

Table 2: Application programming 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)			

complex alternatives such as cuckoo filters, due to their efficiency in memory and computation, as well as the ease of deployment. As discussed in §4.1.1, Bloom filters are critical for point queries, as they avoid traversing all trees in TPBF for a single key.

Dual LRU Cache. To exploit query locality, VIDB maintains two least recently used (LRU) caches: a *young cache* for the active partition and an *old cache* for read-only partitions, in order to adapt to the different access frequencies of the active and read-only partitions. Records go to the young cache if from the active partition, or the old cache if from read-only partitions. Entries in the old cache are removed if later retrieved from the active partition.

Discussion. The TPBF design provides several key advantages for authenticated data management. First, the temporal partitioning approach ensures bounded computational complexity by keeping tree heights stable as data grows, unlike monolithic tree structures, whose height grows with data size. Partition size N_{cap} can be configured to limit the max tree height h_{max} through the relationship $h_{\text{max}} = \log_m N_{\text{cap}}$, where m is the tree fanout. Then, the Bloom filters and dual LRU cache further reduce the query complexity, as discussed in §4. Next, restricting updates to a single active tree reduces the number of modified nodes during write operations, thus mitigating write amplification as analyzed in §4.2. Finally, the temporal ordering of partitions enables efficient tiered storage management, as older and less frequently accessed partitions are easily identified and migrated to lower-cost storage tiers without additional metadata or scanning operations.

3.2 Storage layer

The storage layer provides compact node storage for both records and Merkle tree nodes while enabling efficient page mapping between memory and disk. With a novel location-based page indexing scheme, our *Location-based versioned page storage (LVPS)* controls write and space amplification, supports efficient version operations, and reduces disk I/O of garbage collection.

VIDB uses a page-oriented storage architecture, placing each index or data node in a fixed-size page within the index or data file. Page identifiers (PIDs) are assigned based on physical disk locations for 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 consumption. Figure 2 shows that LVPS supports PID assignment for the pages from 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 would normally determine them only after a page is flushed to disk, which stalls index-layer tree construction. To avoid such stalls, VIDB employs two memory structures that enable early PID determination, as illustrated in Figure 2: *Meta table* maintains version-specific entries each recording root PID, max index PID, and max data PID, while the *FreeMap* organizes free pages into contiguous page spans. Each

Table 3: Notations.

Notation	Meaning	Notation	Meaning
N_{rec}	total data volume	|Page|	page size
N_{cap}	partition capacity	|PID|	PID size
N_{cpu}	number of CPUs	$\|k\|/\|v\|$	key/value size
Φ	disk access cost	N_R	result set size of a range query
m	tree fanout		

FreeMap entry maps span sizes to lists of starting PIDs, enabling efficient free page location without disk access. For new page allocation, LVPS first queries FreeMap; if no free pages are available, it increments the maximum PID from Meta table.

To support garbage collection, VIDB maintains two auxiliary lists: an allocating list L^{alloc} for pages allocated from FreeMap during version rollbacks, and a pending list L^{pend} for pages updated in later versions and eligible for pruning. They enable efficient garbage collection without excessive disk access, as detailed in §4.3. VIDB persists FreeMap as a space-efficient bitmap file, while storing Meta table, allocating list, and pending list in the index file.

Discussion. The location-based page indexing design of LVPS offers three main advantages over existing solutions: (i) constant write amplification: each page is written only once in LVPS, unlike LSM-based systems that rewrite pages during compaction [2, 5, 37]; (ii) lower update and retrieve complexity: LVPS direct addresses the page with PID, and avoids costly searches and page sorting required by content-based schemes [5, 35]; (iii) reduced space amplification: LVPS does not explicitly store PID mapping, unlike content-based approaches that must store page identifiers.

3.3 Application programming interface

VIDB 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 queries 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 versions (F8), and pruning of outdated versions (F9) for storage optimization.

Example 3.1 (VIDB processing workflow). We use the client in Figure 1 as an example. To issue a point query, the client calls Get(k_1) and receives the result $\langle k_1, v_1 \rangle$. For verification, the client invokes Proof(V, k_1) to obtain a proof π_1. The client verifies data integrity by using π_1 to recompute a digest and checking its consistency against the previously published digest δ_{D_V}. The client calls Put(k_2, v_2) to insert a new record, and then calls Commit($V + 1$) to consolidate the updates. VIDB creates the new version $V + 1$ and publishes a new digest $\delta_{D_{V+1}}$.

4 Processing

In this section, we elaborate on how VIDB utilizes TPBF and LVPS to achieve verifiable query, write operations, data versioning, and garbage collection with cost efficiency. We close this section with a theoretical cost analysis to show the benefits of VIDB. Notations used in this section are listed in Table 3.

4.1 Verifiable query processing

We elaborate on the processing of point, range, and historical queries.

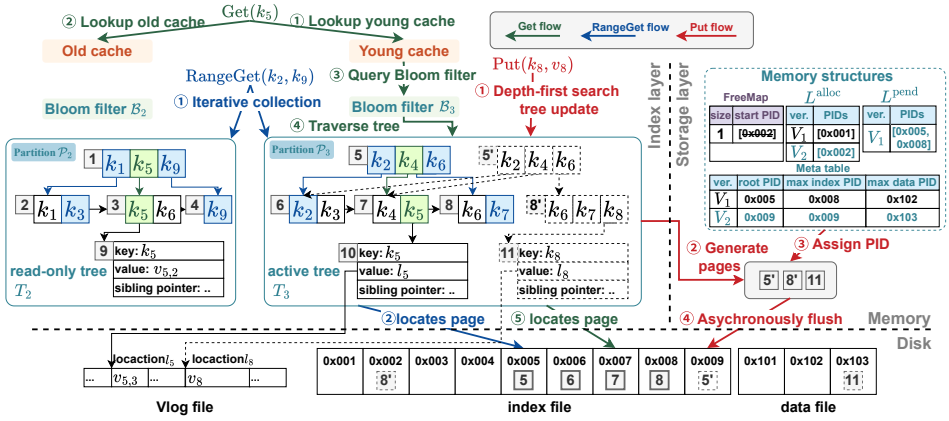


Figure 4: Running example of VIDB.

Figure 5: An example for FreeMap.

Algorithm 1 `Point query`

Input: queried key k;
Output: value v;
1: v = cache-lookup(k)
2: **if** v is found **then return** v
3: **for** $\mathcal{P}$ in $[\mathcal{P}_{N_{\text{rec}}/N_{\text{cap}}}, .., \mathcal{P}_0]$ **do**
4: **if** Bloom-test($\mathcal{P}$, k) **then**
5: v = tree-traverse($\mathcal{P}$, k)
6: **if** v is found **then** break;
7: **end if**
8: **end for**
9: **return** v

4.1.1 Point query The client can issue a point query using the Get(k) API, which is processed through cache lookup, Bloom filter testing, and tree traversal (described in Algorithm 1).

Step 1: Cache lookup. VIDB first checks both the young and old caches and returns the cached record and proof (Line 1). On a cache miss, it searches the target key k by scanning the partitions in order from newest to oldest.

Step 2: Bloom filter testing. For each partition, the Bloom filter is tested (Line 4) to exclude partitions that do not contain k. The partition skips tree traversal if the Bloom filter returns false.

Step 3: Tree traversal. If the Bloom filter returns true, VIDB then traverses the tree to locate the target record, collecting hashes from sibling nodes along each traversal path to construct the integrity proof (Line 5). The record and proof are added to the young cache if they come from the active partition; otherwise, they are added to the old cache. Once the target record is found, VIDB stops searching and returns the result. When the target record is in a recent partition, queries terminate early without visiting older partitions.

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 lookup the young and old cahces (①②). Assuming k_5 is not cached but exists, VIDB searches $\mathcal{P}_3$ and tests Bloom filter $\mathcal{B}_3$ (③). If $\mathcal{B}_3$ returns true, VIDB traverses T_3 (④). VIDB visits nodes 5, 7, and 10 to retrieve $v_{5,3}$ from the value log file. During the traversal, nodes are directly located on disk based on the location-based page indexing (⑤). VIDB returns $\langle k_5, v_{5,3}\rangle$ along with the corresponding integrity proof. The result and proof are cached in the young cache for future queries. Although $\mathcal{P}_2$ contains $\langle k_5, v_{5,2}\rangle$, it is skipped, since the target record is found in a more recent partition.

Analysis. The point query processing complexity, including cache lookup, Bloom filter testing, and tree traversal, is given in Eq. 1.

$$C_{\text{point}} = \underbrace{O(1) + O\left(\frac{N_{\text{rec}}}{N_{\text{cap}} N_{\text{cpu}}}\right)}_{\text{memory access}} + \underbrace{O\left(\Phi(1 + \frac{N_{\text{rec}}}{N_{\text{cap}}} R_{\text{FP}}) \log_m N_{\text{cap}}\right)}_{\text{disk access}} \quad (1)$$

$$\approx O\left(\frac{N_{\text{rec}}}{N_{\text{cap}} N_{\text{cpu}}} + \Phi \log_m N_{\text{cap}}\right) \quad (2)$$

The cache lookup complexity is $O(1)$. Bloom filters can be parallelized, lowering the effective complexity to $O(\frac{N_{\text{rec}}}{N_{\text{cap}} N_{\text{cpu}}})$, where N_{cpu} is the number of CPUs. A point query traverses $(1 + \frac{N_{\text{rec}}}{N_{\text{cap}}} R_{\text{FP}})$ trees. With a low false positive rate, R_{FP}, of the Bloom filter, it degrades to single-tree traversal (Eq. 2). By limiting tree height, TPBF reduces traversal complexity from $O(\Phi \log N_{\text{rec}})$ to $O(\Phi \log N_{\text{cap}})$, with $N_{\text{cap}} \ll N_{\text{rec}}$. Φ denotes the disk access cost for reading a node, which is higher than memory access.

Eq. 3 gives the read amplification of a point query, defined as the ratio between the physical bytes read from disk and the logical bytes required by the query (i.e., the size of a record).

$$A_{\text{point}} \approx O\left(\frac{|\text{page}|}{|k| + |v|} \log_m N_{\text{cap}}\right) \quad (3)$$

With reduced tree height, the read amplification of TPBF is lower than that of a single-tree verifiable index, which is $O(\frac{|\text{page}|}{|k|+|v|} \log N_{\text{rec}})$ for $N_{\text{rec}} \gg N_{\text{cap}}$. Eq. 4 estimates the proof size for a record.

$$S_{\text{proof}} = O\left(m \log_m N_{\text{cap}}\right) \quad (4)$$

4.1.2 Range query The client use the RangeGet($k_{\text{lb}}, k_{\text{ub}}$) API to retrieve records in the key range $[k_{\text{lb}}, k_{\text{ub}}]$. Unlike a point query result resides in one partition, range query results may span multiple partitions, and Bloom filters are ineffective. Instead, VIDB adopts an iterative collection strategy (Algorithm 2).

Iterative collection. At the beginning, there is an iteration initialization for each partition (Line 2). The init-iterator function traverses the tree of each partition, and finds the record with minimal $k \geq k_{\text{lb}}$ as the current record. Afterward, VIDB iterates to retrieve the result set, with each iteration retrieving one record via a forward phase and a collect phase. The forward phase gets a candidate record from each partition. It first checks each partition's current record (Line 9). If the current key is smaller than the last

Algorithm 2 `Range query (iterative collection)`

Input: key range $[k_{\text{lb}}, k_{\text{ub}}]$;
Output: result set $[(k_{\text{lb}}, v_{\text{lb}}), ..., (k_{\text{ub}}, v_{\text{ub}})]$;
1: PartitionList = $[\mathcal{P}_{N_{\text{rec}}/N_{\text{cap}}}, .., \mathcal{P}_0]$
2: **for** $\mathcal{P}$ in PartitionList **do** init-iterator($\mathcal{P}$, k_{lb})
3: ResultSet = [], $\langle k^*, v^* \rangle$ = null
4: /* iterative collection */
5: **while** PartitionList is not empty **do**
6: /* forward phase */
7: CandidateList = []
8: **for** $\mathcal{P}$ in PartitionList **do**
9: $\langle k, v \rangle$ = current($\mathcal{P}$, k_{lb})
10: **if** $\langle k^*, v^* \rangle$is null **or** $k \leq k^*$ **then** $\langle k, v \rangle$ = next($\mathcal{P}$, k_{lb})
11: **if** $k > k_{\text{ub}}$ **then** PartitionList $-\{\mathcal{P}\}$ **else** CandidateList appends $\langle k, v \rangle$
12: **end for**
13: /* collect phase */
14: $\langle k^*, v^* \rangle$ = first record in CandidateList
15: **for** $\langle k, v \rangle$ in CandidateList **do if** $k < k^*$ **then** $k^* = k$
16: append $\langle k^*, v^* \rangle$ to ResultSet
17: **end while**
18: **return** ResultSet

result, it is discarded and the next record is retrieved via the sibling pointer (Line 10). If the current record key exceeds k_{ub}, the partition is excluded from further processing. Otherwise, the current record is added to the CandidateList (Line 11). The collect phase identifies the newest smallest key in the CandidateList. The collect phase updates k^* when it encounters a smaller key k (Line 15). After scanning the CandidateList, k^* represents the newest smallest key, since newer partitions are processed earlier during the forward phase.

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. We denote the value of k_i from $\mathcal{P}_j$ as $v_{i,j}$. VIDB traverses the trees concurrently to initialize the iteration. After the initialization, the current record of $\mathcal{P}_2$ is $\langle k_3, v_{3,2} \rangle$ and $\mathcal{P}_3$ is $\langle k_2, v_{2,3} \rangle$. In the first iteration, the forward phrase adds $\langle k_3, v_{3,2} \rangle$ and $\langle k_2, v_{2,2} \rangle$ to the CandidateList, and the collect phase adds $\langle k_2, v_{3,2} \rangle$ to the ResultSet with the smaller key. In the second iteration, $\langle k_3, v_{3,2} \rangle$ and $\langle k_3, v_{3,3} \rangle$ is added to the CandidateList, and $\langle k_3, v_{3,3} \rangle$ is added to the ResultSet with the newer key. VIDB continues iterating until T_2 reaches k_9 and T_3 reaches k_7. Final results include $\{k_9\}$ from $\mathcal{P}_2$ and $\{k_2, k_3, k_4, k_5, k_6, k_7\}$ from $\mathcal{P}_3$.

Analysis. The overall complexity of a range query is given in Eq. 5, including initialization, forward phase, and collect phase. Initialization can be parallelized, with complexity $O(\Phi \log N_{\text{cap}})$ per partition. In a forward phase, all partitions retrieve the next record in the worst case, yielding complexity of $O(\Phi \frac{N_{\text{rec}}}{N_{\text{cap}}})$. The collect phase scans the CandidateList with a complexity of $O(\frac{N_{\text{rec}}}{N_{\text{cap}}})$.

$$C_{\text{range}} = O\left(\underbrace{\Phi(N_{\text{rec}}/(N_{\text{cap}}N_{\text{cpu}}))(\log N_{\text{cap}} + N_{\text{R}})}_{\text{disk access}} + \underbrace{(N_{\text{rec}}/N_{\text{cap}})N_{\text{R}}}_{\text{memory access}}\right) \quad (5)$$

$$\approx O\left(\Phi \log N_{\text{cap}} + \Phi N_{\text{R}} + (N_{\text{rec}}/N_{\text{cap}})N_{\text{R}}\right) \quad (6)$$

When the number of CPUs is sufficient, i.e., $N_{\text{rec}}/(N_{\text{cap}}N_{\text{cpu}}) \leq 1$, the complexity simplifies to Eq. 6. As data volume grows, the number of trees may exceed the available CPUs, but the multi-tree design is well-suited for horizontal hardware scaling that increases concurrency for larger data scales.

Eq. 7 gives the read amplification of a range query in VIDB.

$$A_{\text{range}} = O\left(\frac{|\text{Page}|}{|k| + |v|} \frac{N_{\text{rec}}}{N_{\text{cap}}} (\log N_{\text{cap}} + N_{\text{R}})\right) \quad (7)$$

The result is comparable to that of a single-tree verifiable index, which is $O(\frac{|\text{Page}|}{|k|+|v|}(\log N_{\text{rec}} + N_{\text{R}}))$. Since the proof of range query consists of the proof for each record in the result set, the per-record proof size is the same as we estimate in Eq. 4.

4.1.3 Historical query A historical version query, invoked via HistGet(V, k), retrieves the record with key k at 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 Meta table records the root PID of all versions, VIDB can retrieve any version at constant cost. In contrast, prior systems store versions as linked lists, resulting in longer I/O paths for older versions. Historical queries have the same complexity, read amplification, and proof size as point queries (see §4.1.1).

4.2 Write operations

VIDB supports three write operation types: inserting or updating a record via the Put(k, v) API, and removing a record via the Delete(k) API. The write operations are consolidated for query after the client invokes the Commit(V) API. Write operations follow a similar processing flow, with an extra optimization dedicated to deletions.

Depth-first tree update. To apply write operations, VIDB traverses and updates the active tree using depth-first search. The search is implemented with an in-memory first-in last-out (FILO) stack. Starting from the root, VIDB visits the child whose subtree contains the target key k, pushing each visited node onto the stack. Upon reaching the last level of index nodes, VIDB creates a new data node for insert or update operations. For deletes, however, VIDB does not immediately remove the data node in order to reduce disk accesses; the details are explained later in this section. VIDB then backtracks by popping nodes from the stack, updating each node based on its child before passing it to the storage layer for persistence. Depth-first search is chosen over breadth-first search to support efficient backtracking.

Batched write. VIDB processes multiple write operations in a batch to amortize node update overhead. Incoming write operations are buffered in an in-memory key-value store implemented as a skip list, where inserts and updates store key-value pairs, and deletes store the key with a tombstone. Write operations are sorted by key, and only the latest is retained for duplicated keys. VIDB updates the active tree when the client calls the Commit(V) API. VIDB does not pop all nodes from the FILO stack for backtracking. VIDB stops backtracking when it pops a node containing the key of the next write operation in the skip list, and resumes depth-first search to update the tree for the next operation. The process ends when all write operations are applied.

In-place deletion. For delete operations, VIDB does not create a new data node but instead replaces the value with a tombstone in the Vlog file. Being smaller than the original value, the tombstone does not overwrite adjacent data in the Vlog file. In-place deletion has two benefits: (i) it avoids writing a new data node to disk, and (ii) repeated deletions of the same key within a partition leave only one tombstone, meaning a key produces at most $N_{\text{rec}}/N_{\text{cap}}$ tombstones.

Example 4.3 (Insertion). Figure 4 illustrates how VIDB inserts a record $\langle k_8, v_8 \rangle$. Newly created nodes are shown with a dashed border, and the processing flow is marked red. VIDB traverses the active tree T_3 and pushes node 5 and 8 into the FILO stack(①). Since node 8 is at the last level of index nodes, VIDB creates a data node 11

Table 4: Processing costs bound of varieties of Merkle trees.

OPs / Index	storage per record	proof size	write complexity	query processing complexity	
				point query	range query
MPT	$(\|\text{Page}\| + \|\text{PID}\|)\|k\|$	$m\|k\|$	$\Phi\|k\|$	$\Phi\|k\|$	$\Phi L_R\|k\|$
MBT	$(\|\text{Page}\| + \|\text{PID}\|)\log_m N_B$	$m\log_m N_B + \frac{N_{\text{rec}}}{N_B}$	$\Phi(\log_m N_B + \frac{N_{\text{rec}}}{N_B})$	$\Phi(\log_m N_B + \log_2 \frac{N_{\text{rec}}}{N_B})$	$\Phi(L_R(\log_m N_B + \log_2 \frac{N_{\text{rec}}}{N_B}))$
Merkle B$^+$-tree	$(\|\text{Page}\| + \|\text{PID}\|)\log_m N_{\text{rec}}$	$m\log_m N_{\text{rec}}$	$\Phi(\log_m N_{\text{rec}})$	$\Phi(\log_m N_{\text{rec}})$	$\Phi(N_R + \log_m N_{\text{rec}})$
TPBF (Ours)	$\|\text{Page}\|\log_m N_{\text{cap}}$	$m\log_m N_{\text{cap}}$	$\Phi(\log_m \frac{N_{\text{cap}}}{N_{\text{batch}}} + 1)$	$\Phi \log N_{\text{cap}} + \frac{N_{\text{rec}}}{N_{\text{cap}} N_{\text{cpu}}}$	$\frac{N_{\text{rec}}}{N_{\text{cap}}}(\Phi \frac{\log N_{\text{cap}} + N_{\text{R}}}{N_{\text{cpu}}} + N_{\text{R}})$

to store k_8 and v_8's location in the Vlog file. VIDB then backtracks, pops nodes 8 and 5 from the FILO stack, updates their pointers and hash values, and stores them in LVPS as nodes 8' and 5'.

The insertion generates three pages for node 11, 8', and 5'(②). LVPS assigns PID 0x002 to node 8' based on FreeMap, and assigns PID 0x009 and 0x103 to node 5' and 11 using 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 FreeMap and added into L^{alloc}. Meta table creates 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 PID of the new version. L^{pend} records page 0x005 and 0x008 as pending pages.

Analysis. Write complexity depends on the number of updated nodes. Although TPBF contains multiple trees, only one tree is active for write operations. Hence, each write operation updates only $\log_m N_{\text{cap}}$ nodes, fewer than the $\log_m N_{\text{rec}}$ nodes in a single-tree verifiable index. For a batch of N_{batch} writes, each node on disk is accessed at most once, and the number of node reads/writes is bounded by $O((\log_m \frac{N_{\text{cap}}}{N_{\text{batch}}} + 1)N_{\text{batch}})$. Eq. 8 calculates the average write operation complexity for batched writes, which approaches $O(\Phi(\log_m N_{\text{cap}} + 1))$ for small batch sizes, and approaches $O(\Phi)$ for large batch sizes.

$$C_{\text{write}} = O\left(\Phi(\log_m(N_{\text{cap}}/N_{\text{batch}}) + 1)\right) \tag{8}$$

Write amplification, given in Eq. 9, is defined as the ratio of physical bytes written to disk to the logical bytes of a write operation (i.e., the record size). Write amplification approaches $O(\frac{|\text{Page}|}{|k|+|v|}(\log_m N_{\text{cap}} + 1))$ for small batch sizes, and $O(\frac{|\text{Page}|}{|k|+|v|})$ for large batch sizes.

$$A_{\text{write}} = O\left(\frac{|\text{Page}|}{|k| + |v|}(\log_m \frac{N_{\text{cap}}}{N_{\text{batch}}} + 1)\right) \tag{9}$$

Eq. 10 gives the storage space required per record in VIDB.

$$A_{\text{storage}} = O\left(|\text{Page}| \log_m N_{\text{cap}}\right) \tag{10}$$

4.3 Version operations with garbage collection

We explain how VIDB processes garbage collection during version pruning and rollback without intensive I/O utilizing the LVPS. For version pruning, VIDB prunes all versions before V_{prune}. For version rollback, VIDB rollbacks the data to version V_{roll}. Both operations remove data and require garbage collection to reclaim the disk space, containing three steps.

Step 1: Identify the free pages. Garbage collection first identifies the free pages whose content is removed after version pruning or rollback, which requires intensive disk I/O to scan all the pages in previous work [47]. VIDB finds free pages without accessing the disk, based on allocating list L^{alloc} and pending list L^{pend}. 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 without scanning all the pages.

Step 2: Remove reference to the free pages. In previous systems, garbage collection updates all pages referencing free pages to maintain pointer consistency and prevent dangling references. In VIDB, version rollback does not produce dangling references because pages are only referenced by other pages of the same or newer version. Version pruning also avoids dangling references, since each pending page in L^{pend} is referenced by a page that is pended in the same version. Therefore, VIDB does not need to explicitly locate and update the pages that reference free pages.

Step 3: Track the free pages. At last, garbage collection has to record a large number of free pages for future reuse. VIDB uses FreeMap to record free pages, which compactly organizes free pages into spans. Figure 5 shows a FreeMap maintaining five page spans The second entry records two spans, each of size 2. The first span starts from page 0x007, indicating 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 FreeMap. Then, VIDB removes all the entries associated with V_1, that is, the entry V_1 in Meta, L^{alloc}, and L^{pend}. The whole process does not access the disk.

Example 4.5 (Version rollback). Based on Figure 4, we describe how VIDB rollbacks to version V_1, which requires removing V_2. VIDB first identifies free pages from the entry of V_2 in allocating list L^{alloc}, i.e. page 0x002. VIDB inserts page 0x002 into FreeMap. At last, VIDB deletes the entries of V_2 in Meta table, L^{alloc}, and L^{pend}. No disk access occurs during the process.

Discussion. Compared with previous works, VIDB significantly reduces the disk I/O for version operations. For version pruning, existing systems, such as QMDB [47] and LETUS [35], require scanning the disk to identify pages of the pruned version. For rollback, Ethereum [37] 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 [35] merge the free pages by reorganizing all the valid data into new files, consuming significant disk I/O. With FreeMap, VIDB efficiently tracks all free pages and reuses them without data reorganization. Therefore, VIDB achieves efficient garbage collection for version pruning and rollback, without intensive disk I/O. In fact, no disk access is required during the garbage collection process.

4.4 Benefits of VIDB

In this section, we analyze the advantages of VIDB in the storage consumption, I/O overhead, and computational complexity, comparing with previous ledger databases that adopts single-tree verifiable indices (MPT, MBT, Merkle B$^+$-tree), summarized in Table 4.

Storage consumption. As seen in Table 4, ViDB reduces the storage consumption with the multi-tree design of TPBF and compact storage of LVPS. In single-tree verifiable index designs, each record is expanded into multiple index nodes, which grow with data volume. Each node additionally requires storage for page identifiers (i.e., |PID|). The multi-tree design of ViDB reduces the number of index nodes to $\log_m N_{\text{cap}}$, independent of tree height. Its location-based page indexing eliminates the need for storing PIDs, reducing the storage of each node from |Page| + |PID| to |Page|.

I/O overhead. ViDB reduces the disk I/O overhead with lower write amplification, small proof size, and efficient garbage collection. For ledger databases based on LSM storage [2, 5], writing nodes to disk incurs background compaction, resulting in a write amplification of $O(\frac{|\text{Page}|}{|k|+|v|}(N_{\text{rec}} \log_m N_{\text{rec}}))$. Recent systems that eliminate background compaction [35] reduce write amplification to $O(\frac{|\text{Page}|}{|k|+|v|}(\log_m N_{\text{rec}}))$. ViDB achieves even lower write amplification, ranging from $O(\frac{|\text{Page}|}{|k|+|v|})$ to $O(\frac{|\text{Page}|}{|k|+|v|}(\log_m N_{\text{cap}} + 1))$, depending on the batch size. In addition, the proof generated for verification requires additional I/O bandwidth for transfer. As summarized in Table 4, single-tree verifiable indices generate proofs that grow with data volume, whereas TPBF's temporal partitioning ensures that proof size remains independent of data volume. Finally, garbage collection in ViDB avoids intensive I/O. With in-memory structures L^{pend} and L^{alloc}, it does not require repeated disk scans to clean outdated data, unlike previous systems [35, 47].

Computational complexity. High write and query complexities demand significant computational resources for deployment. As summarized in Table 4, ViDB reduces the computational complexities for write operations, point query, and range query. For single-tree verifiable indices, complexity depends mainly on key length $|k|$ and record count N_{rec}, and grows with data volume. Range queries are especially costly, and grow with range size $L_R = k_{\text{ub}} - k_{\text{lb}}$ and result size N_R. Consequently, they suffer from unbounded cost growth with data volume and cannot leverage concurrency for acceleration. ViDB bounds complexity by partition capacity N_{cap}, achieving lower costs than single-tree indices, and further accelerates point and range queries through concurrency.

5 Evaluation

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

5.1 Setup

Implementation. ViDB, implemented in Go, is released online for reproduction[2]. Note that the Bloom filters are implemented with an online library[3] and the false-positive rate is set to 0.01. The cache is implemented via ristretto library[4] for its high hit ratios and fast lookups. N_{cap} is configured between 2M and 200M records according to the data volume. The page size is 4KB, and the tree fanout m is 64.

Baselines. We compare ViDB against six state-of-the-art ledger databases, including LETUS [35] and LedgerDB [43] from Ant, QLDB [2] from Amazon, SQL Ledger [5] from Microsoft, GlassDB [44] and QMDB [47] from academia. We use the implementation of QLDB, LedgerDB, SQL Ledger, and GlassDB released on github[5]. We also employ the open-source code for QMDB[6]. LETUS is closed-source, hence we implement it based on its original paper and share our code online[7]. In addition, to evaluate the performance under real blockchain workloads, we integrate ViDB and LETUS into a commercial system called Hyperchain [16] to store the blockchain state, where a block creates a data version.

Workloads. We employ three complementary benchmarks (listed in Table 5) for our evaluation. For systematic analysis, we fix the size of key (32 bytes) and value (1024 bytes), following LETUS [35]. The *micro-benchmark* evaluates the performance of ViDB on specific queries and operations, and compares it with baselines. The *macro-benchmark* evaluates the end-to-end performance of ViDB under generated but diverse database workloads at scale. The *real-world system evaluation* evaluates the end-to-end performance of ViDB in a blockchain scenario in production, leveraging Hyperchain [16] with two transaction types: (i) *Simple payment*, which randomly selects sender–receiver pairs for balance transfers, and (ii) *EVM transfer*, which invokes the ERC-20 [30] transfer function on random accounts.

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.

Metrics. We evaluate system performance using three key metrics: throughput (transactions per second), latency in seconds, and disk usage in bytes. These metrics collectively assess computational efficiency, execution speed, and storage cost-effectiveness.

5.2 Micro-benchmark

In this section, we evaluate ViDB's performance on updates, point queries, range queries, and historical queries under various experimental settings. We measure the throughput and latency of these operations, and compare ViDB against the baselines.

5.2.1 Point query We call the Put(k, v) and Get(k) APIs in batches on preloaded 1 million records per ledger database to measure latency and throughput. Each Put randomly selects a key for value update. Each Get performs a point query on a random key.

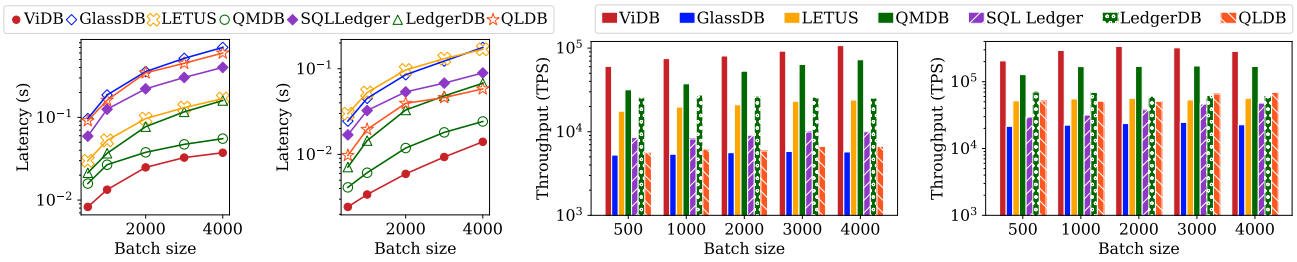
Figure 6(a) shows that ViDB processes Put with throughput rising with batch size, while latency grows sub-linearly in Figure 6(c). This improvement stems from batched writes and asynchronous persistence. With larger batches, each index node is shared by more records, amortizing write complexity (illustrated in Eq. 8) and yielding higher throughput. Asynchronous persistence eliminates disk I/O in updates, further reducing latency. For Get, Figure 6(b) shows that the processing latency of ViDB rises linearly with batch size, and throughput remains relatively stable in Figure 6(d), since point queries are not batched.

Compared with baselines, ViDB shows superior cost efficiency in processing Get and Put APIs. For Put, ViDB achieves throughput gains of 15.2×, 4.5×, 2.0×, 12.4×, 4.4×, and 28.1× over GlassDB, LETUS, QMDB, SQL Ledger, LedgerDB, and QLDB. These differences stem from how systems handle tree updates and persistence.

[2] https://github.com/fishfishfishfishfish/ViDB
[3] https://github.com/bits-and-blooms/bloom
[4] https://github.com/hypermodeinc/ristretto
[5] https://github.com/nusdbsystem/LedgerDatabase
[6] https://github.com/LayerZero-Labs/qmdb
[7] https://github.com/zjuDBSystems/LETUS_prototype

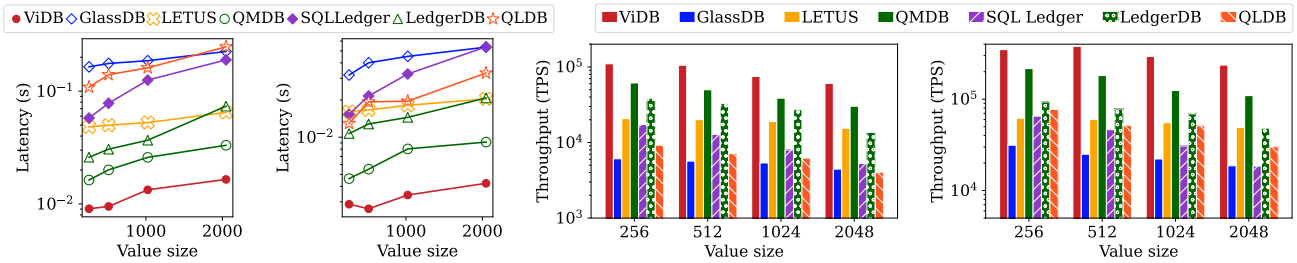
Table 5: Evaluation workload settings

	Micro-benchmark	**Macro-benchmark**	**Real-world system evaluation**
Evaluation target	Performance on specific queries or operations, compared with baselines	End-to-end performance on generated workloads at scale	End-to-end performance on industrial system with real-world workloads
Workloads	Random updates, point queries, range queries (5-2K range size), version operations (up to 40 versions)	Workloads with varying write types, skewness, data scale	Simple payment, EVM transfer
Data volume	1M-10M records	100M-1B records	24M-80M transactions
Logical size	40GB-400GB (1GB-10GB per version, 40 versions)	up to 983 GB (1056 bytes per record)	~4.5GB-15GB (~ 100KB per version, 45K-150K versions)
Distribution	Uniform distribution (Zipfian factor = 0)	Zipfian distribution	Trace-driven distribution



(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.



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

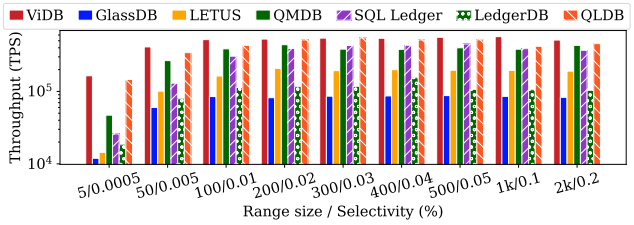
Figure 7: Micro-benchmark results (vs. value size).

QLDB synchronously updates the Merkle tree and persists data, causing low throughput. GlassDB uses asynchronous persistence, but the tree updates are included in the persistence process, so updated data is unavailable until persistence completes, resulting in behavior equivalent to synchronous persistence and throughput similar to QLDB. SQL Ledger separates the tree update from the asynchronous persistence, yielding higher throughput than QLDB and GlassDB. Yet it must update both the Merkle tree and B-tree, lowering throughput relative to LedgerDB and LETUS, which use a unified verifiable index. Based on the unified verifiable index TPBF, ViDB surpasses LedgerDB and LETUS by maintaining one active tree for writes among multiple trees, which reduces the tree height and write complexity (Eq. 8).

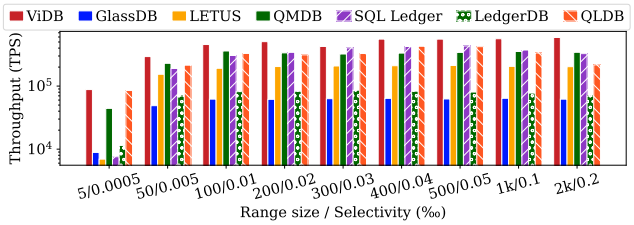
ViDB achieves Get throughput gains of 18.7×, 6.0×, 2.0×, 10.6×, 6.2×, and 428.4× over GlassDB, LETUS, QMDB, SQL Ledger, LedgerDB, and QLDB, primarily due to differences in verifiable index structures. QMDB, QLDB, and SQL Ledger keep a single B-tree index apart from the Merkle tree, which hinders point query performance. QMDB stores the B-tree in memory to enhance query performance; however, without a Merkle tree to ensure integrity, the in-memory index remains vulnerable to tampering. LETUS and LedgerDB use a unified verifiable index (MPT), which slows queries due to long traversal paths. GlassDB uses POS-tree, but its query performance is limited by intensive disk I/O during tree traversal. ViDB excels by using TPBF to combine low height and effective caching, yielding low query complexity as shown in Eq. 2.

We also conduct experiments with value sizes ranging from 256 to 2048 bytes (Figure 7), which demonstrate that ViDB consistently retains performance advantages in both Get and Put operations, independent of value size. The value size analysis shows that both Get and Put throughput decreases as the value size increases. 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 it takes more disk access cost to write a larger value. Across all tested value sizes, ViDB consistently achieves the highest throughput, highlighting the effectiveness of the constant write amplification optimization in its storage layer.

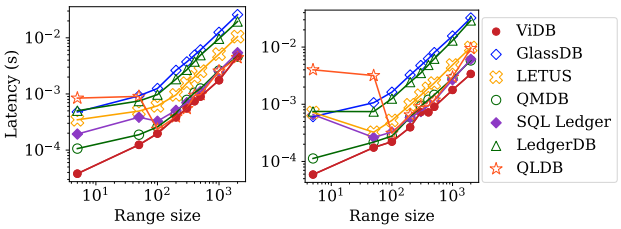
5.2.2 Range query Beyond point queries, we evaluate range queries with varying selectivity. We test range sizes from 5 to 2,000 on datasets with 1M and 10M records, and report throughput in Figure 8. We disable CPU concurrency for a fair comparison, highlighting the benefits of system designs rather than parallelism.



(a) Throughput (1M records)

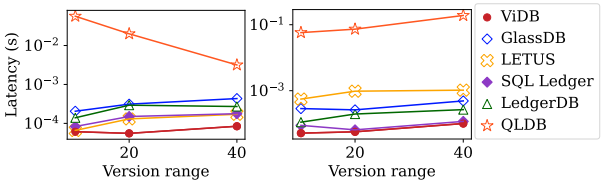


(b) Throughput (10M records)



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

Figure 8: Range query performance.



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

Figure 9: Query historical version.

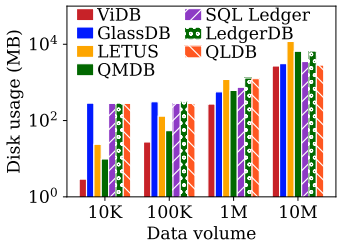


Figure 10: Disk usage.

Higher selectivity yields higher throughput in ViDB as fixed initialization costs ($\Phi \log_m N_{\text{cap}}$ in Eq. 6) are amortized. With 10M records, throughput degrades without concurrency, consistent with the complexity analysis in Eq. 5. ViDB consistently outperforms all baselines in range queries. QLDB, SQL Ledger, and QMDB use B-tree indices, enabling range queries with much higher throughput than point queries. LedgerDB lacks B-tree support, yielding throughput similar to point queries. GlassDB adopts the POS-tree, a B-tree variant enforcing structural invariants. It ensures correctness but incurs recursive calls during range queries, adding traversal overhead. LETUS uses a DMM-Trie, offering moderate acceleration but its tree height limits the performance. ViDB leverages TPBF, combining shallow height with B$^+$-tree efficiency to achieve the best throughput. Even with 10M records (Figure 8(b)), ViDB sustains its advantage.

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. All the tested systems exhibit increasing latencies with the range size, and ViDB demonstrates small latency growth rates with respect to range size, ensuring scalable performance even for large-range queries.

5.2.3 Historical query We evaluate ledger databases on historical version queries, a key feature for audit and compliance. QMDB is excluded as it only keeps the latest version and lacks historical query support. We evaluate historical queries using randomly chosen keys. For each key, we query 10-40 prior versions to simulate realistic audit scenarios.

Figure 9(a) shows that ViDB consistently achieves lower latency across all version depths. Although LETUS and SQL Ledger match ViDB on recent versions (e.g., 10th prior), latency rises sharply for older versions. This degradation arises from their architecture. SQL Ledger, LedgerDB, and QLDB must backtrack from the current to the target version, causing latency to grow with version depth. In contrast, ViDB leverages its Meta table (§3.2) to directly access any historical version, achieving complexity nearly invariant with version depth. LETUS and GlassDB also maintain multiple versions of root nodes, similar to ViDB's Meta table. However, they incur higher latency due to reliance on a single-tree verifiable index, resulting in greater height than TPBF's multi-tree design. ViDB's advantage becomes more pronounced with larger datasets, as it maintains sub-millisecond latency when querying the 40th prior version in the 10M dataset (Figure 9(b)).

5.2.4 Storage cost To demonstrate ViDB's effectiveness in storage consumption, we measure disk usage for 10K-10M records. ViDB substantially reduces disk usage (Figure 10). RocksDB-based systems (GlassDB, QLDB, LedgerDB, SQL Ledger) exhibit constant overhead for small data volumes (10K-100K). LETUS and QMDB manage pages directly to reduce disk usage. ViDB achieves lower storage consumption thanks to the elimination of storage for identifiers. It maintains efficient disk usage for 1M-10M records. In RocksDB-based systems, compression reduces storage for large volumes but degrades get/put performance due to decompression.

For ViDB, we scale the data volume to 1B records, which logically occupies 983 GB of disk space, while ViDB physically uses 1 TB, resulting in a low space amplification of 1.02. As we shows in §5.5.3, the baselines can not scale to 1B data records due to the memory limitation, and we do not include their result in Figure 10.

5.2.5 Version operations We evaluate ViDB's performance on version management operations: pruning and rollback.

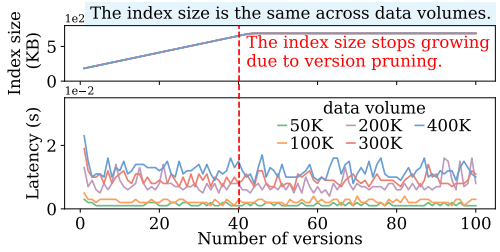


Figure 11: Update latency in each version.

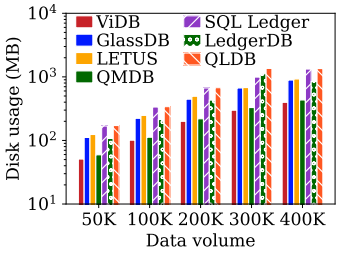


Figure 12: Disk usage for repeated updating.

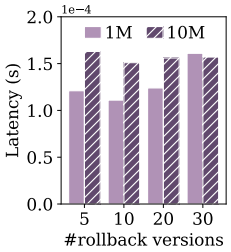


Figure 13: Version rollback.

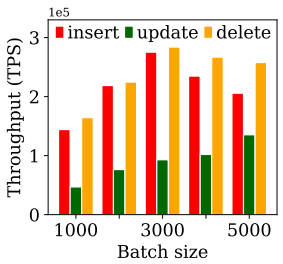


Figure 14: Performance of different types of write operations.

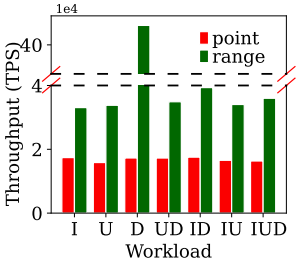
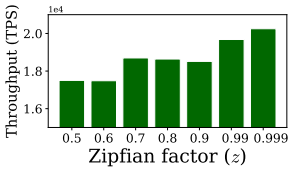
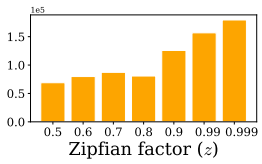


Figure 15: Query performance affected by write operation types.



(a) get throughput



(b) put throughput

Figure 16: Performance under various skewness.

Version pruning. We measure pruning impact by monitoring update latency as the version history grows. We use datasets of 500-4,000 records, each updated up to 100 times to create substantial version histories, resulting in a total data volume of 50K-400K. Pruning triggers automatically at 40 versions and maintains this threshold for subsequent updates.

Figure 11 shows index size (upper part) and update latency (lower part) for each version. After triggering pruning at version 40, the index size stabilizes across data volumes. Update latency remains consistent during pruning, since the LVPS avoids the intensive I/O of traditional garbage collection. Thus, ViDB performs pruning transparently, without noticeable impact on update latency.

Figure 12 compares disk usage under repeated updates across systems with increasing data volumes. ViDB exhibits lower storage overhead than baselines lacking pruning (GlassDB, SQL Ledger, LedgerDB, QLDB). QMDB achieves similar storage savings but sacrifices historical query support. ViDB achieves a balance by providing both historical data access and efficient storage management through efficient version pruning.

Version rollback. Rollback efficiency is evaluated across data volumes and rollback distances of 5-30 versions. Figure 13 shows consistently low rollback latency, with little sensitivity to dataset size or rollback depth. This owes to ViDB's page management that enables rollback performed entirely in memory without disk I/O, enabling rapid recovery for production.

5.3 Macro-benchmark

We use the YCSB benchmark to evaluate ViDB under diverse workloads. We examine performance across write operation types, access skewness, and scaling data volume.

5.3.1 Write operation types In this section, we first measure the performance of insert, update, and delete operations. Then we analyze the impact of write operations on the query performance. Specifically, we test point and range queries under diverse write workloads. We use YCSB to generate 100M writes on a 10M-record dataset with varying ratios of inserts, updates, and deletes. We denote workloads by operation types: I (insert), U (update), D (delete). In each workload, operations are equally distributed. For example, UD has 50% updates and 50% deletes, while IUD has equal shares (33.3%) of each type.

Figure 14 shows the throughput of inserts, updates, and deletes on a dataset of 100M records. The throughput of all three operations increases with batch size, showing that batched optimization in ViDB is effective across operation types. Deletes achieve the highest throughput due to the in-place deletion optimization (§4.2). Inserts achieve higher throughput than updates, since updates often modify more nodes, causing greater write amplification.

Figure 15 shows point and range query throughput for each workload with various write operation types. In most cases, operation types have a negligible impact on query throughput. Exception appears in the workload D dominated by deletes, where the range query throughput is 10 times higher than in other workloads. With N_{cap} =20M, 100M writes usually yield five trees to be traversed during the range query. As in-place deletion (§4.2) prevents multiple tombstones, repeated deletes in workload D neither increase node count nor enlarge the index, yielding a compact single tree. Consequently, in workload D, range queries traverse one tree. However, point queries are unaffected, as they always target one tree.

5.3.2 Access skewness We evaluate ViDB under skewed access patterns. In YCSB, skewness is controlled by the Zipfian parameter $z \in [0, 1]$, with higher z producing more skewed workloads. With k generated by YCSB, we call the Put(k, v) and Get(k) APIs on the dataset with 100M records to measure throughput shown in Figure 16. Get throughput rises with skewness since hot keys are more likely to be cached. Put throughput also rises with z, owing to the batched update design (§4.2). Skewness increases the chance of updating records sharing index nodes, reducing disk I/O. The skip list further eliminates duplicate updates, which occur more frequently under skewed workloads.

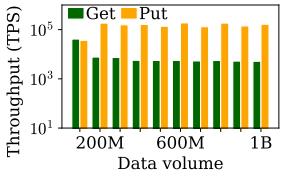


Figure 17: Data scalability.

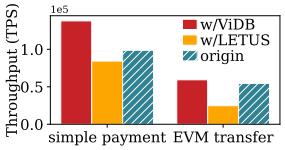


Figure 18: Real System Test.

5.3.3 Data volume We use YCSB to generate datasets up to 1B records to evaluate the scalability of ViDB. We execute Get(k) and Put(k, v) with YCSB-generated keys and measure throughput shown in Figure 17. Beyond 200M records, Get and Put throughput remains stable, confirming that read/write amplification does not vary with data volume, consistent with Eq. 3 and Eq. 9. Get throughput declines between 100M and 200M records because the caches are more effective on smaller datasets. In contrast, Put throughput increases in this range, as denser trees raise index-node sharing, thereby reducing average write complexity.

5.4 Real-world System Evaluation

We demonstrate ViDB's effectiveness when deployed as the storage backend for blockchain systems, representing a real-world scenario for ledger databases. We integrate ViDB into Hyperchain [16], a production blockchain platform, replacing its default storage layer. For a solid comparison, we evaluate three configurations: the original Hyperchain with its native LSM-tree-based storage (origin), Hyperchain with LETUS (w/LETUS), and Hyperchain with ViDB (w/ViDB). They process identical workloads: the Simple Payment and EVM Transfer described in §5.1. This end-to-end evaluation captures the complete performance impact of different storage backends under realistic blockchain transaction processing patterns.

Figure 18 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 performance degradation due to its replay-based page access mechanism, which requires reading multiple physical pages to reconstruct a single logical page. It causes LETUS to perform worse than even the original Hyperchain configuration for both workloads.

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

5.5 Ablation & breakdown studies

In this section, we analyze how configuration affects ViDB's performance. We further break down query processing latencies and evaluate the memory usage.

5.5.1 Partition capacity We analyze how partition capacity (N_{cap}) affects the performance. We fix the dataset at 100M records, vary N_{cap} (60M, 40M, 20M), call the Get(k)/Put(k, v) APIs using random keys, and measure the throughput. Both Get and Put throughput improve with a smaller N_{cap}, as a shorter tree height lowers query complexity (Eq. 3) as well as write complexity (Eq. 8).

5.5.2 Latency breakdown Figure 20 shows the latency breakdowns for point and range queries. The dataset is fixed at 100M records, with N_{cap} yielding 1, 2, 5, or 10 trees. Point query latency comprises Tree (tree traversal), Bloom (Bloom filter testing), and Cache (cache lookup). Range query latency includes Tree (iteration and forward phrase traversal with disk access) and Merge (collect phrase).

Point query latency is reduced with more trees. Tree traversal shortens as tree height decreases. Bloom filter overhead increases slightly with more trees but remains negligible, keeping overall latency reduced.

Multiple trees shorten range query latency more significantly than point query latency. Because multiple trees not only reduce tree height, but also enable concurrent traversal. Merge overhead grows with more trees, but remains limited as it is memory-only. Thus, overall range query latency decreases.

5.5.3 Memory usage We evaluate ViDB's memory usage with up to 1B records. Memory is consumed by seven structures. The Nodes, FreeMap, Meta table, allocating list, and pending list consume negligible memory that remains stable regardless of data volume. Memory usage is mainly dominated by Bloom filters and caches, which ensure low false positive rates and high cache hit ratios. As Figure 22 shows, ViDB maintains about 10GB memory usage with up to 1B records, while baselines exhaust memory beyond 100M or 250M records.

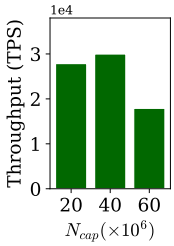
6 Related Work

Verifiable storage. Blockchain systems offer basic verifiable storage but face severe functionality and performance limitations [43]. Even with optimized backend designs [3, 4, 10, 22, 28, 29, 32, 36], decentralization overhead and per-transaction verification still limit blockchain storage performance. In contrast, ledger databases [2, 5, 11, 13, 15, 19, 39, 41–43, 48] ensure immutability and tamper-evidence while offering superior performance over blockchain systems. However, as data grows, existing ledger databases struggle to remain cost-efficient in storage, I/O, and computation.

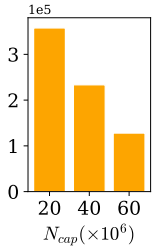
Verifiable index. The verifiable index is critical for ledger databases in data indexing and verification. "Previous optimizations include encoding [35], node caching [10, 29], etc. However, existing designs use a single tree for the entire dataset, leading to growing tree height and degraded performance as the data volume increases. ViDB maintains a forest as the verifiable index, effectively reducing the tree height and improving performance.

Page Management in ledger database. Early ledger databases use LSM-based storage (e.g. LevelDB [12], RocksDB [26]) for verifiable index nodes [2, 37, 43]. Random page identifiers ignore node locality, causing write amplification from intensive background compaction. ChainKV [8] and Block-LSM [9] restore locality by prefixing node identifiers with tree level or version, but assume a specific key format. LETUS [35] reduces write amplification via delta tree encoding, with replay overhead when fetching pages. ViDB designs a storage layer with low write amplification.

Garbage collection. Generic garbage collection [21, 40, 49] used by programming languages and system runtimes has to track the

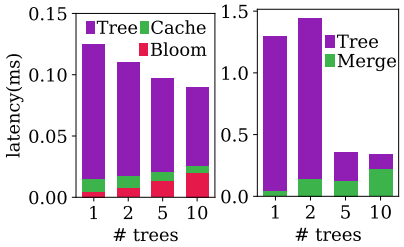


(a) get throughput



(b) put throughput

Figure 19: Effect of partition capacity.



(a) Point query (b) Range query

Figure 20: Latency breakdown.

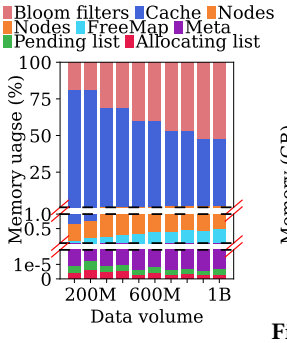


Figure 21: Memory usage.

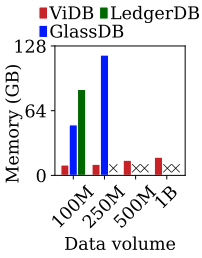


Figure 22: Memory usage comparison

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 [7, 18, 33]. However, the garbage collection process is rarely discussed in ledger databases, even though it is necessary for version operations. ViDB optimizes garbage collection for ledger databases and improves the efficiency of version operations.

7 Conclusion

We present ViDB, a ledger database that maintains low hardware costs while scaling with data volume. It consists of an index layer and a storage layer that reduce the costs of storage, disk I/O, and computation. Based on temporal partitioned forest, the index layer bounds tree height and the complexity of query and write operations. The storage layer minimizes I/O and space amplification and supports in-memory garbage collection. Experiments show that ViDB delivers 2×–20× performance gains and reduces storage by over 70% compared with existing systems. Despite these advantages, it has limitations, such as the high memory overhead of Bloom filters. Future work includes optimizing Bloom filters (e.g., hierarchical designs) and tuning partition capacity and cache size.

References

[1] Anes Abdennebi and Kamer Kaya. 2021. *A Bloom Filter Survey: Variants for Different Domain Applications.* Cornell University. arXiv:2106.12189

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

[3] 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.

[4] 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.

[5] Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason Anderson, and Jakub Szymaszek. 2021. SQL Ledger: Cryptographically Verifiable Data in Azure SQL Database. In *Proceedings of the 2021 International Conference on Management of Data.* ACM, Virtual Event, China, 2437–2449.

[6] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of The ACM* 13, 7 (1970), 422–426.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] R. Koo and S. Toueg. 1987. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering* SE-13, 1 (1987), 23–31.

[21] 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.

[22] 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.

[23] Feifei Li, Marios Hadjieleftheriou, 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.

[24] 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.

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

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

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

[28] 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.

[29] 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.

[30] Ethereum Improvement Proposals. 2015. *ERC-20: Token Standard*. Ethereum Improvement Proposals. https://eips.ethereum.org/EIPS/eip-20

[31] 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.

[32] 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.

[33] 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.

[34] Michael Sober, Markus Levonyak, and Stefan Schulte. 2024. Efficient Cross-Blockchain Token Transfers with Rollback Support. In *2024 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, Shanghai, China, 9–18.

[35] 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.

[36] 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.

[37] 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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.