QMDB: Quick Merkle Database

Isaac Zhang Ryan Zarick Daniel Wong Thomas Kim Bryan Pellegrino
Mignon Li Kelvin Wong

LayerZero Labs

1 Introduction

Updating, managing, and proving world state is the key bottleneck facing the execution layer in modern blockchains. This storage layer has traditionally traded off performance (throughput) and decentralization (capital and infrastructural barriers to participation). Blockchains typically implement active state management using an Authenticated Data Structure (*ADS*) such as a Merkle Patricia Trie (*MPT*). Unfortunately, typical MPT-based ADSes incur a high amount of write amplification (*WA*) with many costly random writes for each state update, which necessitates storing the entire structure in DRAM to avoid getting bottlenecked by the SSD. As a result, the performance and scaling of blockchains is IO-bound, and the key to achieving higher performance with larger datasets is to use disk IOPS more efficiently by reducing WA. In this paper, we present Quick Merkle Database (*QMDB*), a resource efficient SSD-optimized ADS with in-memory Merkleization that implements a superset of the app-level features of existing RocksDB-backed MPT ADSes with 6× throughput on large datasets. QMDB implements state reads with 1 disk read, state updates with O(1) disk IO, and *in-memory* (0 disk reads or writes) Merkleization, all on a DRAM footprint small enough to run on consumer-grade PCs. This constant storage complexity (of DRAM) allows blockchain developers to shift their focus to optimizing the new bottlenecks of the compute and network.

Blockchain state storage is typically handled by an Authenticated Data Structure (*ADS*) that can be thought of as a proof layer (e.g. Merkle Patricia Trie (*MPT*)) in combination with a physical storage layer. The proof layer efficiently generates inclusion and exclusion proofs against the world state, while the physical storage layer stores the actual world state keys and values. In many existing blockchains, these layers are each stored in a separate general-purpose key-value store such as RocksDB, resulting in duplicated data and general inefficiency. This storage of MPT ($O(logN)$ insertion) on a general-purpose key-value store ($O(logN)$ insertion) results in a state update incurring $O(log^2N)$ disk IOs.

QMDB eliminates this inefficiency by unifying the world state and Merkle tree storage, persisting all state updates in an append-only log, and eliminating all disk reads and writes from Merkleization. By grouping updates into fixed-size immutable subtrees called *twigs*, QMDB can Merkleize state updates without reading or writing any world state; this essentially *compresses* the Merkle tree by several orders of magnitude, allowing it to be stored in a modest amount of DRAM.

These optimizations enable QMDB to achieve 6× throughput compared to RocksDB, a general-purpose key-value database that does not perform Merkleization. We also show that QMDB outperforms a prerelease version of NOMT, a state-of-the-art verifiable database, by up to 8×.

2 Background

In this section, we explain the design of other verifiable databases and related data structures, including prior work reducing write amplification of verifiable databases [18, 13].

MPTs combine the efficient proof generation of the Merkle tree with the fast lookups of the Patricia trie and are a common choice for ADS on today's blockchains [22]. In a database of N items, the time complexity of updating a single state entry in an MPT is $O(\log(N))$ [16]. However, MPT and other existing trie-based ADSes suffer from large proofs and a dependency on the validator node having a large amount of physical memory to avoid excessively reading from disk. At the same time, MPTs are not suitable for storage on flash storage, as the randomly distributed update-heavy workload results in high WA. To top it off, the worst-

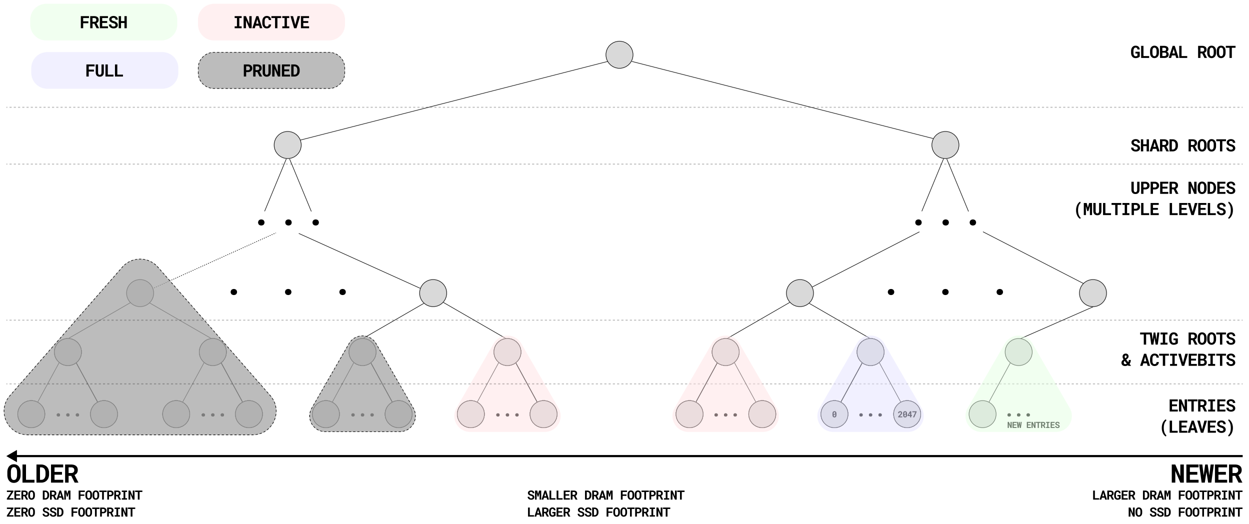

Figure 1: Entries are inserted sequentially into the Fresh twig, which eventually transitions into the Full state. As Entries are deleted, Full twigs become Inactive, then transition to Pruned. Upper nodes are recursively pruned after both of their children are pruned.

case size for inclusion and exclusion proofs can be quite large [14]. These factors result in Merkleization becoming a significant bottleneck that limits the overall throughput of the execution layer and the blockchain.

AVL tree based ADSes have seen popularity as an alternative to MPTs, as they achieve faster updates, lookups, and proof generation due to the self-balancing AVL tree. AVL trees provide a marginal performance increase over MPTs in the average case, but still suffer from O(log N) tree nodes modifications per state update.

NOMT is a promising state-of-the-art ADS that uses a contiguous disk layout for a binary Merkle tree with compressed metadata, overcoming some limitations of existing MPT-based ADS implementations. This design results in better performance than existing solutions and has garnered interest in the space. NOMT implements an array of improvements including tree arity, disk native layout, and caching. Similarly to AVL tree-based ADSes, NOMT is fundamentally an incremental optimization in place of various components and hyperparameters of existing ADS deployments.

MoltDB is similar to NOMT in that it is an incremental low-level improvement on the existing two-layer MPT design. MoltDB segregates states by recency and introduces a compaction process that increases throughput by 30% [15] by reducing disk IO.

Merkle Mountain Range (MMR) [21] enable compact inclusion proofs and are append-only, which makes the IO pattern for updating state conducive to efficient usage of SSD IOPS. Each MMR is a list of Merkle subtrees (*peaks*), and peaks of equal size are merged as new records are appended.

MMRs are not suitable for live state management, as they cannot natively handle deletes, updates, lookups by key, and exclusion proof generation. As a result, MMRs have generally found success in their use for historical data management [17] where the key is just an index.

Acceleration of Merkle tree computation has been an area of active research, with several proposed techniques such as caching [8, 5], optimizing subtrees [4], and using specialized hardware [12, 6]. These improvements are orthogonal to QMDB and could be applied to QMDB to further improve its performance and efficiency.

Verifiable ledger databases are systems that allow users to verify that a log is indeed append-only, of which blockchains are a subset. A common approach to implementing a verifiable ledger database is deferred verification [24, 23, 3]. GlassDB [24] uses a POS-tree (a Merkle tree variant) as an ADS for efficient proofs. Amazon's QLDB [2], Azure's SQLLedger [3], and Alibaba's LedgerDB [23] are commercially available verifiable databases that use Merkle trees (or variants) internally to provide transparency logs. VeritasDB [20] uses trusted hardware (SGX) to aid verification. The key difference between these databases and QMDB is that QMDB is optimized for frequent state updates and real-time verification of the current state (as opposed to verification of historical logs and deferred verification).

Field	Description	Purpose
Id	Unique identifier (e.g., nonce)	Prove key inclusion
Key	Hash of the key	Identify the key
Value	Current state value the key	Serve application logic
NextKey	Lexicographic successor of \text{Key}	Prove key exclusion
OldId	Id of the Entry previously containing \text{Key}	Prove historical inclusion / exclusion
OldNextKeyId	Id of the Entry previously containing \text{NextKey}	Prove key deletion
Version	Block height and transaction index	Query state by block height

Table 1: The fields in a QMDB entry. ID and Version are 8 bytes, Key is 32 bytes, and Value can hold up to 2^{24} bytes.

3 QMDB Design

QMDB is architected as a binary Merkle tree illustrated in Figure 1. At the top is a single *global root* that connects a set of *shard roots*, each of which represents the subtree of the world state that is managed by an independent QMDB shard. The shard root itself is connected to a set of *upper nodes*, which, in turn, are connected to fixed-size subtrees called *twigs*; each of these twigs has a root that stores the Merkle hash of the subtree and a bitmap called ActiveBits to track which entries are part of the most current world state. Entries (the twig's leaves) are immutable, making it unnecessary to read or write the entry root during Merkleization; this results in QMDB only ever reading/writing the global root, shard roots, upper nodes, and twig roots during Merkleization. The twig essentially compresses the actual state keys and values into a single hash and bitmap, making the data required for Merkleization small enough to fit in a small amount of DRAM rather than being stored on disk.

In this section we begin by explaining the underlying storage primitives used to organize state data (Section 3.1), followed by a discussion of the indexer in Section 3.2. In Section 3.3 we describe the high-level CRUD interface exported by QMDB to clients. In Section 3.4 we describe how the storage backend and indexer facilitate generation of state proofs, and discuss how these state proofs can be statelessly validated. Finally, in Section 3.5 we explain how QMDB takes advantage of additional optimizations such as sharding and pipelining to scale throughput via improved parallelism.

3.1 Entries and Twigs

The **entry** (Table 1) is the primitive data structure in QMDB, encapsulating key-value pairs with the metadata required for efficient proof generation. Entries can be extended to support features such as historical state proof generation (Section 3.4). QMDB keys entries by the *hash* of the application-level key, resulting in improved load balancing via uniform key distribution across shards (Section 3.5)

State	Description	Entries	Twig Root
Fresh	Entries ≤ 2047	DRAM	DRAM
Full	2048 Entries	SSD	DRAM
Inactive	0 active Entries	Deleted	SSD
Pruned	Subtree deleted	Deleted	Deleted

Table 2: As twigs progress through their lifecycle, their footprint in DRAM gets smaller. An inactive twig has 99.9% smaller memory footprint than a full twig.

Twigs are subtrees within QMDB's Merkle Tree; each twig has a fixed depth, by extension a fixed number of entries stored in the leaf nodes (2048 in our implementation). A set of *upper nodes* connects all twigs to a single shard root, with null nodes to represent uninitialized values; these upper nodes are immutable once all their descendant entries are initialized. In addition to the actual Merkle subtree, Twigs also store the Merkle hash of their root node and *ActiveBits*, a bitmap that describes whether each contained entry contains state that has not been overwritten or deleted. The twig essentially compresses the information required to Merkleize 2048 entries and their upper nodes ($\geq 256kb$) into a single 32-byte hash and a 256-byte bitmap (99.9% compression). This compression is the key to enabling fully in-memory Merkleization in QMDB.

Fresh twigs reside completely in DRAM, and entries are sequentially inserted into its leaf nodes. Once a twig reaches 2048 entries, its contents are asynchronously flushed to SSD in a large sequential write and deleted from DRAM, maximizing SSD utilization and minimizing DRAM footprint.

Each twig follows a lifecycle of 4 states: Fresh, Full, Inactive, and Pruned (Table 2). An example of the layout of QMDB's state tree is presented in Figure 1 There is exactly one fresh twig per shard, and entries are always appended to the fresh twig. After all entries in the twig are outdated as a result of update and delete operations, the twig transitions into the inactive state before eventually being pruned and replaced by the Merkle hash of the root. Upper nodes that contain only pruned twigs are recursively pruned, further reducing the memory footprint of QMDB.

The grouping of entries into twigs reduces the DRAM footprint of QMDB to the degree that *all nodes* affected by Merkleization can be stored in a small amount of DRAM. In a hypothetical scenario with 2^{30} entries (approx. 1 billion), the system must keep at most 2^{19} 288-byte (32-byte twig root hash & 2048-bit activeBits bitmap) full twigs, 1 fresh twig and $2^{19}-1$ 32-byte (node hash) upper nodes totaling around 160 megabytes. In practice, the majority of the 2^{19} twigs will be pruned, resulting in the average size being much smaller.

Inactive and Pruned twigs can never be modified, and thus never require further Merkleization. Fresh and Full twigs must be Merkleized every time the activeBits bitmap is changed, and Fresh twigs must additionally be Merkleized every time an entry is added. The upper nodes of the Merkle tree are recomputed on startup and are never persisted to disk–this recomputation requires reading all twig hashes from SSD and performing 2 hashes per twig, and can be completed in a matter of milliseconds for the previous example of 1 billion entries.

QMDB stores an entry every time state is modified, making the state tree grow proportionally to the number of state modifications. To combat this tree growth a dedicated compressor worker periodically *compresses* QMDB's state tree by re-appending old entries to the fresh twig, thus accelerating the progression of the twig lifecycle and allowing more subtrees to be pruned. However, the Merkle proof size and proof generation complexity grows proportionally to $log2$ of the number of state *updates* rather than the number of unique keys. This grows very slowly, and the order-of-magnitude improvement in Merkleization performance outweigh this minor cost. In addition, ZK-proofs can be used to compress the proof witness data size and dramatically reduce proof verification cost, thus eliminating any end-to-end bottleneck regarding the proof size.

3.2 Indexer

The *indexer* maps the application-level keys to their respective entries, enabling QMDB's CRUD interface. To support efficient creation and deletion of entries (Section 3.3), the indexer must support ordered key iteration. The indexer can be freely swapped for different implementations depending on specific application needs, but we expect that QMDB's default in-memory indexer will meet the resource requirements of the majority of use cases.

QMDB's default indexer consumes about 15.4 bytes of DRAM per key and serves key lookups in-memory to reduce disk I/Os; this is achieved by keeping only the 9 most significant bytes of the key, allowing a slightly higher probability of key collision and trading worst-case performance for reduced DRAM usage. Using just 8 gigabytes of DRAM, the in-memory indexer can index more than 1 billion entries, making it suitable for a wide range of applications. We chose the B-tree map as the basis for the underlying structure of QMDB's default indexer to take advantage of B-tree's high cache locality, low memory overhead, support for ordered key iteration, and graceful handling of key collisions. We use fine-grained reader-writer locks (determined by the first two bytes of the key hash) to minimize contention when updating entries.

3.3 CRUD interface

QMDB exposes a CRUD (Create, Read, Update, Delete) interface, and in this section we provide a high-level overview of how each operation is implemented. In all examples, we present the operation of the system when using the default in-memory indexer; other indexers may require more reads or writes to serve the same workload. For each operation, we present an intuitive explanation followed by a more formal description along with a description of the disk IO required to synchronously handle the request. All writes in QMDB are buffered in twigs (DRAM) and persisted to SSD in batches, so each disk write is amortized across 2048 entries; to precisely express the cost of each operation, we refer to an *entry write* as $\frac{1}{2048}$ of a single batched flush to SSD. For brevity, we omit the Id, Version, and Value fields when describing new entries, so an entry E is defined as:

$$E = (Key,\ NextKey,\ OldId,\ OldNextKeyId)$$

Read begins by querying the indexer for the file offset of the entry corresponding to a given key; this file offset is used to read the entry in a single disk IO.

Update first *reads* the most current entry for the updated key, then appends a new entry to the fresh twig. More formally, if E is the most current entry, the new entry E' appended to the fresh twig derives its OldId and OldNextKeyId from E as follows:

$$E' = (K,\ E.nextKey,\ E.Id,\ E.OldNextKeyId)$$

Updating a key in QMDB incurs 1 disk read and 1 *entry write*.

Create intuitively involves appending one new entry and *updating* one existing entry; the existing entry whose Key and NextKey define a range that coincides with the created key must be updated with a new NextKey.

This begins by first *reading* the entry E_p corresponding to the lexicographic predecessor (*prevKey*) to the created key K. Note that E_p must fulfill the condition $E_p.Key < K < E_p.nextKey$, as K is not yet part of the current state. Then, two new Entries are appended to the fresh Twig:

$$E_K = (K,\ E_p.nextKey,\ E_p.Id,\ E_n.Id)$$

$$E'_p = (prevKey,\ K,\ E_p.Id,\ E_n.OldId)$$

The activeBit of E_p is set to false (in memory), and the indexer is updated so that *prevKey* points to the file offset of E'_p and K points to the file offset of E_K.

Creating a key in QMDB incurs 1 disk read and 2 *entry writes*.

Delete is implemented by first setting the activeBit to false for the most current entry corresponding to K, then *updating* the entry for *prevKey*. First, the entry E_K and E_p corresponding to keys K and *prevKey* are read from SSD, and the activeBits for the twig containing E_K is updated. Next, a new entry for prevKey is appended to the fresh twig:

$$E'_p = (prevKey,\ E_k.nextKey,\ E_p.Id,\ E_K.OldNextKeyId)$$

Deleting a key in QMDB incurs 2 disk reads and 1 *entry write*.

3.4 Proofs

The remainder of this section describes how each field of the QMDB entry enables the generation of various state proofs. For illustrative purposes, we present proofs of the state corresponding to a key K and the most current Merkle root R, and denote fields of an entry E as $E.fieldName$. All proofs are Merkle proofs and as a result can be statelessly verified.

Inclusion is proved by presenting the Merkle proof π for entry E such that $E.Key = K$; this entry E can be obtained after querying the corresponding file offset from the indexer.

Exclusion is proved by presenting the inclusion proof of E such that $E.Key < K < E.nextKey$. The QMDB indexer supports efficient iteration by key, so E can be located quickly by querying the lexicographic predecessor to K.

Historical inclusion and exclusion at block height H can be proven for a key K by providing the inclusion proof of an entry such that K is represented by this entry at the given version (block height).

QMDB uses OldId and OldNextKeyId to form a graph that enables tracing of keys over time and space through updates, deletions, and creations. OldId links the current entry to the last inactive entry with the same key and OldNextKeyId links to the entry previously referenced by NextKey (when the entry for NextKey is deleted). When proving historical inclusion or exclusion, QMDB traverses the OldId pointer to move backwards in "time", and the NextKey and OldNextKeyId pointers to move to different parts of the keyspace at a given block height.

Reconstruction of historical state The graph structure defined by OldId and OldNextKeyId can also be used to reconstruct the Merkle tree and world state at any block height The Version field tracks the block height and transaction index where the entry was created, enabling precise reconstruction of historical states at specific block heights.

3.5 Parallelization

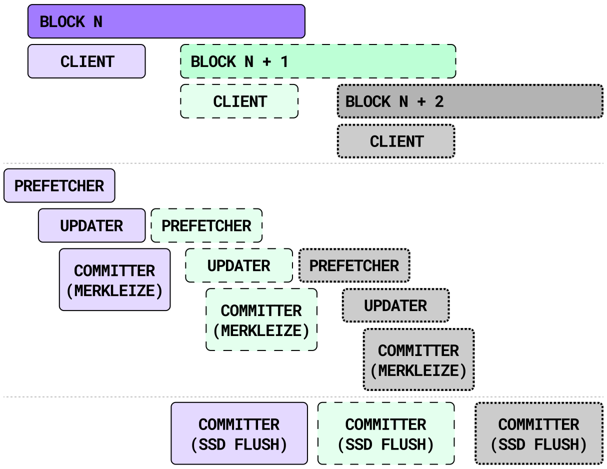

Figure 2: QMDB prefetches data (*prefetcher*), performs the state transition (*updater*), then commits the updated state to the Merkle tree and persistent storage (*committer*).

State updates are parallelized in QMDB through sharding and pipelining.

QMDB shards its keyspace into contiguous spans with *sentry nodes* to define logical boundaries that prevent state modifications from crossing shard boundaries (i.e., prevKey and nextKey will always fall within the same shard). This sharding enables QMDB to better saturate underlying hardware resources and scale to bigger or multiple physical servers.

In addition, QMDB implements a three-stage pipeline (Prefetch-Update-Flush) to allow the transaction processing layer to better saturate QMDB itself. For applications with relaxed synchronicity for state updates, QMDB is able to interleave computation across overlapping blocks. This cross-block and intrablock parallelism allows QMDB to more fully saturate available CPU cycles and SSD IOPS.

Clients interact with QMDB by enqueueing key-value CRUD requests; updates are requested by writing the old Entry and new Value into the EntryCache directly, while Deletions and Creations only require the key and new entry respectively.

The pipeline is illustrated in Figure 2, and is managed by three workers: the fetcher, updater, and committer. Each stage is shown in rectangles with solid lines, and the workers communicate via producer-consumer task queues in shared memory. The fetcher reads relevant entries from SSD into the EntryCache in DRAM when necessary (Deletion and Creation), while the updater appends new entries and updates the indexer. Once the fetcher and updater finish processing a block, the committer asynchronously Merkleizes the updates and flushes the full twigs to persistent storage.

The QMDB pipeline has *N+1 serializability*, which guarantees state updates are visible in the next block. This is implemented by enforcing that the prefetcher cannot run for block N until the updater finishes processing block $N-1$.

4 Evaluation

In this section, we present a preliminary evaluation of the performance of QMDB and compare it to RocksDB and NOMT. On a comparable workload, QMDB achieves 6× higher updates per second than RocksDB and up to 8× higher than NOMT.

4.1 6X more updates/s than KV DBs

Figure 3 shows the throughput of QMDB compared to RocksDb, demonstrating that QMDB delivers 6× more updates per second than RocksDB. This speedup is in fact an underestimate of QMDB's advantage over two-layer systems, given that all benchmarks compare QMDB with Merkleization to RocksDB without Merkleization. The primary factor driving this speedup is QMDB's single-layer design with no in-place overwriting of persistent data, essentially trading off unused functionality for extra throughput. Some examples of features and characteristics of RocksDB that are not present in QMDB are efficient range/prefix queries and spatial locality of key-value pairs.

We caveat that our RocksDB evaluation is preliminary and could be better optimized, as our results were gathered on an unsharded RocksDB instance with default parameters. We also tested RocksDB with the parameters recommended by the RocksDB wiki [9] with direct I/O enabled for reads and compaction, but did not observe noticeably better performance. We have also informally tested with MDBX but do not show those results here, as MDBX was significantly slower than RocksDB.

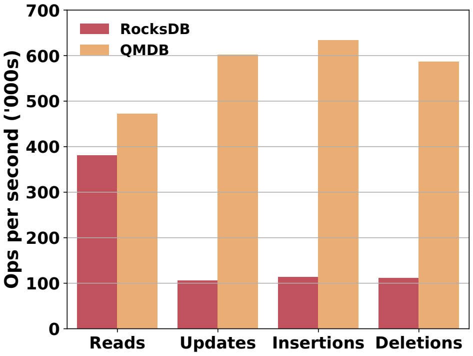

Figure 3: **QMDB shows a 6× increase in throughput over RocksDB.** QMDB is able to do 601K updates/sec with 6 billion entries and demonstrates superior performance across all operation types. These results were obtained on an AWS instance with 2 SSDs and 64 vCPUs.

4.2 Up to 8X throughput vs state-of-the-art

For a similar comparison to a verifiable database that also performs Merkleization, we compared QMDB to NOMT [10]. NOMT performs Merkleization and stores Merkleized state directly on SSD, and can be directly compared to QMDB in terms of functionality. Both QMDB and NOMT aim to be drop-in replacements for general-purpose key-value stores like RocksDB, and aim to leverage the performance of NVMe SSDs.

At the time of writing, both QMDB and NOMT are pre-release, making it impossible to exactly match the configuration between the two systems in our evaluation. We believe that we have taken sufficient steps to present a fair comparison: we evaluated QMDB and NOMT using their respective benchmark utilities, verified the NOMT parameters with the authors [1], used the same hardware when evaluating each system, and normalized the performance results against the workload. Unfortunately, we were unable to eliminate *all* variability, as NOMT does not support client-level pipelining and QMDB does not support direct IO or `io_uring` (preliminary results for QMDB with `io_uring` and direct IO are shown in §4.3).

Table 3 shows the results of our evaluation, demonstrating up to an 8× speedup in *normalized updates per second* (transaction count multiplied by state updates per transaction). NOMT's default workload is a 2-read-2-write transaction, whereas QMDB is evaluated with a 9-write-15-read-1-create-1-delete transaction (based on the operation composition of historical Ethereum transactions). By normalizing the results based on the workload, we provide what we believe to be a fair representation of the comparative performance of these two systems. Read latency was comparable (30.7μs for QMDB and 55.9μs for NOMT) and close to the i3en.metal SSD read latency, which is to be expected for both systems.

This performance gap is primarily driven by SSD write amplification, which is caused primarily the fact that NOMT buffers updates in a write-ahead log whereas QMDB uses completely immutable entries. This results in persistent storage writes for state update and Merkleization for NOMT, compared to QMDB where each update incurs $\frac{1}{2048}$ disk writes for the state update and zero disk accesses for Merkleization.

Normalized updates per second			
# Keys (M)	**QMDB**	**NOMT**	**Speedup**
4 (2^{22})	614,948	162,190	4×
256 (2^{28})	346,843	42,277	8×
4096 (2^{32})	294,349	37,057	8×

Table 3: **QMDB is up to 8× faster than NOMT.** Results are normalized by multiplying the transactions per second by the number of state updates per second.

4.3 Reaching 2M updates per second

We show preliminary results indicating that QMDB can double its throughput by incorporating asynchronous I/O (`io_uring`) and direct I/O (`O_DIRECT`), improving CPU efficiency and eliminating VFS-related overhead respectively.

Continuous advancements in SSD performance has resulted in even modern *consumer-grade* SSDs (e.g., Crucial T705, Samsung 980 [11]) can reach over 1 million IOPS. These high-IOPS SSDs are not yet available on AWS, so we approximate the performance in our preliminary experiments by using RAID0.

After populating QMDB with 14 billion entries, we measured 2.28 million updates/second on i8g.metal-24xl (6 SSDs) and 697 thousand updates/second on i8g.8xlarge (2 SSDs), which are very promising early results. 2.28 million updates is sufficient to support over one million native token transfers per second (each transfer requiring two state updates). QMDB's CPU utilization averages 77% on the 32-core AWS i8g.8xlarge instance and 58% on the 96-core AWS i8g.metal-24xl instance, indicating that with faster SSDs the bottleneck is no longer SSD IO but rather CPU and synchronization overheads.

We also evaluated NOMT with a lower capacity of 1 billion entries on the same instances (i8g.metal-24xl and i8g.8xlarge), and observed a maximum of 60,831 updates/second. We acknowledge that comparing these numbers would not be fair given that NOMT is focused on supporting single-drive deployments, and RAID0 has different performance characteristics than a single SSD. We plan to perform a more comprehensive evaluation with a single high-performance SSD once we are able to secure a testbed with the necessary hardware.

4.4 Scaling up *and* down

QMDB scales *up* to huge datasets *and down* to ultra-low minimum system requirements, enabling it to meet the needs of both performance-oriented and highly decentralized blockchains.

Scaling up to hundreds of billions of entries. The hybrid indexer trades off SSD capacity and system throughput to reduce the DRAM footprint of the QMDB indexing layer to just 2–3 bytes per entry, allowing servers with a high ratio of SSD capacity to DRAM capacity to scale to huge world states. Table 4.4 shows the theoretical maximum number of entries that can be stored in QMDB running on various different AWS instances. We demonstrate that the i3en.metal instance with high SSD capacity and a reasonable amount of DRAM scales up to *280 billion* entries, far exceeding the needs of any existing production blockchain. Due to the prohibitive amount of time necessary to populate hundreds or even tens of billions of keys, we only run experiments up to 15 billion entries and conservatively extrapolate the results. The marginal DRAM overhead actually drops as more entries are inserted; 1 billion entries costs about 3 bytes of DRAM per entry, which drops to just 2.2 bytes per entry for 15 billion entries.

Scaling down to consumer-grade budget servers. We built a low-cost Mini PC (parts totaling about US$540 as of November 2024) to test QMDB under resource-constrained conditions. The system featured an AMD R7-5825U (8C/16T) processor, 64 GiB DDR4 DRAM, and a TiPro7100 4 TB NVMe SSD rated at approximately 330K IOPS. Despite these modest specs, QMDB achieved tens of thousands of operations per second with billions of entries. Using the in-memory indexer configuration, we were able to achieve 150,000 updates per second up to 1 billion entries, and stayed above 100,000 updates per second as we inserted up to 4 billion entries. With the hybrid indexer, QMDB maintained 63,000 updates per second storing 15 billion entries. These results highlight QMDB's ability to operate on commodity hardware, improving decentralization by lowering the capital requirements and infrastructural barriers blockchain participation.

Instance type	DRAM (GiB)	SSD (TB)	Maximum entries (billions) In-Memory	 Hybrid	Factor
c7gd.metal	128	3.8	9.2	18	1.9
m7gd.metal	256	3.8	18.3	18	1.0
i3.metal	512	15.2	36.7	71	1.9
i8g.metal-24xl	768	22.5	55.0	105	1.9
i4i.metal	1024	30	73.3	140	1.9
i3en.metal	768	60	55.0	280	5.1

Table 4: **QMDB can scale to hundreds of billions of entries.** The hybrid indexer uses only 2–3 bytes of DRAM per entry. This table shows extrapolated theoretical world state sizes for different hardware configurations, and compares the maximum entries stored using the in-memory indexer vs the hybrid indexer.

5 Discussion

Spatial locality is reduced in QMDB compared to systems such as RocksDB. Keys that were stored close together in QMDB can be separated if one is updated at a different time than the other. However, most blockchains implement measures to uniformly distribute keys across storage with some exceptions (e.g., arrays in EVM); this reduces or eliminates spatial locality. Unlike traditional computing workloads, blockchains cannot rely on average improvements in performance due to locality, as any such reliance exposes the blockchain to denial-of-service attacks in a Byzantine fault model.

Provable historical state enables new applications such as a TWAP price aggregation at the tip of the blockchain with arbitrary time granularity.

Peer-to-peer syncing of state can be easily and efficiently implemented by sharing state on twig granularity. A downloaded twig accompanied by the inclusion proof of this twig against the global Merkle root can be independently inserted into the state tree opportunistically and on demand.

Memory-efficient indexers may be important for heavily resource-constrained use cases or when trying to decentralize blockchains with tens of billions of keys. We implemented a memory-efficient SSD-optimized *hybrid indexer* that uses only 2.3 bytes per key but requires one disk read per lookup. The hybrid indexer stores key-to-file offset mappings in immutable SSD-resident log-structured files and implements an overlay layer to manage entries in the SSD that have gone stale due to updates. In addition, the hybrid indexer uses a DRAM caching layer to take advantage of the spatial and temporal location of the application workload.

State bloat is one of the many problems plaguing modern blockchains–as blockchains see growth in widespread adoption, world state is continuously growing to the point that it limits the ability of non-professional users to adequately run the validator software. QMDB achieves a memory footprint that is an order of magnitude smaller than existing verifiable databases, and using the hybrid indexer can further reduce the memory footprint and reduce barriers to validator participation.

Recovery after failure (crash, reorg) is currently implemented via replaying up to a checkpoint before trimming inactive entries. We purposely did not build any specific optimizations for recovering after consensus reorg into QMDB as no one-size-fits-all solution can serve the widely varying characteristics of each chain's consensus protocol. QMDB can be extended to support quick switches with an undo log, but in general we expect applications to build a buffering layer on top of QMDB and only write finalized data to QMDB.

Trusted Execution Environments (*TEE*s) offer several security advantages to blockchains, and to the best of our knowledge QMDB is the first TEE-ready verifiable database. Running a blockchain full node in a TEE (e.g., Intel SGX) protects the validator's private key from leaking, provides a secure endorsement that the state root was generated by a particular binary, guarantees peers that the validator is non-byzantine, and prevents censorship.

Current TEEs protect the integrity of CPU and DRAM, but cannot fully isolate persistent storage resources; QMDB protects its persistently stored data via AES-GCM [7] encryption using keys dynamically derived from the virtual file offset to protect against copy attacks. This design complicates recovery after reorg without compromising security, making it better overall to implement reorg protection in a layer above QMDB and only submit finalized data to QMDB.

Zero-knowledge (ZK) proof generation is growing to be increasingly important, yet the prohibitively long time required to generate ZK proofs of state transitions is one of the primary factors limiting the widespread adoption of ZK technology. The generation of ZK proofs can be parallelized per state commitment [19] (e.g., each block can be proven individually and then chained together); thus, the degree of parallelization depends on the frequency of state root generation. QMDB's high performance in-memory Merkleization is capable of computing a new state root *per-transaction* if desired, enabling the maximum degree of parallelism for ZK proof generation.

6 Conclusion

QMDB represents a significant leap in blockchain state databases, providing an order of magnitude improvement in throughput over state-of-the-art systems in datasets 10× than Ethereum at the time of writing. Organizing and compressing state updates into append-only *twigs*, QMDB is able to update and Merkleize world state with near-zero write amplification, improving performance and reducing cost through efficient utilization of SSD IOPS. The immutability of non-fresh twigs allows state to be compressed by more than 99.9% for Merkleization, making it the first live-state management system capable of performing fully in-memory Merkleization with *zero* disk IO on a consumer grade machine.

With these architectural innovations, we show that QMDB can achieve up to 2 million updates per second and scale to world states of 15 billion keys. QMDB achieves lower minimum hardware requirements for all throughput benchmarks and world state sizes, democratizing blockchain networks by enabling affordable home-grade setups (US$540) to participate in large blockchains. At the same time, it provides substantial cost savings for large-scale operators due to its flash-heavy design that eliminates the need for large amounts of expensive and power-hungry DRAM.

QMDB also implements many new features not seen in other ADSes, such as historical state proofs, opening opportunities for a new class of applications not seen on blockchain before. These features, together with order-of-magnitude advancements in performance and efficiency, establish QMDB as a solid foundation for scalable and verifiable databases.

7 Acknowledgments

The authors thank the various teams and individuals who have provided invaluable feedback and helped us to refine our system; in particular, Patrick O'Grady from Commonware for lending us his expertise and knowledge throughout the development of QMDB and the writing of this paper.

References

[1] Reproducing benchmark numbers. https://github.com/thrumdev/nomt/issues/611.

[2] Amazon Web Services. Amazon Quantum Ledger Database (QLDB), 2019.

[3] Antonopoulos, P., Kaushik, R., Kodavalla, H., Rosales Aceves, S., Wong, R., Anderson, J., and Szymaszek, J. Sql ledger: Cryptographically verifiable data in azure sql database. In *Proceedings of the 2021 international conference on management of data* (2021), pp. 2437–2449.

[4] Ayyalasomayajula, P., and Ramkumar, M. Optimization of merkle tree structures: A focus on subtree implementation. In *2023 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)* (2023), IEEE, pp. 59–67.

[5] Dahlberg, R., Pulls, T., and Peeters, R. Efficient sparse merkle trees: Caching strategies and secure (non-) membership proofs. In *Secure IT Systems: 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016. Proceedings 21* (2016), Springer, pp. 199–215.

[6] Deng, Y., Yan, M., and Tang, B. Accelerating merkle patricia trie with gpu. *Proceedings of the VLDB Endowment 17*, 8 (2024), 1856–1869.

[7] Dworkin, M. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Special Publication 800-38D, NIST, 2007.

[8] El-Hindi, M., Ziegler, T., and Binnig, C. Towards merkle trees for high-performance data systems. In *Proceedings of the 1st Workshop on Verifiable Database Systems* (2023), pp. 28–33.

[9] Facebook. Setup options and basic tuning - rocksdb wiki. https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning, 2024. Accessed: 2024-12-21.

[10] Habermeier, R. Introducing nomt. Blog post, May 2024.

[11] Habermeier, R. Nomt: Scaling blockchains with a high-throughput state database. Presented at sub0 reset 2024, November 2024.

[12] Jeon, K., Lee, J., Kim, B., and Kim, J. J. Hardware accelerated reusable merkle tree generation for bitcoin blockchain headers. *IEEE Computer Architecture Letters* (2023).

[13] Li, C., Beillahi, S. M., Yang, G., Wu, M., Xu, W., and Long, F. Lvmt: An efficient authenticated storage for blockchain. *ACM Transactions on Storage 20*, 3 (2024), 1–34.

[14] Li, Y. Understanding ethereum execution layer performance. Presentation at SevenX Research Day: Restaked Modularity @ETHDenver, 2024. Slides available at https://docs.google.com/presentation/d/e/2PACX-1vSe0F9Qhq09n624T5D05weyoxJ81UaM9d02c1tvL8uQtPRVUPdlq21nZzMvhi0psH pub?start=false&loop=false&delayms=3000&slide=id.p5.

[15] Liang, J., Chen, W., Hong, Z., Zhu, H., Qiu, W., and Zheng, Z. Moltdb: Accelerating blockchain via ancient state segregation. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[16] Paradigm. Reth: A modular and high-performance ethereum execution layer client. https://github.com/paradigmxyz/reth, 2022. Accessed: 2024-11-25.

[17] Protocol, H. Merkle mountain ranges: Historical block hash accumulator. https://docs.herodotus.dev/herodotus-docs/protocol-design/historical-block-hash-accumulator/merkle-mountain-ranges. Accessed: 2024-11-18.

[18] Raju, P., Ponnapalli, S., Kaminsky, E., Oved, G., Keener, Z., Chidambaram, V., and Abraham, I. {mLSM}: Making authenticated storage faster in ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (2018).

[19] Roy, U. Introducing SP1: A performant, 100% open-source, contributor-friendly zkVM, 2024. Retrieved on December 20, 2024.

[20] Sinha, R., and Christodorescu, M. Veritasdb: High throughput key-value store with integrity. *Cryptology ePrint Archive* (2018).

[21] Todd, P. Merkle mountain ranges. https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md, 2016. Accessed: 2024-11-18.

[22] Wood, G. Ethereum: A secure decentralized generalized transaction ledger. In *Ethereum Yellow Paper* (2014).

[23] Yang, X., Zhang, Y., Wang, S., Yu, B., Li, F., Li, Y., and Yan, W. Ledgerdb: A centralized ledger database for universal audit and verification. *Proceedings of the VLDB Endowment 13*, 12 (2020), 3138–3151.

[24] Yue, C., Dinh, T. T. A., Xie, Z., Zhang, M., Chen, G., Ooi, B. C., and Xiao, X. Glassdb: An efficient verifiable ledger database system through transparency. *arXiv preprint arXiv:2207.00944* (2022).