At their core, blockchains are software programs that replicate the state of a network across many independent computers which may not trust each other. Each time the state advances (via appending new transactions to the blockchain), nodes must verify every transaction against the current state and then apply the resulting state changes. If I attempt to spend 1eth, participants on the network must confirm that my account contains at least that balance by producing a proof & verifying a proof of that part of the state. Each network node's ability to store state, serve proofs of this state, and verify the state in a timely manner (known as executing "Performant Data Authentication") creates limitations on the network's scaling capacity.

"Decentralized systems require the ability to prove authenticity of data received from potentially untrustworthy sources." --Roberto blog post

For the same standard of hardware, a node that performs data authentication in memory, i.e. does fewer SSD lookups, can scale to execute far more transactions than a node that does not. How can nodes take advantage of speedy memory accesses and reduce SSD lookups? They can either buy expensive machines with more memory or use efficient data structures that make authentication cheaper. At the moment, the standard data structures for data authentication used by most blockchains are variations of Merkle Trees, for example Ethereum's Merkle-Patricia-Tries. But as you will see, these have several inefficiencies which motivates a search for better options.

Through some clever innovations and modifications to existing merkle tree variants, LayerZero created the Quick Merkle Database (QMDB) which significantly improves the performance of data authentication in comparison to existing data structures. The high level advancement with QMDB is that the sections of the data structure relevant to new updates/writes are compacted into a smaller contiguous space which can be held mostly in memory. This allows us to do speedy writes to QMDB's, whereas writes to other variants of merkle trees incur a large cost in SSD accesses (relevant portions of the tree that need to be updated are more scattered in SSD + memory). A blockchain that stores state via QMDB theoretically enables nodes to achieve significantly greater throughput than blockchains using alternative data structures.

This write-up explains how QMDB works and why it is a huge improvement for performant data authentication. We'll build up to QMDB in three steps:

1. Merkle Mountain Range (MMR)
2. MMR + an Indexer
3. QMDB: MMR + an activity bitmap

I'll assume you already know the basics of merkle trees (merkleization and merkle proofs).

LayerZero and Commonware are implementing variations of these ideas as open-source software, and this post is largely based on Commonware's awesome write-ups plus the LayerZero whitepaper and related Rust crates. My goal here is to stitch those sources into a single, beginner-friendly write-up (with graphics) so you can read end-to-end, understand QMDB from first principles, and still have pointers to the more technical references when you want to go deeper.

Merkle Mountain Ranges for Performant Data Authentication (Commonware blog)
The Simplest Database You Need (Commonware blog)

[Grafting Trees to Prove Current State (Commonware blog)]
[QMDB All The Things (Commonware blog)]
[QMDB Paper (LayerZero)]
[commonware_storage crate]

Merkle Mountain Rages

[Merkle Mountain Ranges for Performant Data Authentication (Commonware blog)]

Traditional merkle trees and their variants store key value pairs as leaves: {account_address → account_data}. Each account corresponds to one leaf where the account data can be smart contract code, smart contract data, balance, etc. After an account executes a transaction, every node in the network will have to do the computationally expensive process of updating the account data stored in the corresponding leaf of the merkle tree and re-merkleizing the tree (i.e. recompute hashes of the chain of nodes from merkle root to this leaf).

When executing a transaction, every node in the network will have to do the computationally expensive process of updating the leaves corresponding to account involved in the transaction. The account data in leaves need to be updated and then the tree needs to be re-merkleized (i.e. recompute hashes of the chain of nodes from merkle root to this leaf).

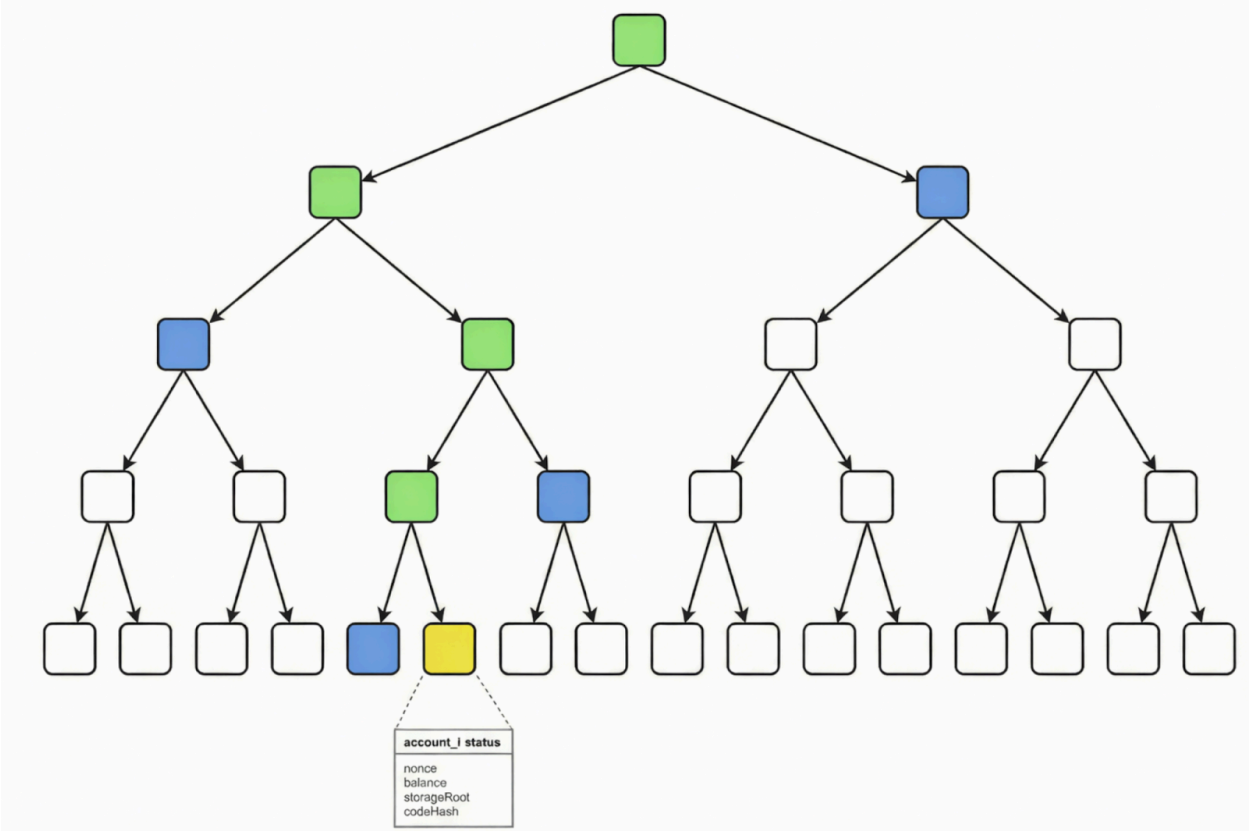

Figure 1: An update to the state of account_i represented by the yellow colored leaf requires re-hashing nodes up the tree by accessing blue node values to recalculate each green node's value.

This merkle tree update & ensuing merkleization of the tree is expensive because a single transaction can touch multiple distant leaves. For each leaf, you must access & update the arbitrary leaf that represent the account involved in the transaction (for example the yellow leaf node in fig. 1), then you must re-merkleize the tree by recomputing hashes for many internal nodes up to the root node. This write amplification is very costly because it will trigger read/writes across the data structure that involve randomly scattered SSD accesses. A more efficient update method would involve mostly reading from memory and only committing contiguous writes. So how could you do that?

This is where Merkle Mountain Ranges (MMRs) are useful! An MMR is a collection of merkle trees (where we call each merkle tree a 'mountain') that only allows appending new data, rather than modifying existing leaves. To be more specific:

"An MMR is a list of perfect binary trees, called mountains, each of strictly decreasing height."

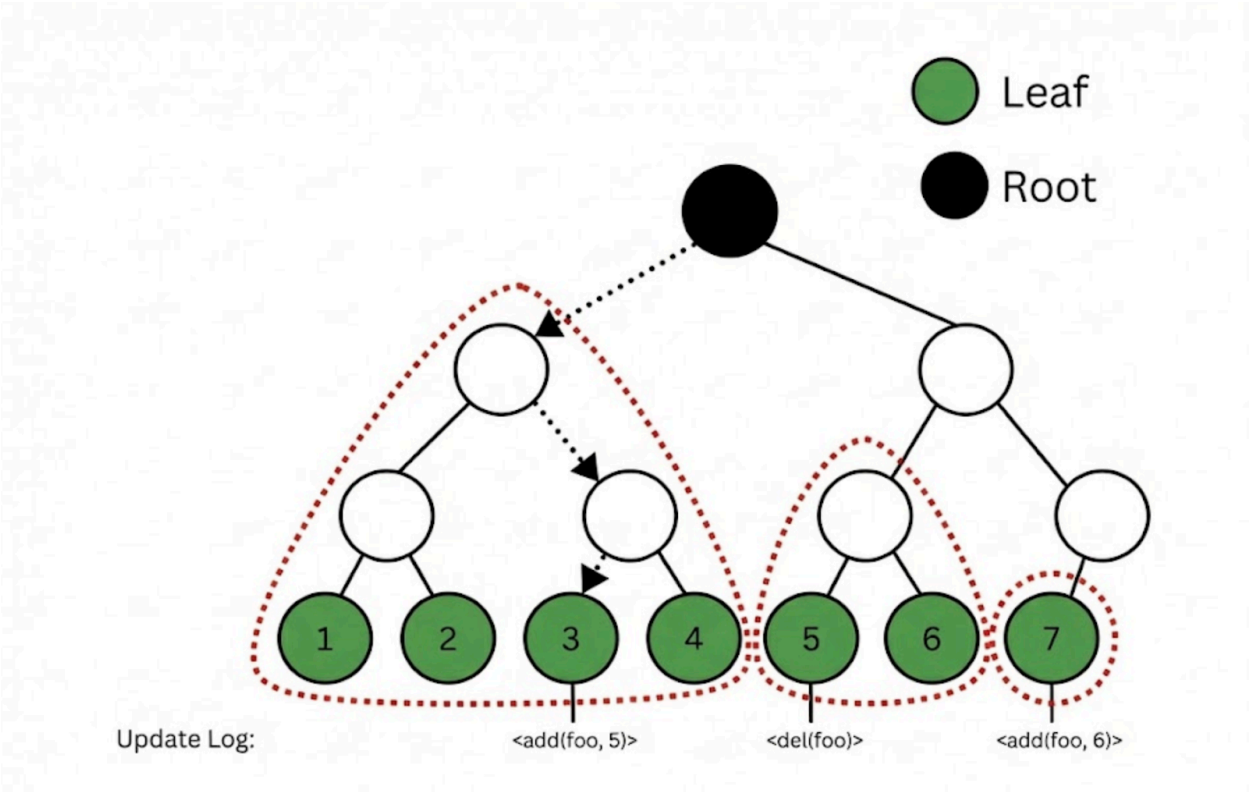

Figure 2: This MMR consists of three mountains (merkle trees) outlined in the red dotted lines. Each update to the variable foo corresponds to appending a new leaf which represents the update.

Where merkle trees usually represent a key-value pair in a single leaf and update the key's value by updating that same leaf in place, merkle mountain ranges instead append a new leaf to the MMR per update. In other words, the MMR acts as a log of operations. Creating a new variable or updating an existing variable are both represented by appending a new leaf to the MMR.

MMRs are very easy to maintain. Every update (i.e. appending a new leaf to the data structure), "...requires generating at most log2(N) number of new internal nodes to re-impose the required properties without modifying any existing nodes." Reference. In simpler terms, every update results in adding a small number of new internal nodes along the far right side of the MMR and it is not necessary to over-write any existing nodes. Write amplification is now heavily reduced!

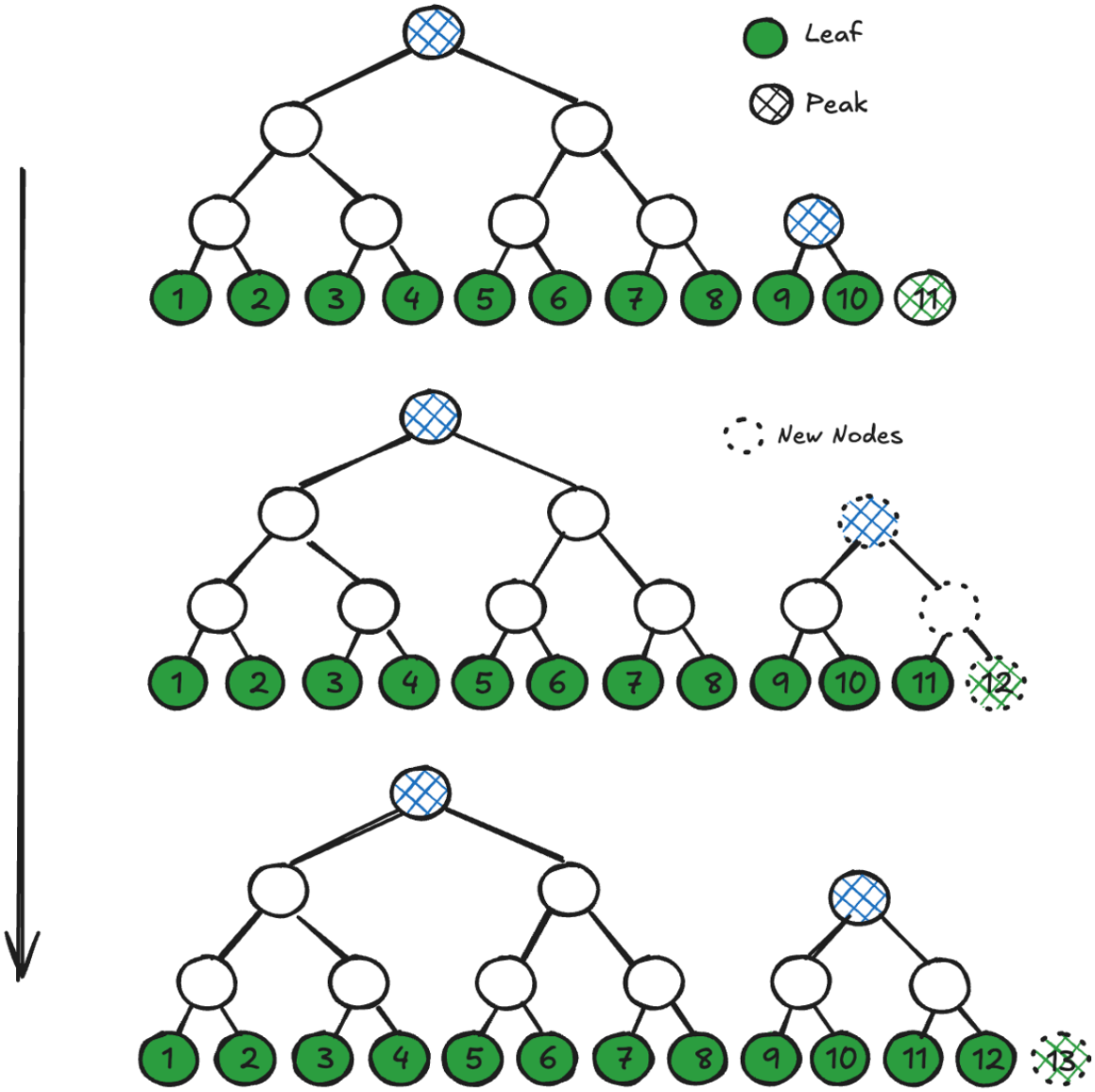

Figure 3: This image depicts the process of adding two new leaves to an MMR and the corresponding internal nodes that are also created in the tree.

The append only restriction and the nature of where writes to the MMR are located means that any update to the data structure results in (1) minimal reads from storage (retrieve the values of a few hashes throughout the MMR) and (2) a single contiguous write to storage (write the values of the newly

created nodes in the MMR). (1) can be explained by the fact that the hashes necessary to re-merkleize after appending an update can be found in locations known ahead of time. Looking at the example from fig. 3, re-merkleizing the tree after appending nodes 12 and 13 only required accessing the root nodes of the other mountains in the MMR. (2) Writes are naturally contiguous because the append-only restriction means that new nodes are allocated sequentially in a file/LSM segment. Additionally, the few reads of existing hashes necessary to calculate the hashes of the newly created nodes is likely small enough to entirely fit in cache/memory. *"Contrast this to a standard Merkle tree, where adding or updating an element can require reading and updating a logarithmically sized amount of data scattered randomly across storage."* Reference

An additional feature can be added to the MMRs to prune old/unnecessary leaves. Because leaves are appended in time order, you can compact the log by discarding old ranges (e.g., everything before an "inactivity floor") and rebuilding a new MMR over the remaining operations. The irrelevant/old leaves can be called inactive leaves, while leaves with necessary data can be called active leaves.

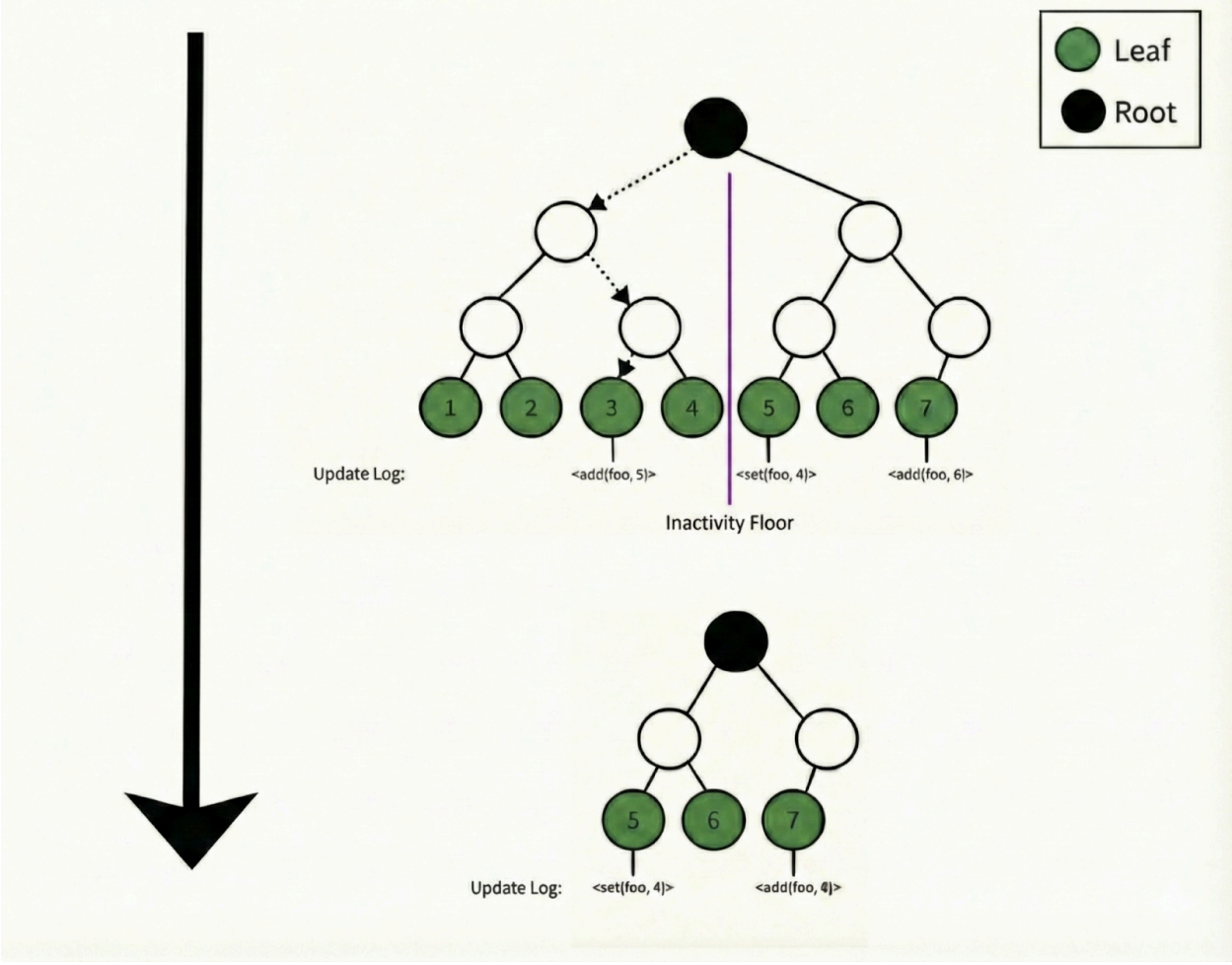

Figure 4: This image depicts the process of dropping old leaves from the MMR and the resulting new MMR.

In practice, “active” and “inactive” operations may be interleaved. One compaction strategy is to scan from the beginning, discard old operations, and re-append any still-relevant ones to the end, see figure 5.

In a blockchain setting, compaction has to be deterministic. Nodes must agree on when to compact and exactly which operations to replay, otherwise they’ll diverge on the root and produce incompatible proofs. Therefore compaction would need some parameters to standardize when it should be run, for example defining either how many inactive leaves are tolerated or how many leaves should be removed at a time.

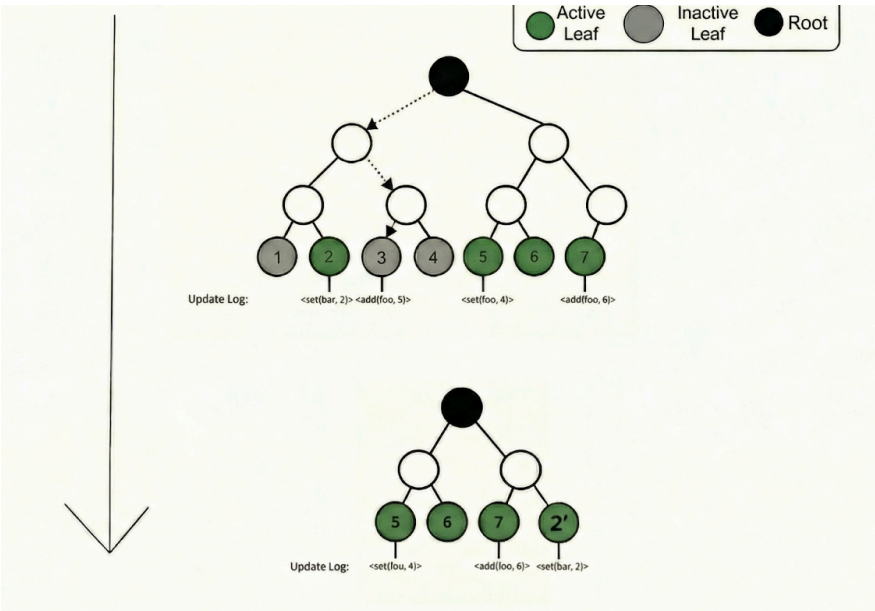

Figure 5: This image depicts the process of pruning old leaves and replaying operations to append them back to the end of the MMR.

There is still an issue with this MMR construction for our needs: it can prove that an operation occurred, but not that it’s the latest operation for a given key. For example in figure 5, someone may provide a merkle proof of operation 5 attempting to prove that foo == 4, which would look valid. However, two steps later foo’s value is updated again and we would have no idea that foo is no longer equal to 4 since we only validated the merkle proof of operation 5. To find the current value of foo, we would need to replay every operation stored in the MMR. Our goal is to create a lightweight authenticated database which is capable of storing a mutable blockchain’s state, this simple MMR doesn’t quite satisfy all of our needs yet.

Merkle Mountain Rages + an Indexer

[The Simplest Database You Need (Commonware blog)]

As a reminder, MMRs are a data structure that allow us to efficiently prove inclusion of an element in a growing ordered list. To be useful for blockchains, what we need is a *lightweight authenticated database* (ADB) capable of storing and proving mutable state efficiently. [Reference]

A pretty simple addition to help us efficiently access the current value of a key is maintaining an indexer alongside the MMR. The indexer is a simple hash map from hash(key) to the position of the most recent update leaf for that key.

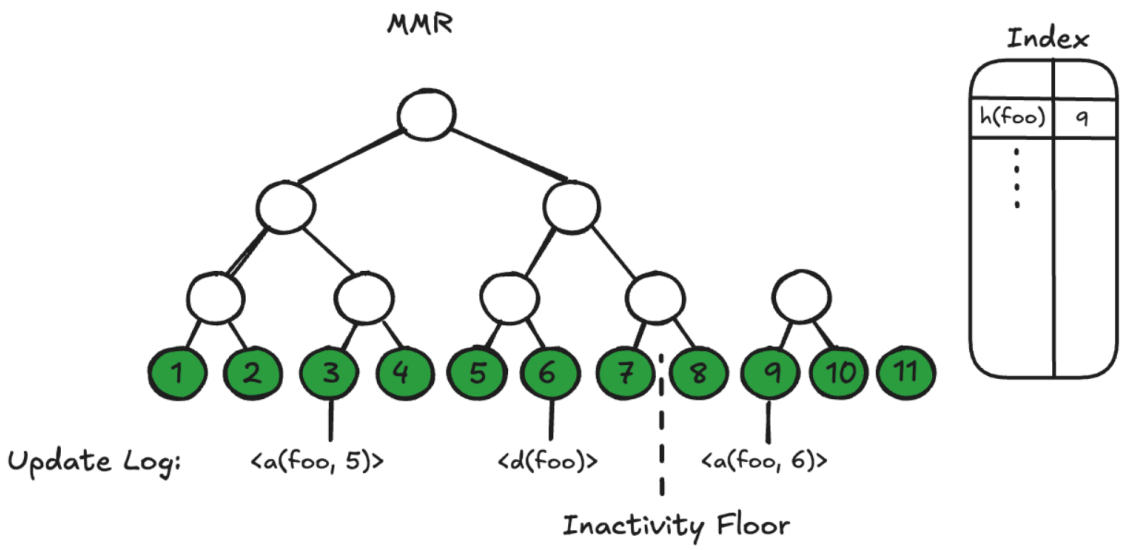

Figure 6: The indexer allows us to easily find the latest value of the key ‘foo’ since it maps to the hash of ‘foo’ to the 9th leaf in the MMR, which stores the most recent update to ‘foo’.

If the indexer is optimized to fit in memory, or largely in memory, reads become much cheaper because you can jump directly to the latest update in the log. However, this indexer isn’t directly built into our merkle proofs. A merkle proof for the value associated with the key ‘foo’ still doesn’t prove that it’s the most recent update for that key. To create proofs of a key’s value along with proof that the value is active requires a little more trickery.

QMDB (Merkle Mountain Rages + a Bitmap)

Grafting Trees to Prove Current State (Commonware blog)
QMDB All The Things (Commonware blog)

The last piece of this puzzle is to modify our MMR to efficiently prove a key currently holds a specific value. In other words, the MMR will need to generate proofs that differentiate if data stored in a leaf is ‘active’ vs ‘inactive’. For example, in figure 5, the MMR would need to be able to distinguish that leaves 3 and 6 are inactive while leaf 9 is active (because leaf 9 is the most recent operation on the key ‘foo’).

A naive way to implement this functionality would be to create a merkle tree mirroring our existing MMR that stores the activity status of each operation (figure 7). In this setup, we have the existing ‘Update Log’ MMR that stores operations and a new ‘Activity Status’ Merkle Tree (MT) which uses a unique leaf to represent the activity status of each operation in the Update Log MMR. Now when a key's value is updated:

1. A new leaf representing the operation is appended to the Update Log MMR.
2. A new leaf representing the active status of this operation is appended to the Activity Status Merkle Tree.
3. The leaf corresponding to the key's previous update is flipped from active to inactive in the Activity Status Merkle Tree. REFERENCE

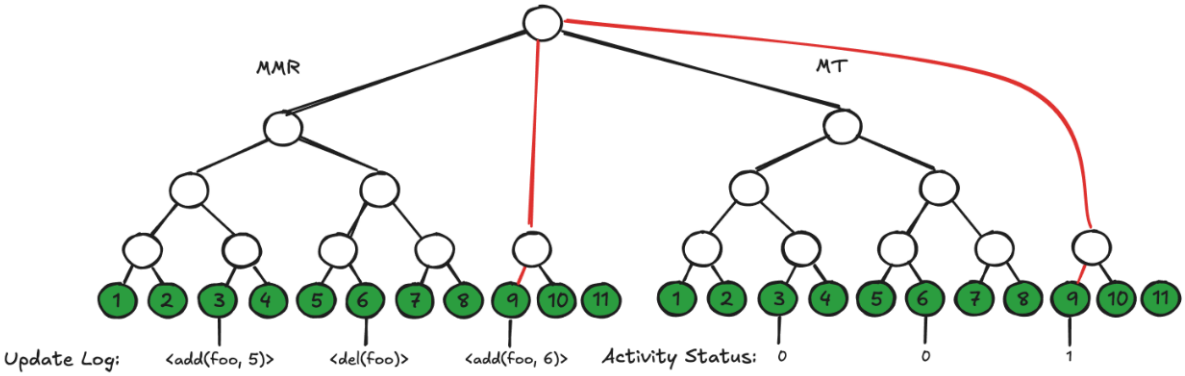

Figure 7: A combined MMR + Merkle Tree (MT) of identical structure allows proving which value for a given key is current. In this figure the siblings along the two paths from root highlighted in red provide the necessary digests for proving "foo currently has value 6". Leaf X in the MMR is an operation with its activity status stored in leaf X of the merkle tree. In the merkle tree, a leaf value of 0 == 'inactive' while 1 == 'active'. Reference

The root hash of this new data structure comes from tying the peaks of the MMR and MT together. A proof of the current value of foo therefore includes the path to the latest operation at leaf 9 in the Update Log MMR as well as the path to the active bit in the Activity Status MT.

You may notice that the new Merkle Tree will reintroduce the write amplification problem because we no longer just append to the MMR. If an existing key's value is updated, we need to flip the activity status bit of the key's previous operation in the MT and re-merkleize the entire tree. This process erases the performance gains that MMRs introduced because the bit representing activity status of the previous update can be located anywhere in the merkle tree and therefore requires randomly scattered reads/writes in storage to re-merkleize.

(ADD A POPOUT OF AN EXAMPLE HERE: Say we modify a variable 'bar' that was last modified in operation 5. First we would need to flip the activity status bit of operation 5 from 1 to 0, then append the operation modifying 'bar' to the 'Update Log' MMR, then append a new 'Activity Status' bit set to 1 to the end of the 'Activity Status' merkle tree. Appending leaves to the merkle tree and MMR are performant as we have already explained, but flipping the bit in leaf 5 representing operation 5's activity status will require re-merkleizing several internal nodes from storage.)

However, this issue can be solved by compacting the MT to fit in memory. Figure 8 depicts a compacted Activity Status MT that represents the activity status of several leaves in a compressed bitstring.

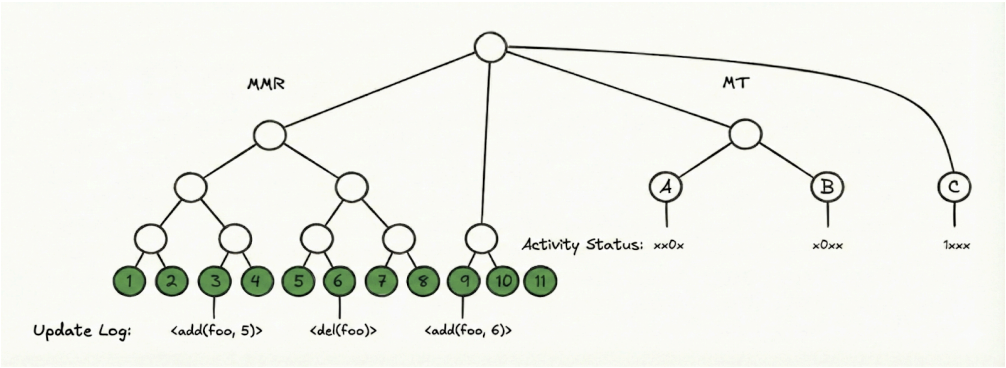

Figure 8: Node ‘A’ contains the activity status of leaves 1-4 as a 4 digit binary number ‘xx0x’ (the ‘x’ is for an ambiguous status since those leaves aren’t defined in this example image, but you can imagine that the ‘x’ is supposed to be a 1 or 0), node B contains the activity status for leaves 5-8, and node C contains the activity status for leaves 9-11.

This compaction technique scales efficiently to support much larger merkle trees. As the Update Log MMR grows with more levels and leaves, nodes like ‘A’, ‘B’, and ‘C’ can simply use longer bitstrings—such as 16, 32, or 64 bits—to act as a bitmap that represents activity of a greater number of operations in the MMR. This allows the MT to represent activity status of a large number of operations without significantly increasing its size. The end result is a Merkle Tree representing the activity status of every operation that can fit entirely in memory, significantly improving the performance of updates and re-merkleization!

Now we have fast updates to the MMR with minimal write amplification, but a current-value proof from this data structure could be up to 2x larger because it includes the path to an operation leaf and to an activity status leaf. However, if we maintain the structure of the Activity Status MT to match exactly that of the MMR with its bottom N levels removed, we can layer each leaf of the MT onto its corresponding internal node of the MMR. REFERENCE .

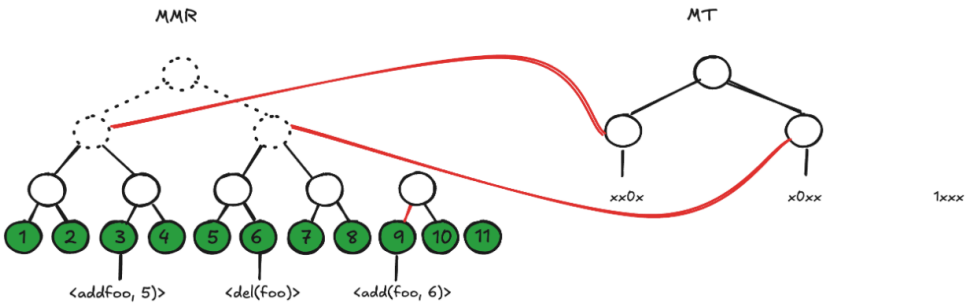

Figure 9: Layering each MT leaf onto its corresponding node in the MMR.

Generating a proof of an operation includes the same internal nodes in the MMR as before, except it also includes a bitmap which contains the operation’s activity bit. REFERENCE. This structure is now essentially QMDB!

The QMDB Weeds

QMDB Paper (LayerZero)

The previous section described QMDB at a high level in casual language, now I'll dive deeper into some details from the QMDB whitepaper with LayerZero's more technical terms.

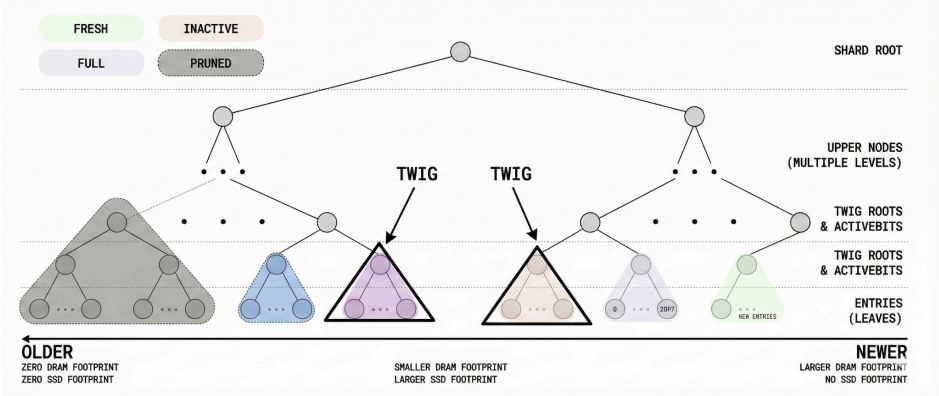

Figure 10: Detailed QMDB diagram labeled with technical terms.

Entry: The primitive data structure in QMDB where key-value pairs are stored, each leaf is an entry that stores an operation updating a key's value alongside some metadata. Entries are append only and immutable. The QMDB key is generated by hashing the application level key.
Twigs: Subtrees with fixed depth and and entry count. Additionally, twigs store a bitmap at the twig root which represents the activity status of each of its entries. (Commonware's QMDB implementation has 256 leaves in a twig and therefore a 256 bit map at the twig root)
Upper Nodes: Can be several levels of nodes that connect twigs.
Shard Roots: Connects upper nodes to represent the state that is managed by an independent QMDB shard. The QMDB is separated into shards because (1) each shard can be defined by the prefix of keys it contains, allowing the DB to save space by storing only the suffix of the key (the prefix for each shard is already known, for example shard #1 could contain keys with prefix 0xAAA, shard #2 contains keys with prefix 0xAAA, ...) and (2) locking at the per-shard level on CRUD operations will enable greater concurrency than locking on the DB-wide state if no sharding was used.
Global Root: Connects all of the shard roots to represent the entire world state.

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

Figure 11: Fields in a QMDB entry.

Entries have several fields, each of which is necessary for storing information directly related to the entry or for generating a state proof related to the entry. Key, Value, and Version are relatively straightforward. Id's are a nonce-like value that allow distinguishing between several entries that may be stored in the QMDB for the same key. NextKey, OldId, and OldNextKeyId act as pointers to related keys or entries, storing these pointers in the entry itself allows the generation of various types of inclusion, exclusion, and deletion proofs.

Twigs compress all of their entries into a hash at the twig root level alongside a bitmap for activity status. Therefore, merkleization requires only reads/writes to the global root, shard roots, upper nodes, and twig roots (i.e. the data below the twig root level is not needed for merkleization). All of this data needed for merkleization is small enough to fit in DRAM rather than being stored in SSD. (Section 3, QMDB Whitepaper)

Twig States:

- Fresh: Twig with space for more entries. The twig root and twig entries should live in DRAM because as new entries are appended, hashes throughout the Fresh twig will need to be recomputed.
- Full: Twig that has filled its capacity of entries where at least 1 of the entries is active. The twig root will live in DRAM and twig entries can live in SSD.
- Inactive: Full of entires that are all inactive. The twig root will live in SSD and twig entries are ideally deleted
- Pruned: Twig root and all twig entries deleted.

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

Figure 12: Twig states defined for twigs that have a capacity of 2048 entries.

Twigs cycle through the four states listed above. There is only ever one 'fresh' twig in a QMDB

shard, and new entries are sequentially inserted into the fresh twig until it is ‘full’. All of the entries in a full twig can be flushed to SSD in a sequential write and deleted from DRAM. Remember that merkleization does not require updating nodes below the twig root, so once a twig is full, we only need to keep the twig root in DRAM.

After all of the entries in a full twig are marked inactive due to later update or delete operations, the twig as a whole transitions to ‘inactive’. Entries in inactive twigs can be deleted from SSD because these operations are no longer in use, however, the twig root still needs to live in SSD so that the QMDB structure can be reconstructed. Finally, entire inactive twigs can be deleted, transitioning the twig to ‘pruned’, and the QMDB is reconstructed without the pruned twigs. Upper nodes that contain only pruned twigs below can also be recursively pruned to further compact the QMDB.

Since the QMDB grows in size with new operations, we are motivated to accelerate the twig lifecycle and delete old entries. This can be done by removing old entries and re-appending them to the fresh twig (process previously described HERE), allowing more opportunities to prune subtrees.

QMDB Indexer

LayerZero’s whitepaper includes an in-memory indexer which maps application level keys to their latest entry in the MMR. This indexer is implemented as a B-Tree which allows iterating over keys in order, a necessary feature to generate *exclusion proofs* (more on this later). Commonware provides several varying implementations of QMDB, including versions which build the indexer with a simple hash map but are incapable of generating exclusion proofs.

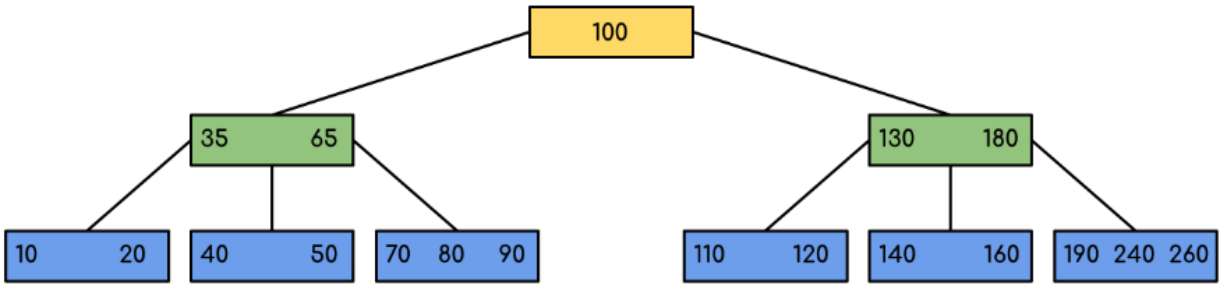

Figure 13: Example of what a simple B-Tree looks like. It is out of scope to explain how it works here, but you can read more online. Starting at the root, it is O(logn) to find an item stored in the tree. Also, you have the ability traverse items sequentially by value in the B-Tree.

CRUD Operations

(White paper section 3.3)

QMDB supports Create, Read, Update, and Delete capabilities on key-value pairs in the database. In this section I’ll walk through the details of how each of these works. For the moment, we will ignore an entry’s Id, Version, and Value fields because these aren’t relevant. So each entry E will be defined as:

E = (Key, NextKey, OldId, OldNextKeyId)

Read: When reading the value associated with a key, first query the indexer by the key you are searching for, then it will return the location in SSD of the key's latest QMDB entry, use this location to read the entry and retrieve the key's value in a single SSD IO.

Update: Requires first reading the most recent update for the key, flipping it to inactive, then appending a new entry to the Fresh twig. That is 1 SSD read and 1 entry write. For an existing entry E, the new entry *E′* updating the key's value would be:

$$E' = (E.Key, E.NextKey, E.Id, E.OldNextKeyId)$$

(Where E.Key is entry E's key value, E.NextKey is entry E's NextKey, etc.)

Since *E′* is updating the same Key as E, they both share E.Key.

Since the NextKey value is defined as the "Lexicographic successor of Key", the NextKey value is the same in E and *E′*. In a little more detail, say we have a QMDB that stores values for two keys 0xA and 0xB and say E is an existing entry for key 0xA. Key 0xA's lexicographic successor is 0xB in the QMDB, so E.NextKey == 0xB. When *E′* is appended to the QMDB to update key 0xA's value, the lexicographic successor of the key updated by *E′* is the same as E, i.e. the NextKey value is the same in E and *E′*.

Since the OldId value is defined as the Id of the entry that last modified Key, the OldId of *E′* must be E's Id, E.Id.

Lastly, as a part of this entry write, we must flip the active bit of the existing entry E to inactive at the twig root level (this is done in memory).

Create: Involves appending a new entry corresponding to the key-value pair being created and updating an existing entry whose NextKey should be the newly created Key. The 'update' is required because the new key in the data structure needs to fit into its lexicographically correct position among the chain of NextKey pointers stored in entries. There may be an existing key 0xA that stores its next lexicographic key as another existing key 0xC, but if 0xB is created, the key 0xA should now point at 0xB which should point at 0xC.

So the first step is to read in the entry E_p corresponding to the lexicographic predecessor (prevKey) to the created key K. Right now, the entry E_p stores some E_p.nextKey, the created key K must satisfy: $E_p < K < E_p$.nextKey. Now we must 'update' the entry E_p to make K its lexicographic successor and append K with E_p.nextKey as its successor. As we explained earlier, an update is simply flipping the active bit of the existing operation to inactive and appending a new active entry for the same key. So after flipping the active bit of E_p to false, we append these two entries to the fresh twig:

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

So now the lexicographic successor of key of E_p has been updated in E'_p to be K while the lexicographic successor of E_K is now E_p.nextKey (again remember that K is being created in an existing tree where $E_p < K < E_p$.nextKey). The oldId for E'_p is set to E_p.Id. The oldId for E_K is also set to E_p.Id despite the fact that K is a new key which has no previous Id.

So a Create operation requires two entry writes for both newly appended entries along with one SSD read to retrieve the lexicographic predecessor of the created key.

Delete: This operation is straightforward and the whitepaper explains it nicely:

“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 entries E_K and E_p corresponding to the 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:

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

Deleting a key in QMDB incurs 2 SSD reads and 1 entry write.”
(Section 3.3, CRUD Interface)

Proofs

Say QMDB is used by a blockchain to maintain state. If a node wants to prove to others that account X holds a balance of 5 coins, how would it do that? It would need to generate an *inclusion proof* for the key corresponding to account X. If instead the node wants to prove that account X does *not* exist in state or has no balance entry, it would need to generate an *exclusion proof* for the key corresponding to account X. Additionally, we could create *historical inclusion/exclusion* proofs for the key associated with some account X at any block height H.

An inclusion proof for key K is simply the Merkle proof π for entry E such that E.Key = K. This entry E can be obtained by querying the indexer for the latest operation done by K.

An exclusion proof for a key K is the Merkle proof π for entry E such that E.Key < K < E.nextKey. Since K != E.nextKey, π proves that K does not exist in the QMDB. Finding E such that E.Key < K < E.nextKey is quite performant if the indexer is a B-tree because the B-tree is built for iteration over the ordered list of keys.

The OldId and OldNextKeyId can be used to create a historical graph of operations which then can generate historical inclusion & exclusion proofs. Additionally, the entire world state at any previous block height can be reconstructed using the OldId, OldNextKeyId, and Version fields. I won’t dive deep into this here because it is a little more niche, but this capability explained in more detail in the whitepaper (Section 3.4 Proofs).