Understanding SHAMap

Valentin Balaschenko, Ripple

XRPL Core Dev Bootcamp 2025

Commons

Core Ledger Team @ Ripple

- Low-level protocols, high-performance code
- Personal focus: memory + throughput scalability
- We're hiring! Engineers welcome
  - Staff Software Engineer -> https://ripple.com/careers/all-jobs/job/6437475/
  - Senior Software Engineer -> https://ripple.com/careers/all-jobs/job/6437475/

Agenda

- What is SHAMap?
- Why SHAMap matters
- Data Structure Fundamentals
- Nodes and Hashing
- Mutability & Snapshots
- Traversal & Iteration
- Synchronization & Proofs
- Storage, Caching, and Thread Safety
- Q&A

SHAMap in One Sentence

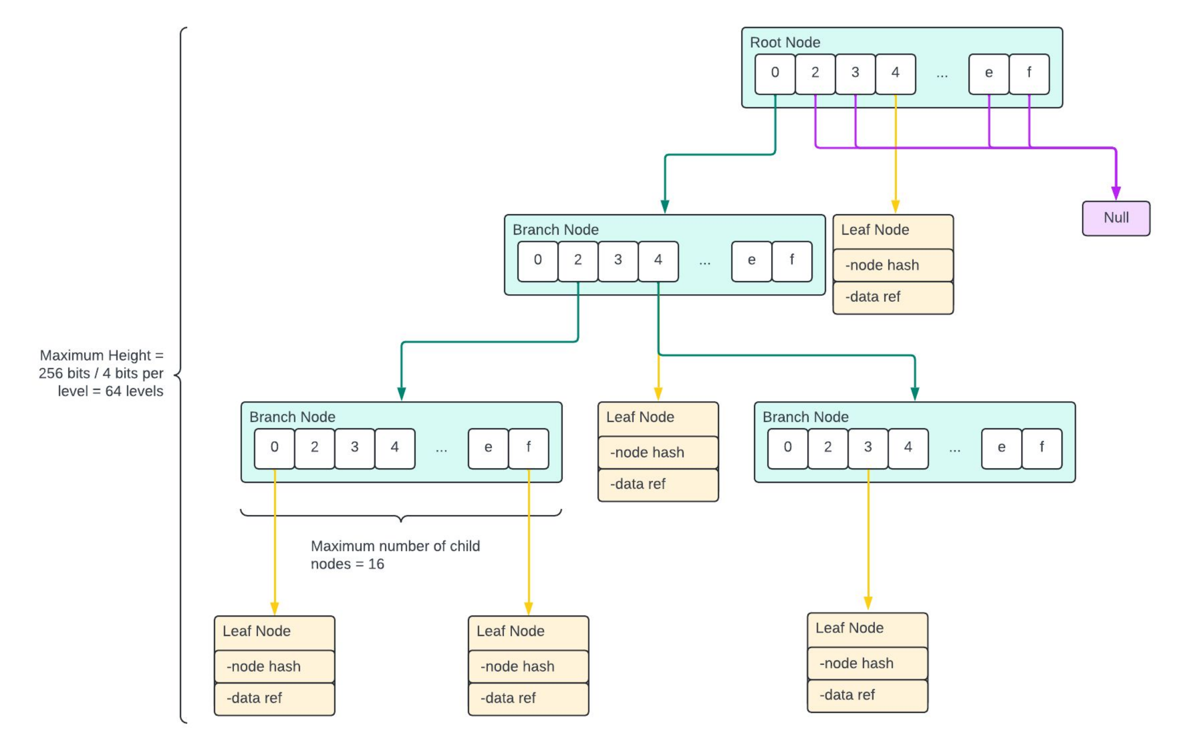
- A hybrid of Merkle tree and radix trie used to store transactions or ledger state

TODO: remind Merkle and Radix tree/trie structure

Role of SHAMap in XRPL

- Backbone of ledger structure
- Used to store:
  - Ledger State
  - Transactions

High-Level Diagram



Commons

Why Merkle Tree + Patrcia Trie?

- Efficient comparison: O(1) with hashes
- Cryptographic integrity
- Ordered key space

Uniform Leaf Type

- SHAMap is homogeneous: leaves are of one type per map
- State map vs. Tx map

Types of SHAMap Nodes

- SHAMapTreeNode (base)
- SHAMapInnerNode
- SHAMapLeafNode (+ subclasses)

SHAMapInnerNode Overview

- Up to 16 children
- Bitset + hashes
- Holds no data
- FullBelow optimization

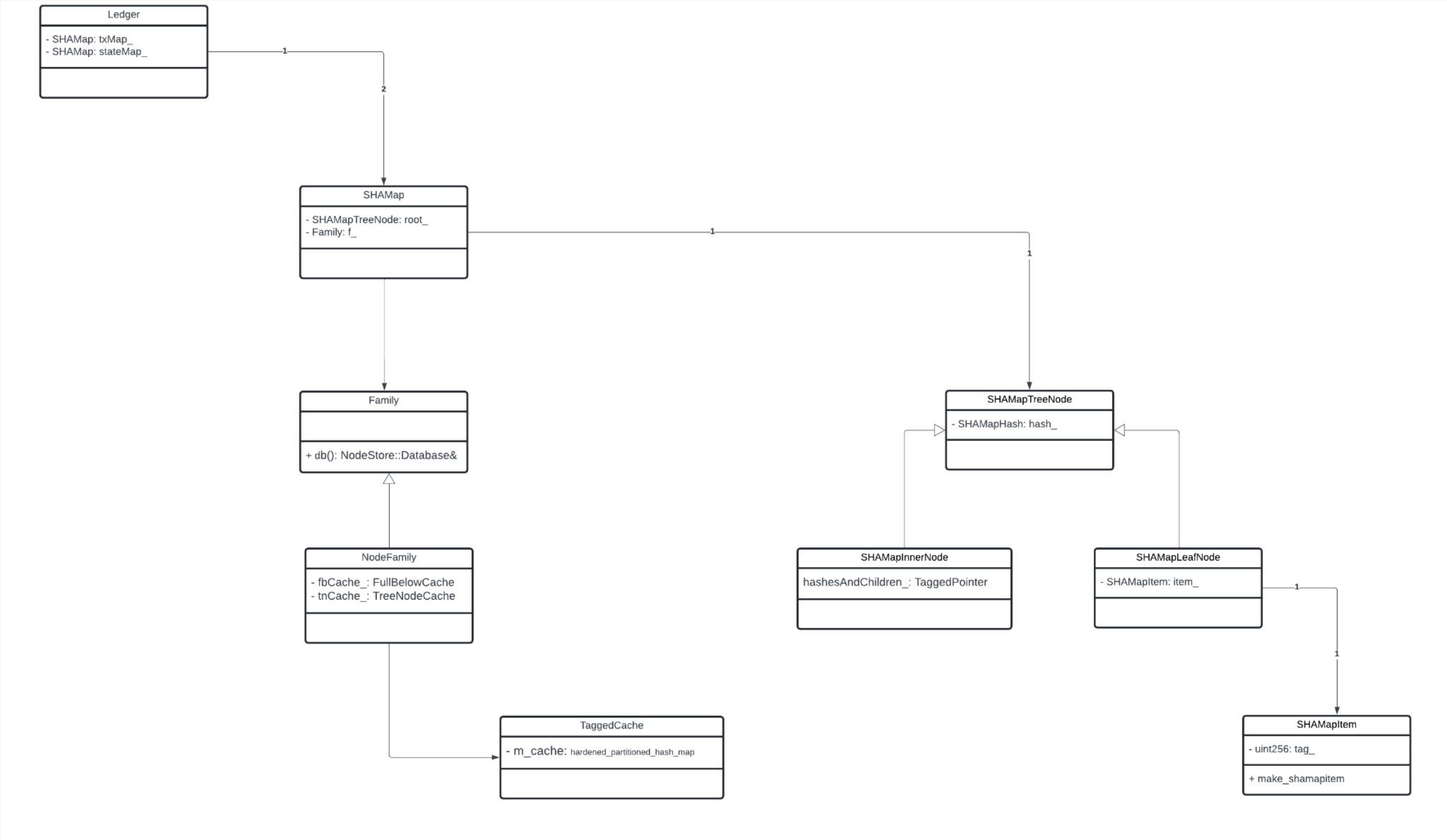
SHAMapLeafNode Overview

- Holds SHAMapItem
- Can be account state, tx, or tx+meta
- Immutable once inserted

SHAMapItem

- tag
- size
- raw data
- intrusive_ptr based

Class diagram



Hashing: The Foundation

- Each node stores a hash
- Inner = hash of children
- Leaf = hash of content

Hash Update Logic

- `updateHash()` on leaves: compute hash
- Inner node: recompute after all children are updated

SHAMapHash Type

- Custom hash type for storage
- Used pervasively across nodes

Use Cases for Hashes

- Fast comparison
- Merkle proofs
- Data integrity in DB serialization

Copy-On-Write Model

- Nodes shareable via shared_ptr
- Modified via clone (copy-on-write)
- Each SHAMap has a `cowid`

Mutability & Snapshots

- Immutable: for validation/sync
- Mutable: for building ledgers
- Snapshots: efficient cloning

Traversal

- visitNodes: DFS traversal
- visitLeaves: filter + callback

Iteration via const_iterator

- Ordered iteration
- STL-style API: `begin()`, `end()`, `lower_bound()`

walkMap vs walkMapParallel

- walkMap: missing node detection
- walkMapParallel: multithreaded sync

Syncing – getMissingNodes

- Traverses tree
- Uses deferred async reads
- Marks fullBelow on success

Proof Paths

- `getProofPath()` builds Merkle path
- `verifyProofPath()` checks it
- Used for light clients

Canonicalization

- Ensures one node per hash
- Avoids duplicates in memory

Node Identification (SHAMapNodeID)

- Encodes depth + path as uint256
- Methods: `getChildNodeID`, `createID`, etc.

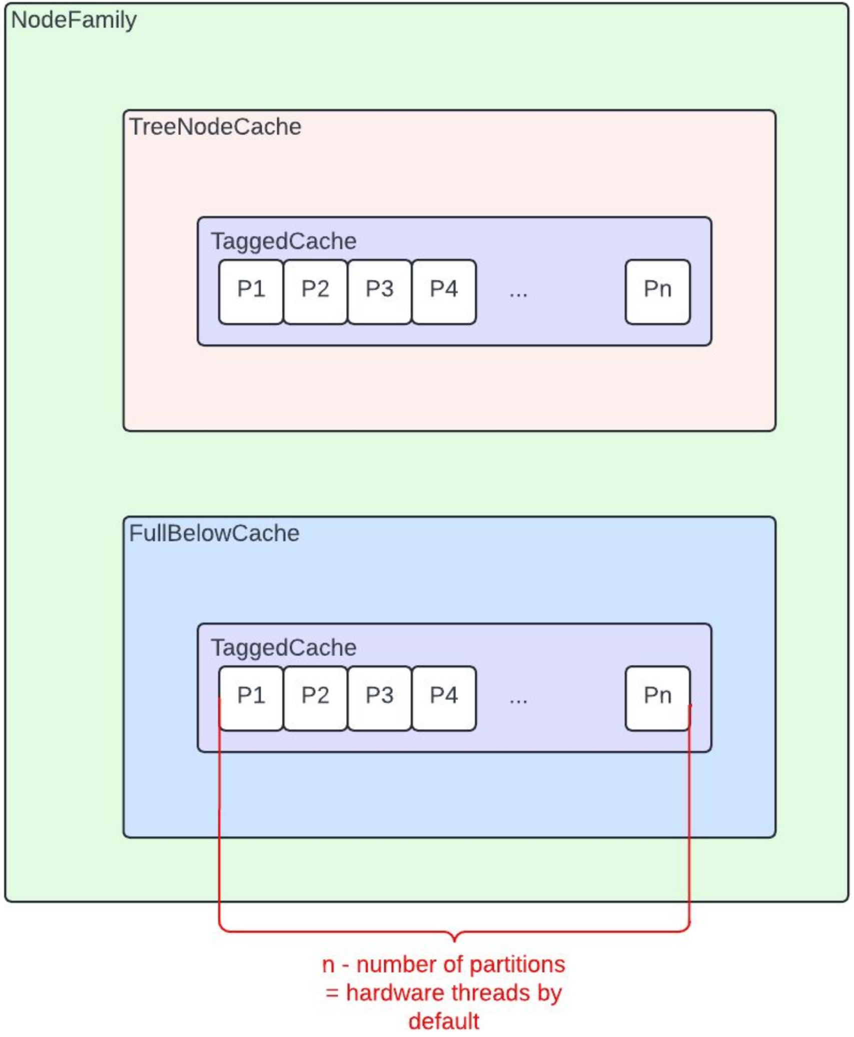
Inner Node Serialization

- Full format: all 16 branches
- Compressed: skip empty branches
- Chosen automatically

Commons

Caching Architecture

- TreeNodeCache (hash-indexed)
- FullBelowCache (fully synced subtrees)
- Family abstraction (cache+DB mgmt)

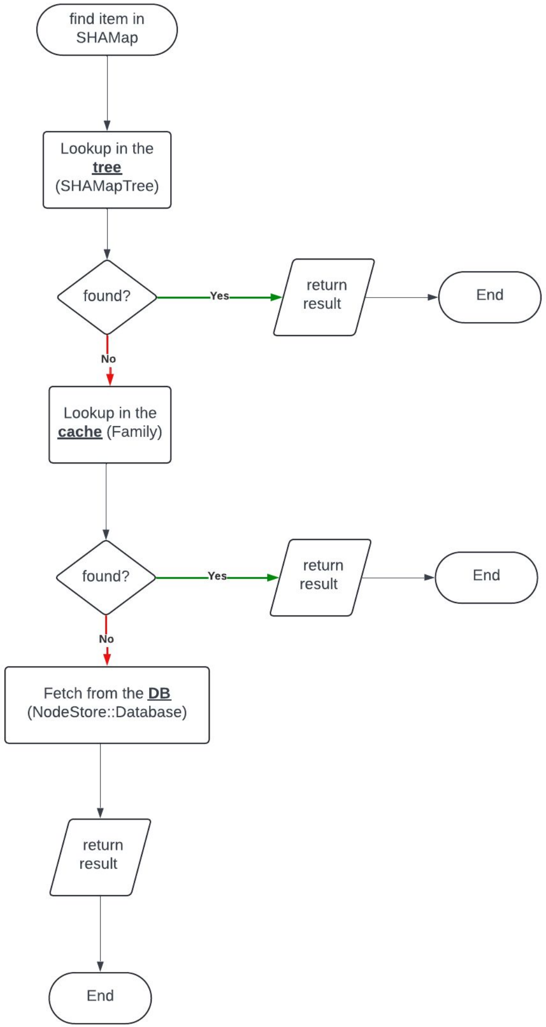


Commons

Thread Safety Mechanisms

- Atomic access in SHAMapInnerNode
- canonicalize() logic
- Cache-level synchronization

SHAMap as a gateway



Commons

Design Summary

- Immutable-by-default
- Radix-16 + Merkle
- Fast sync, proof, comparison

What's Next?

- Memory footprint optimisatons
- Transactions throughput optimisations

Thank you!

- Questions?
- Exercise (TODO)

Commons