mLSM: Making Authenticated Storage Faster in Ethereum

Pandian Raju[1], **Soujanya Ponnapalli**[1], Evan Kaminsky[1], Gilad Oved[1], Zachary Keener[1]
Vijay Chidambaram[1,2], Ittai Abraham[2]

[1]The University of Texas at Austin;
[2]VMware Research

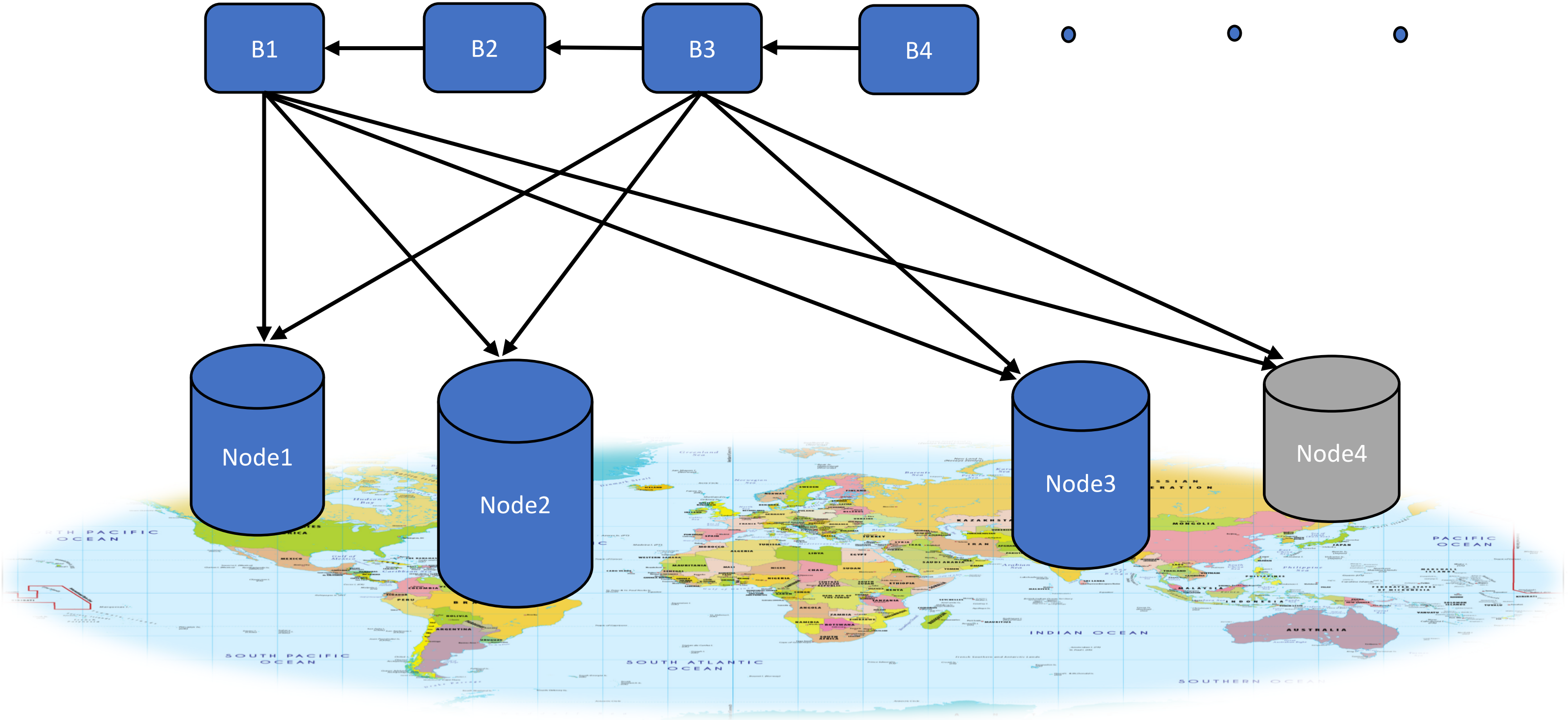
TEXAS
The University of Texas at Austin

vmware

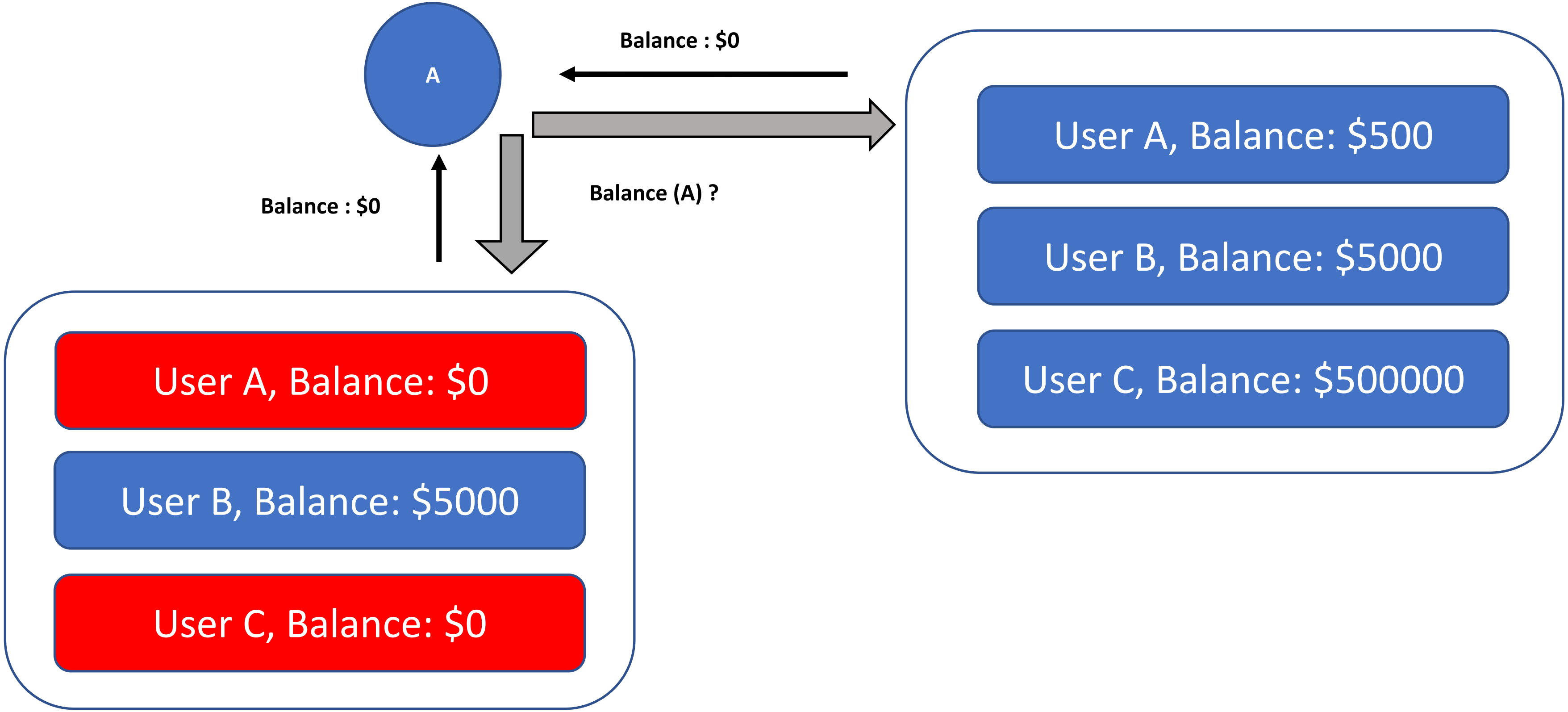
Ethereum

- Distributed software platform
- Cryptocurrency applications
- Key-value store
  - Accounts : Balances
  - Trustless Decentralized setting

Ethereum – Distributed Decentralized System

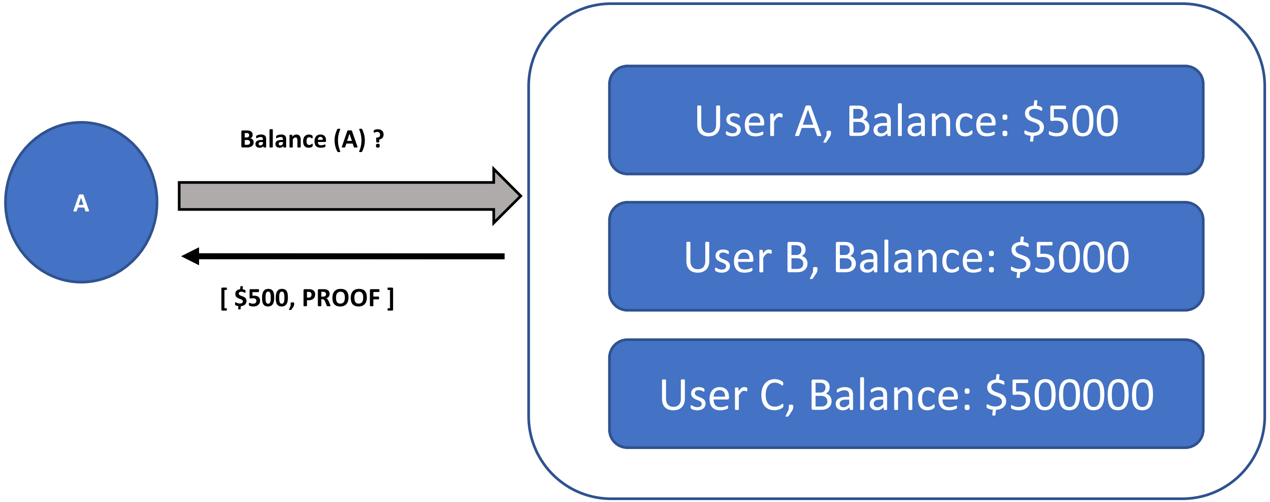


Need for Authenticated Storage



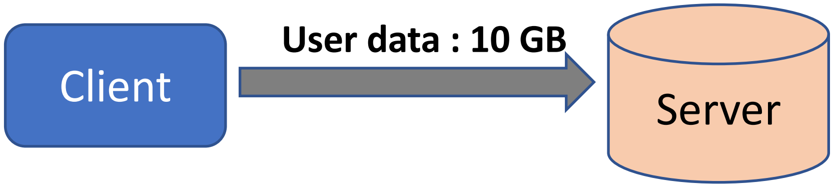
Authenticated Storage

- Users can verify the value returned by a node
- Each read is returned with the value and a proof



Authentication Techniques in Ethereum

- Ethereum authenticated storage suffer from high IO Amplification
- 64x in the worst case
- **IO Amplification**
  - Ratio of the amount of IO to the amount of user data



User data : 10 GB
Total write IO : 500 GB

Write Amplification : 50

Why is IO Amplification bad?

- Reduces the write throughput
- Directly impact the life of Flash devices
  - Flash devices wear out after limited write cycles

(Intel SSD DC P4600 can last ~5 years assuming ~5 TB write per day)

For the same SSD life expectancy, with 65x IO Amplification, instead of 5TB of data we can now only write ~75 GB of user data per day

How to design an authenticated storage system that minimizes IO amplification?

Merkelized LSM

- Maintains multiple mutually independent binary merkle trees
- Decouples lookup from authentication
- Minimizes IO Amplification

Outline

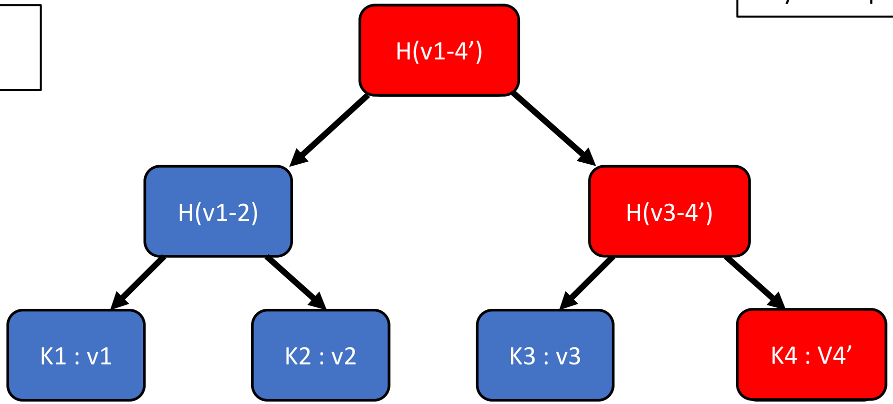
- Authentication in Ethereum
- Why caching doesn't work?
- Merkelized LSM

Authenticated Storage in Ethereum

Merkle Trees – Fundamental building blocks

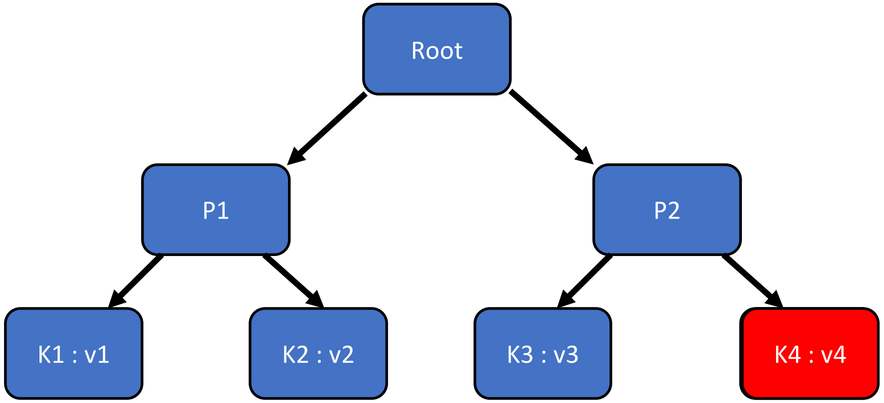
Root hash is publicly available to all clients

With a constant sized root hash, we can authenticate all the key-value pairs



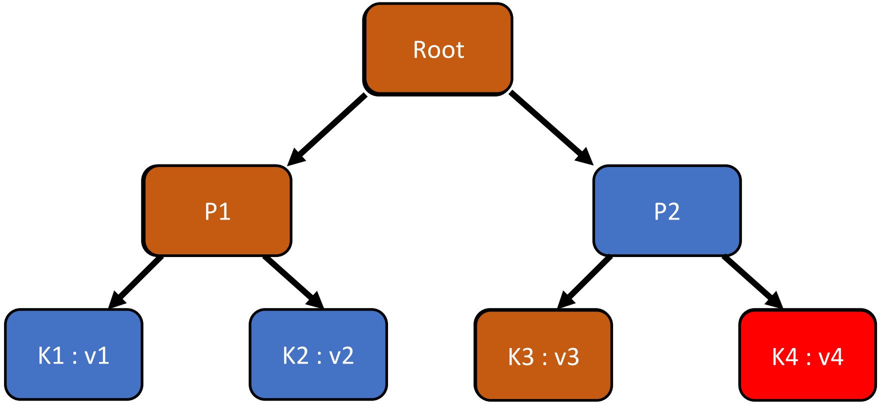
Authentication using Merkle Trees

- Client queries for value of key k4
- Server replies with the value



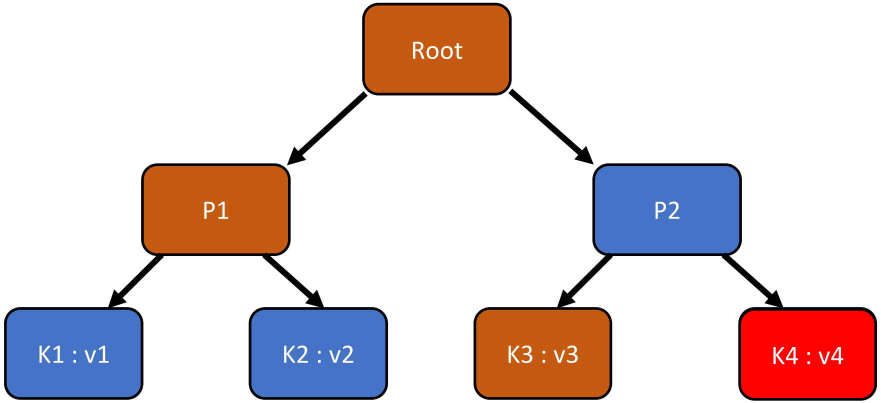
Authentication using Merkle Trees

- Client queries for value of key k4
- Server replies with the value
- Along with a Merkle Proof

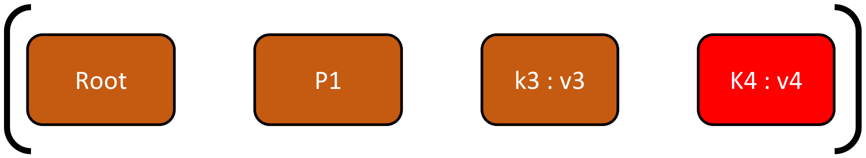


Authentication using Merkle Trees

- Client queries for value of key k4
- Server replies with the value
- Along with a Merkle Proof

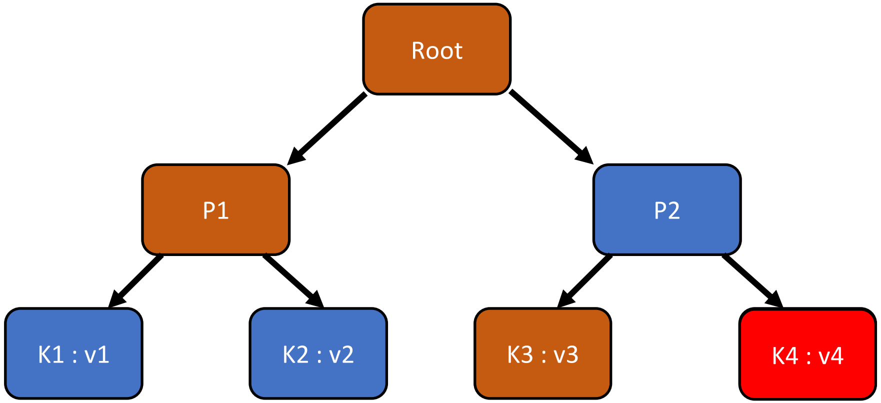


Response :

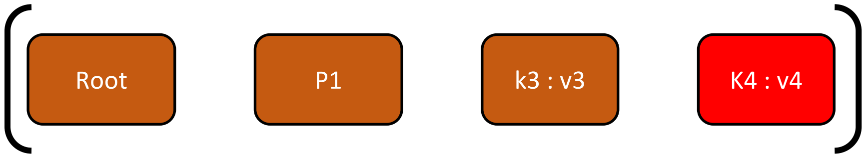


Authentication using Merkle Trees

- Client verifies the value by calculating the root hash from the value and Merkle proof

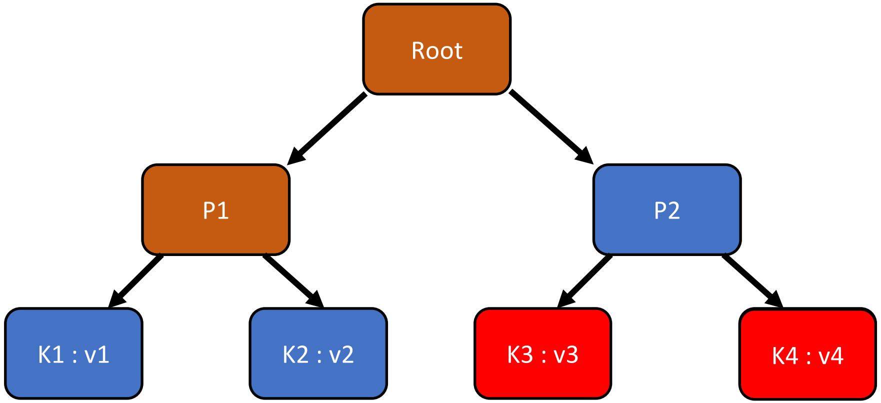


Response :

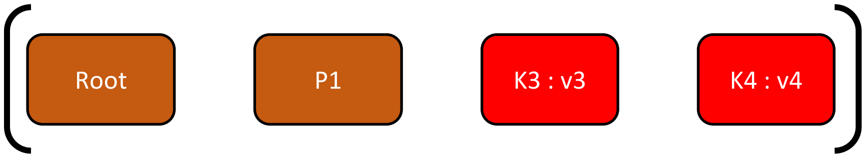


Authentication using Merkle Trees

- Client verifies the value by calculating the root hash from the value and Merkle proof

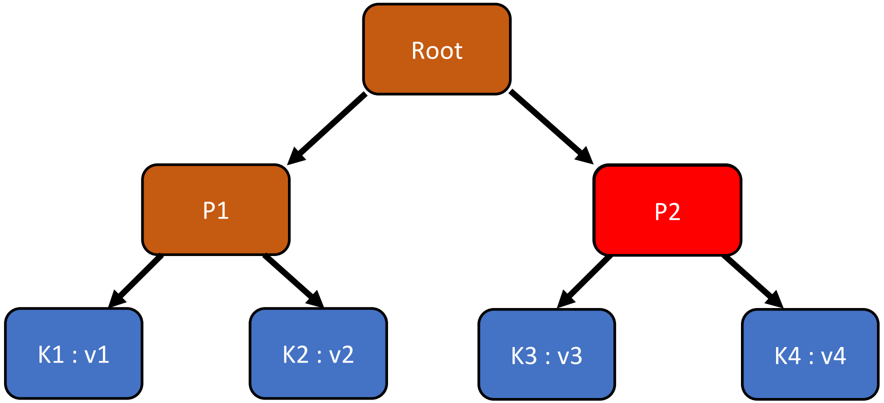


Response :

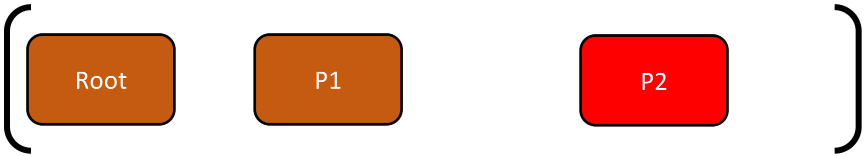


Authentication using Merkle Trees

- Client verifies the value by calculating the root hash from the value and Merkle proof

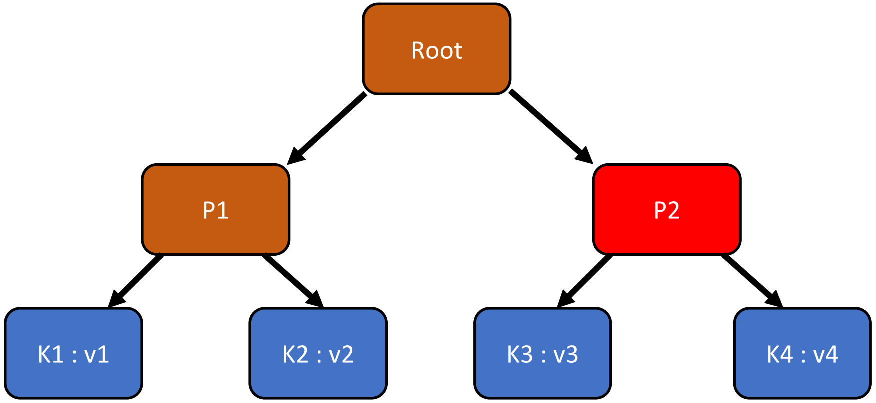


Response :

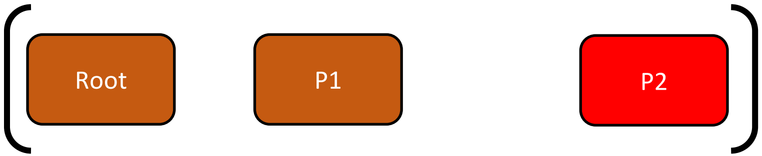


Authentication using Merkle Trees

- Client queries for value of key k4
- Server replies with value and a Merkle Proof

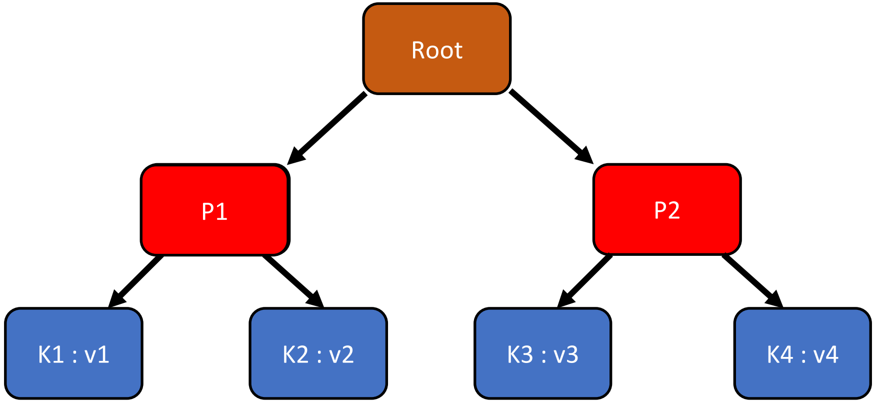


Response :

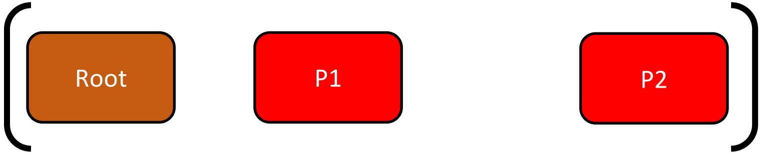


Authentication using Merkle Trees

- Client queries for value of key k4
- Server replies with value and a Merkle Proof

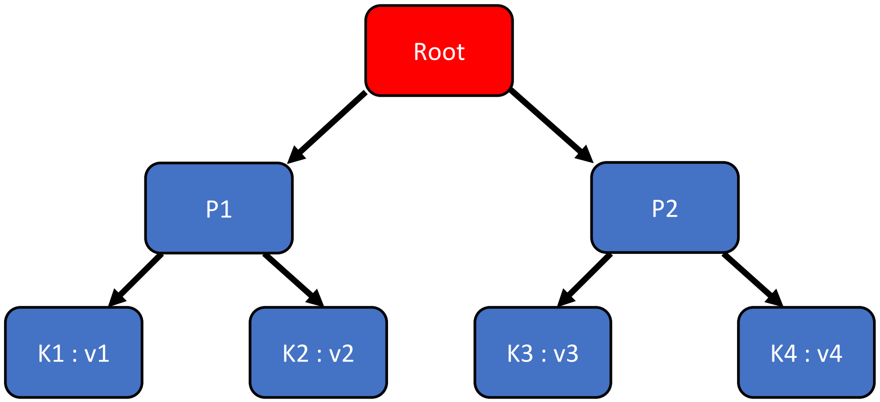


Response :

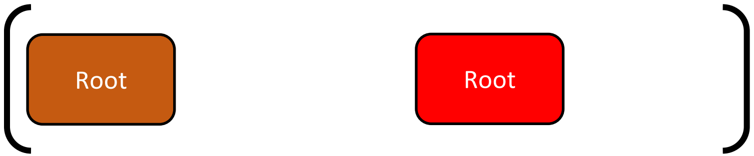


Authentication using Merkle Trees

- Client verifies the value by calculating the root hash from the value and Merkle proof

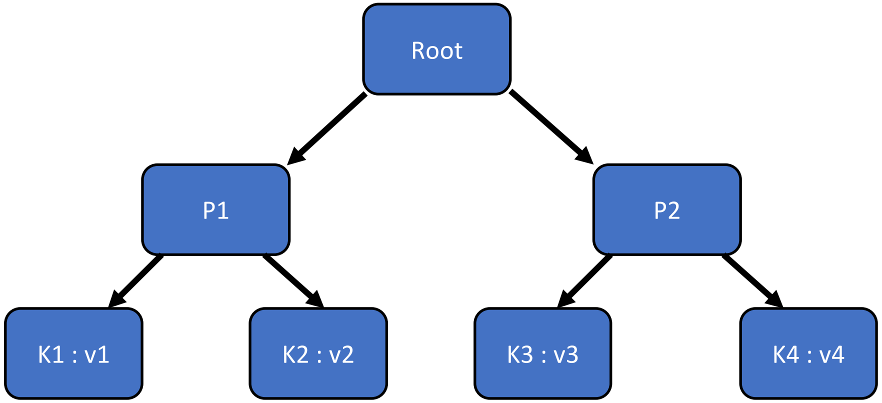


Response :

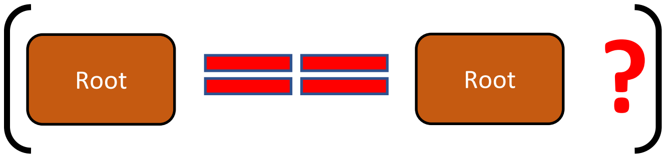


Authentication using Merkle Trees

- Client verifies the value by calculating the root hash from the value and Merkle proof

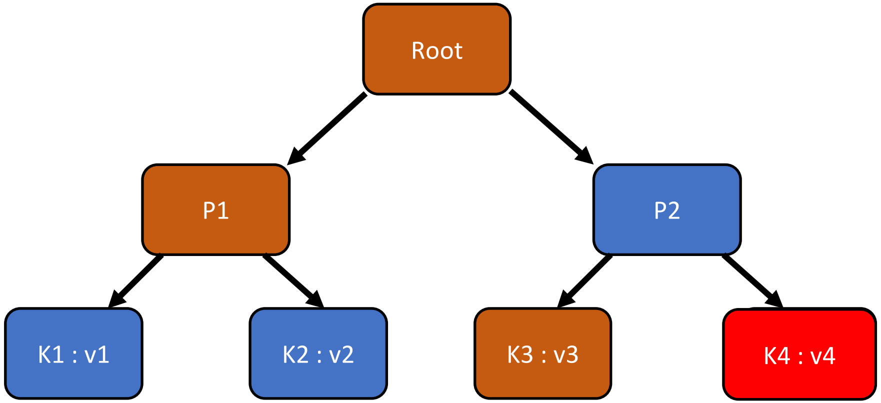


Response :

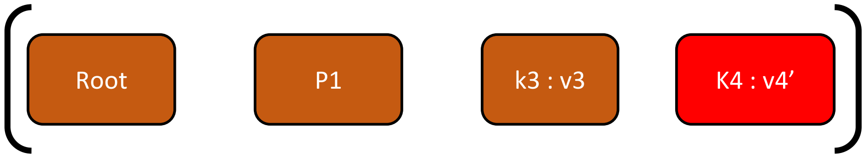


Authentication using Merkle Trees

- Server can no longer lie about the data

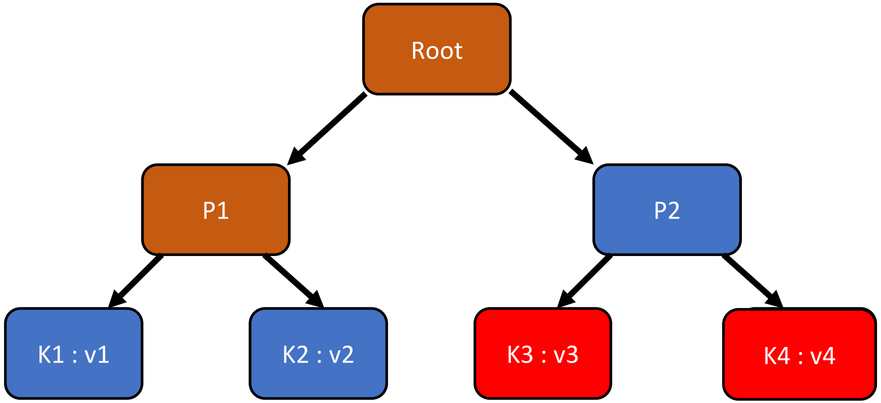


Response :

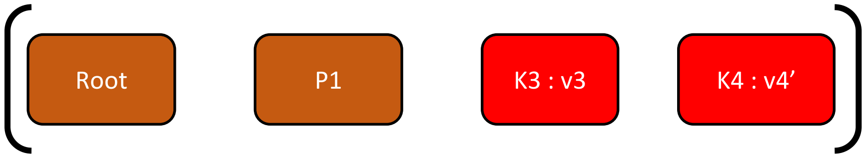


Authentication using Merkle Trees

- Server can no longer lie about the value

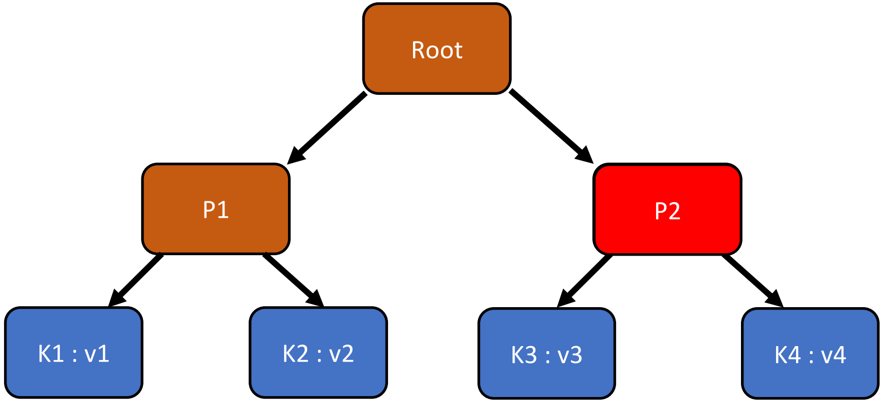


Response :

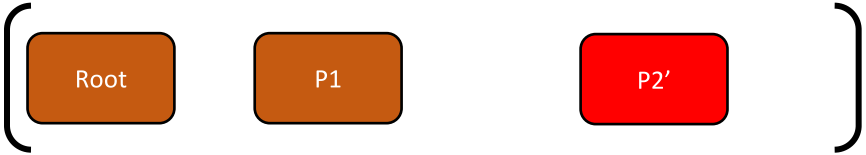


Authentication using Merkle Trees

- Client verifies the value by calculating the root hash from the value and Merkle proof

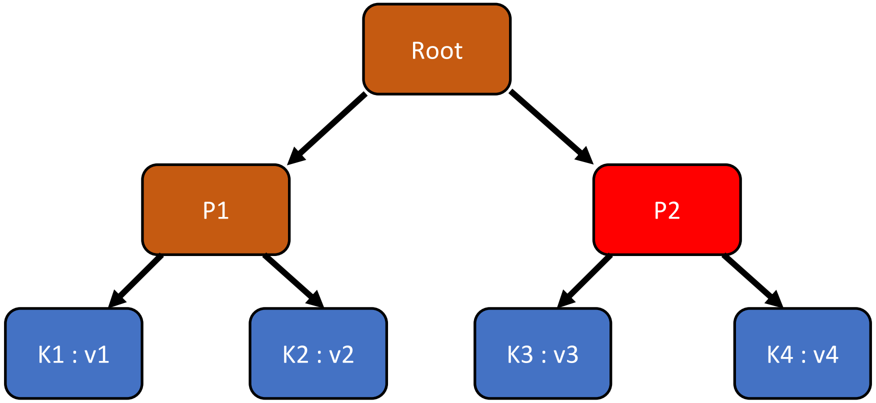


Response :

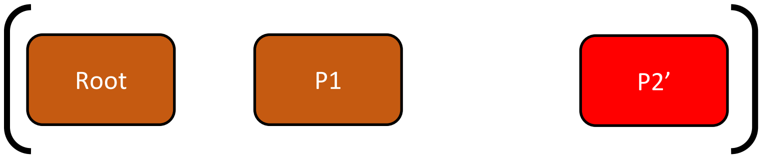


Authentication using Merkle Trees

- Client queries for value of key k4
- Server replies with value and a Merkle Proof

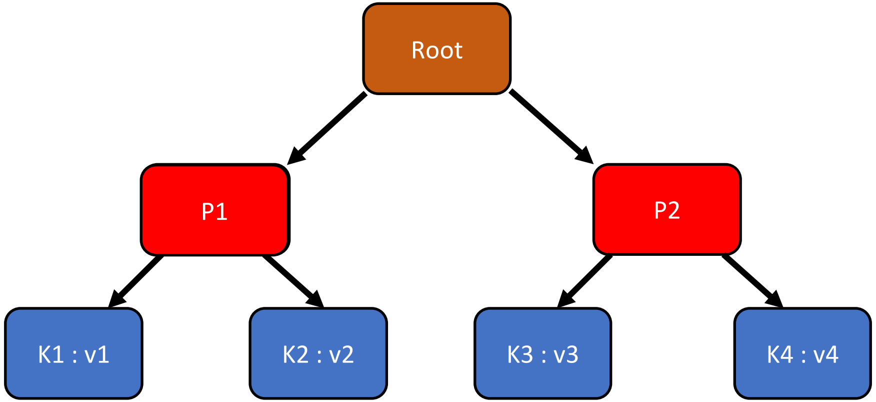


Response :

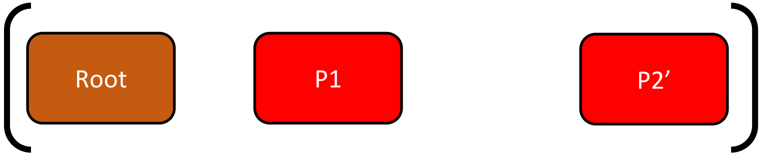


Authentication using Merkle Trees

- Client queries for value of key k4
- Server replies with value and a Merkle Proof

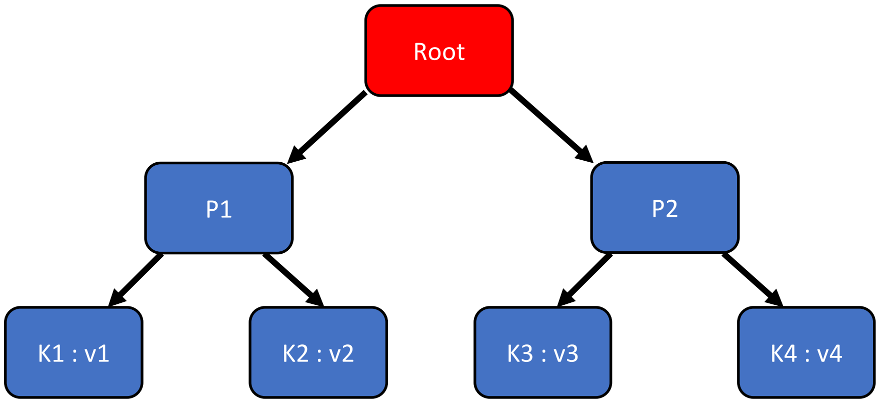


Response :

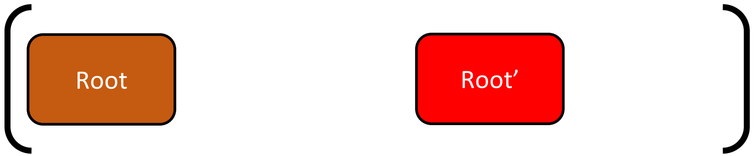


Authentication using Merkle Trees

- Client verifies the value by calculating the root hash from the value and Merkle proof

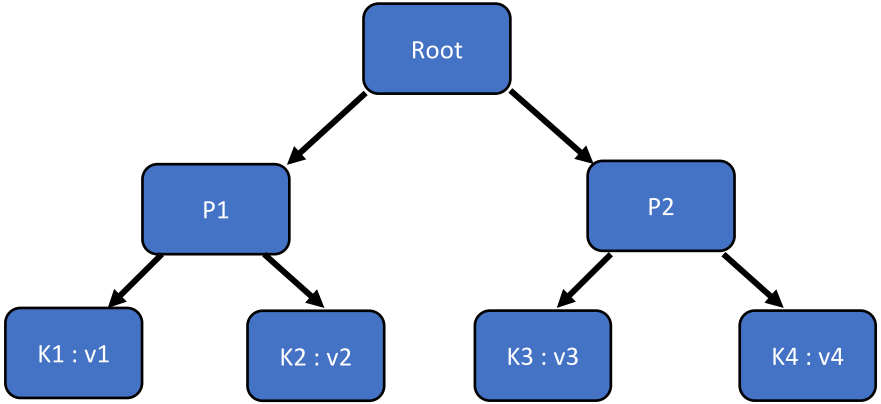


Response :

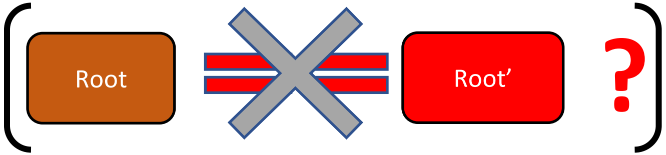


Authentication using Merkle Trees

- Server cannot lie about the value

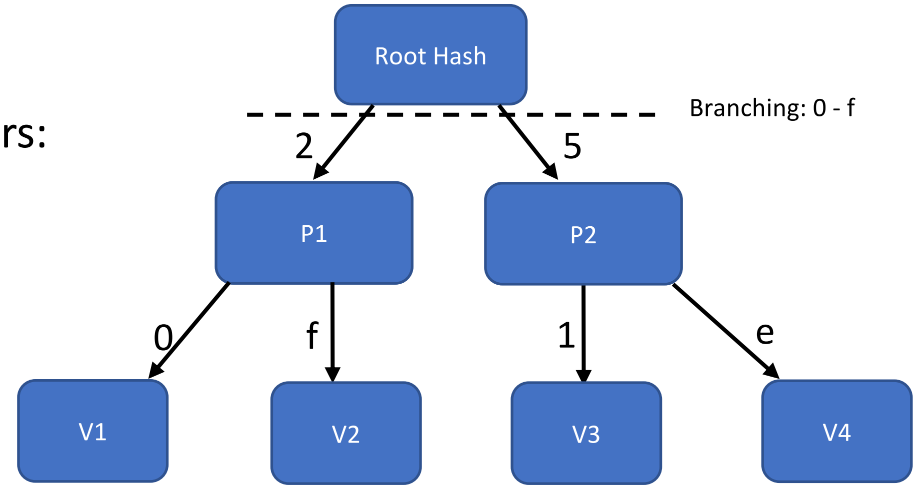


Response :



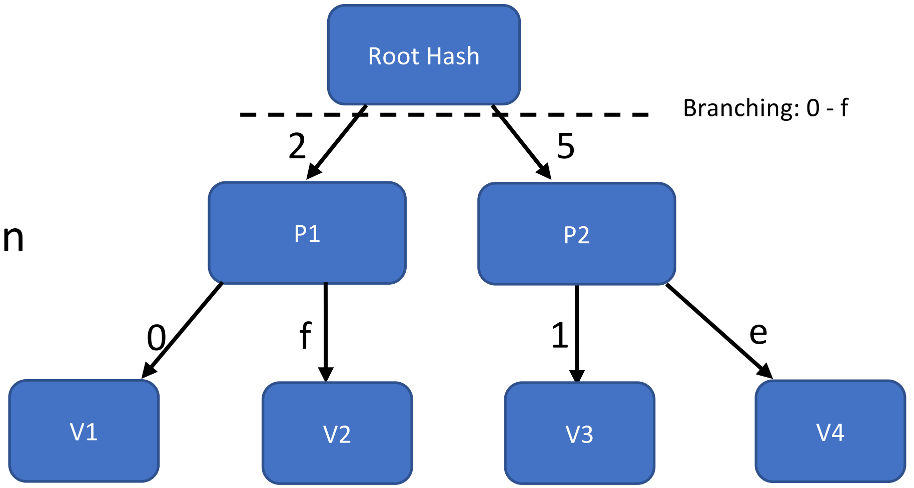
Merkle Patricia Trie

- Similar to Merkle trees
- Lookup based on the key structure
- Considering 4 bit hex key-value pairs:
  - 0x20 – V1
  - 0x2f – V2
  - 0x51 – V3
  - 0x5e – V4



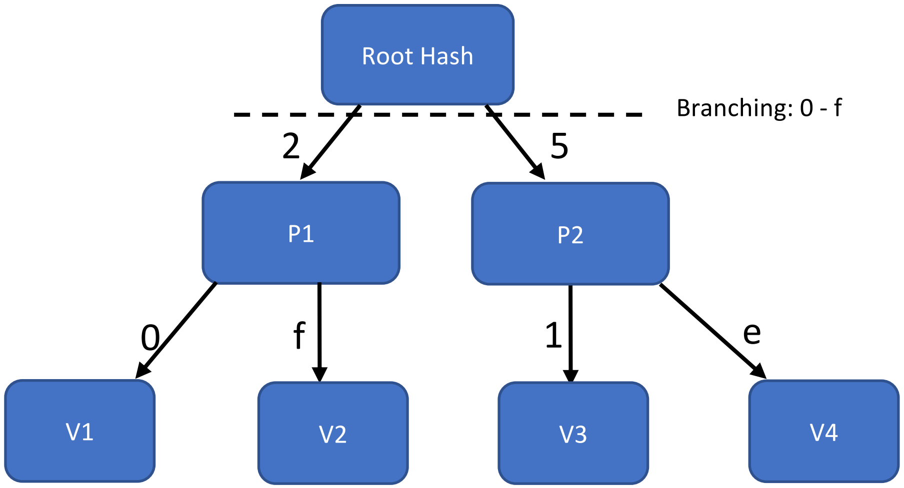
Authenticated Storage in Ethereum

- Trie is flattened and stored as key value pairs
- For every leaf node V, we store [Hash(V) -> V]
- For every parent node P, we have an [Hash(P) -> […]].



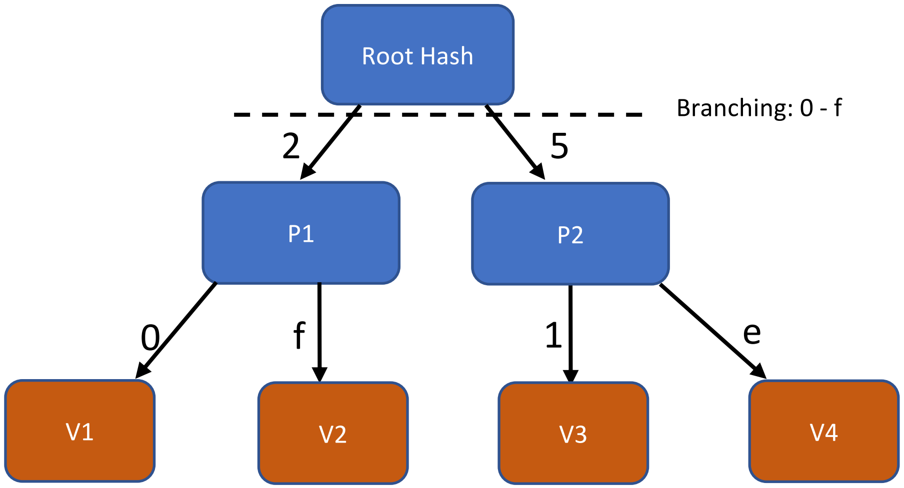
Authenticated Storage in Ethereum

KEY	VALUE



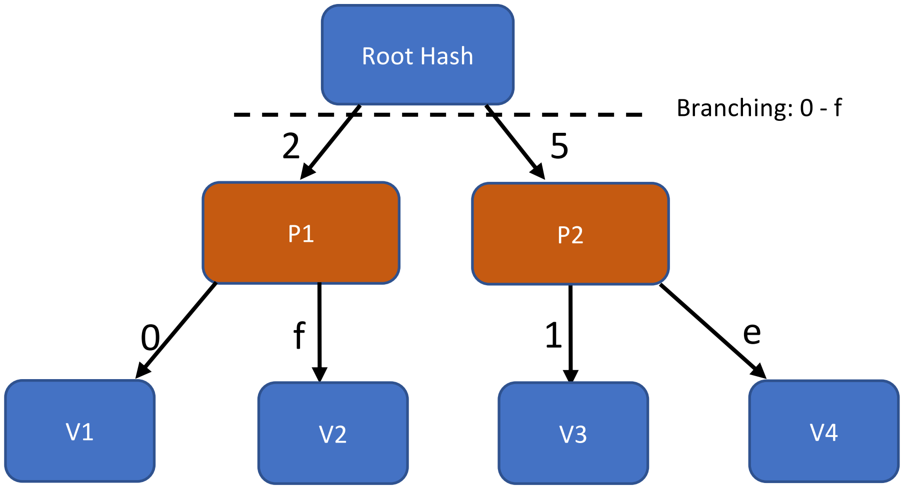
Authenticated Storage in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V2)	V2
Hash (V3)	V3
Hash (V4)	V4



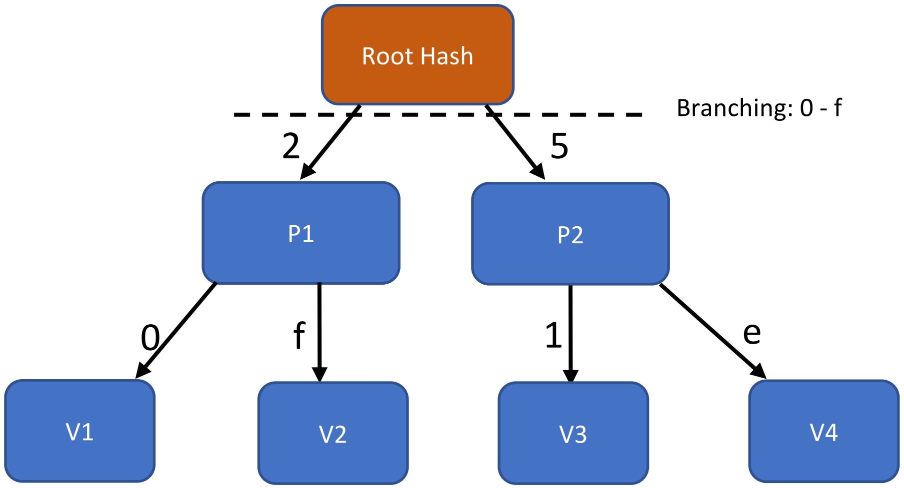
Authenticated Storage in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V2)	V2
Hash (V3)	V2
Hash (V4)	V3
Hash (P1)	Hash (V1), Hash (V2)
Hash (P2)	Hash (V3), Hash (V4)



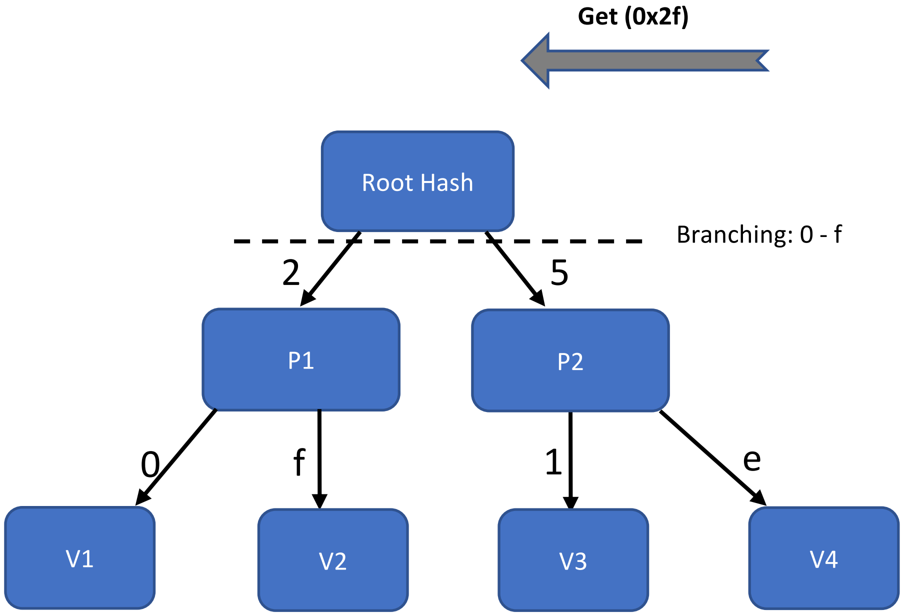
Authenticated Storage in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V2)	V2
Hash (V3)	V3
Hash (V4)	V4
Hash (P1)	Hash (V1), Hash (V2)
Hash (P2)	Hash (V3), Hash (V4)
Hash (RH)	Hash (P1), Hash (P2)



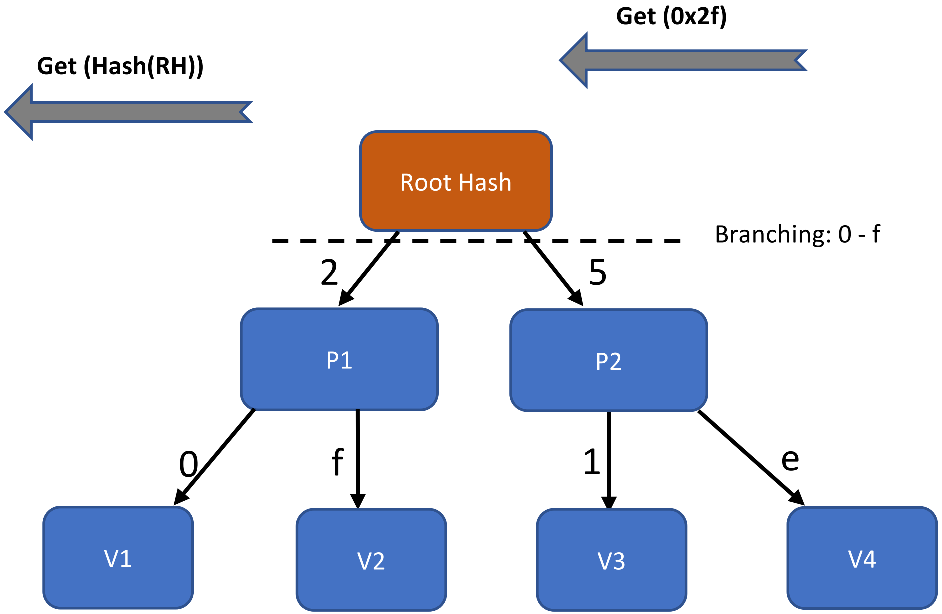
Read Amplification in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V2)	V2
Hash (V3)	V3
Hash (V4)	V4
Hash (P1)	Hash (V1), Hash (V2)
Hash (P2)	Hash (V3), Hash (V4)
Hash (RH)	Hash (P1), Hash (P2)



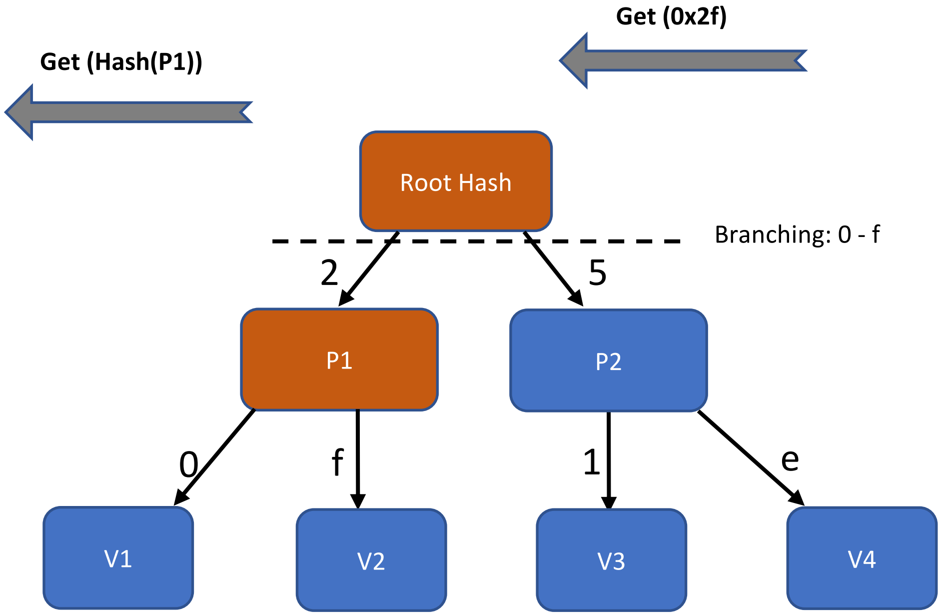
Read Amplification in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V2)	V2
Hash (V3)	V3
Hash (V4)	V4
Hash (P1)	Hash (V1), Hash (V2)
Hash (P2)	Hash (V3), Hash (V4)
Hash (RH)	Hash (P1), Hash (P2)



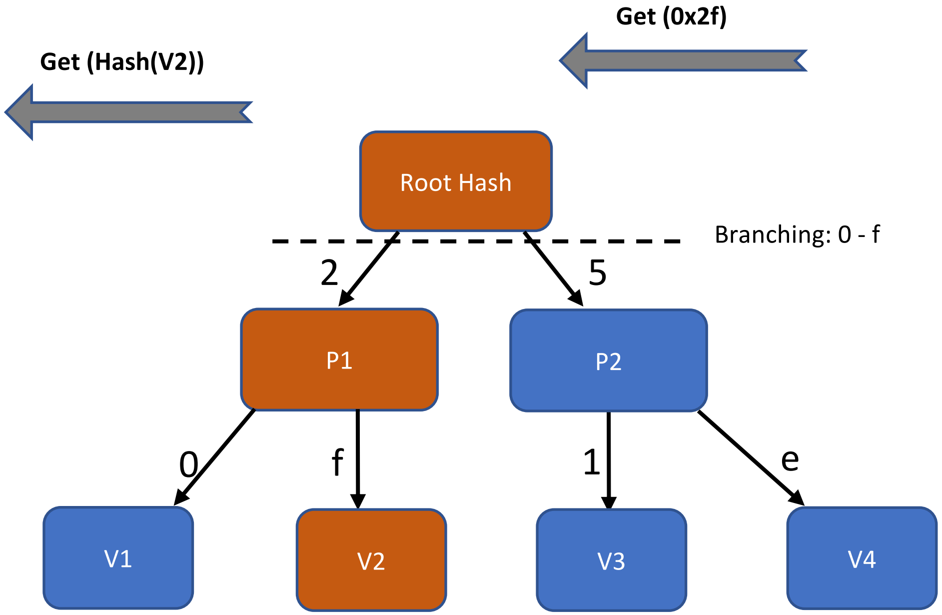
Read Amplification in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V2)	V2
Hash (V3)	V3
Hash (V4)	V4
Hash (P1)	Hash (V1), Hash (V2)
Hash (P2)	Hash (V3), Hash (V4)
Hash (RH)	Hash (P1), Hash (P2)



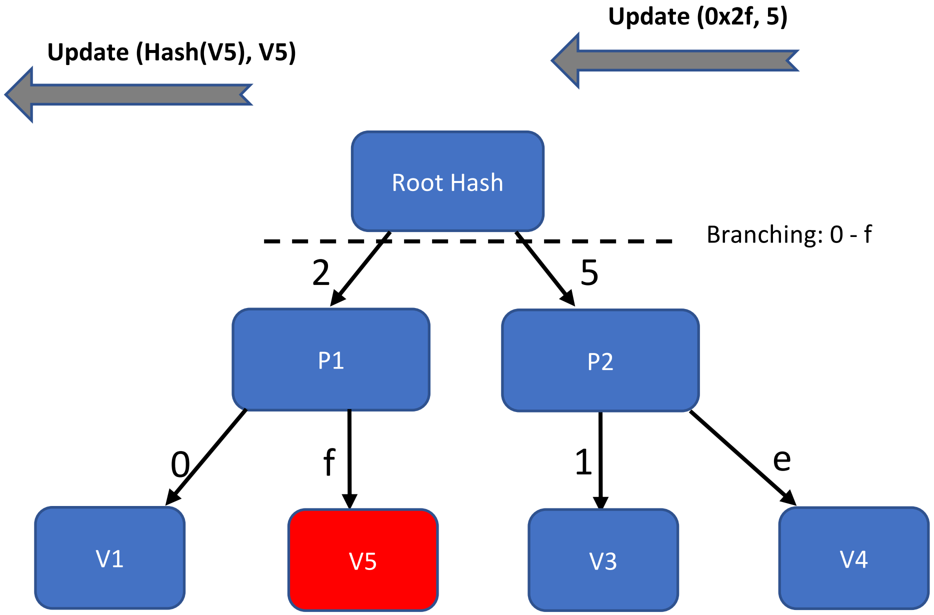
Read Amplification in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V2)	V2
Hash (V3)	V3
Hash (V4)	V4
Hash (P1)	Hash (V1), Hash (V2)
Hash (P2)	Hash (V3), Hash (V4)
Hash (RH)	Hash (P1), Hash (P2)



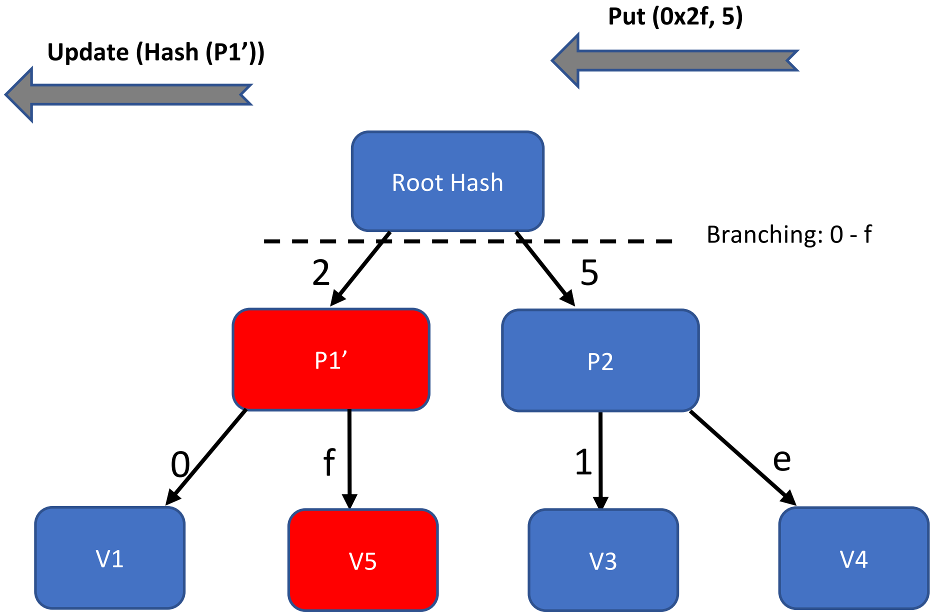
Write Amplification in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V5)	V5
Hash (V3)	V3
Hash (V4)	V4
Hash (P1)	Hash (V1), Hash (V2)
Hash (P2)	Hash (V3), Hash (V4)
Hash (RH)	Hash (P1), Hash (P2)



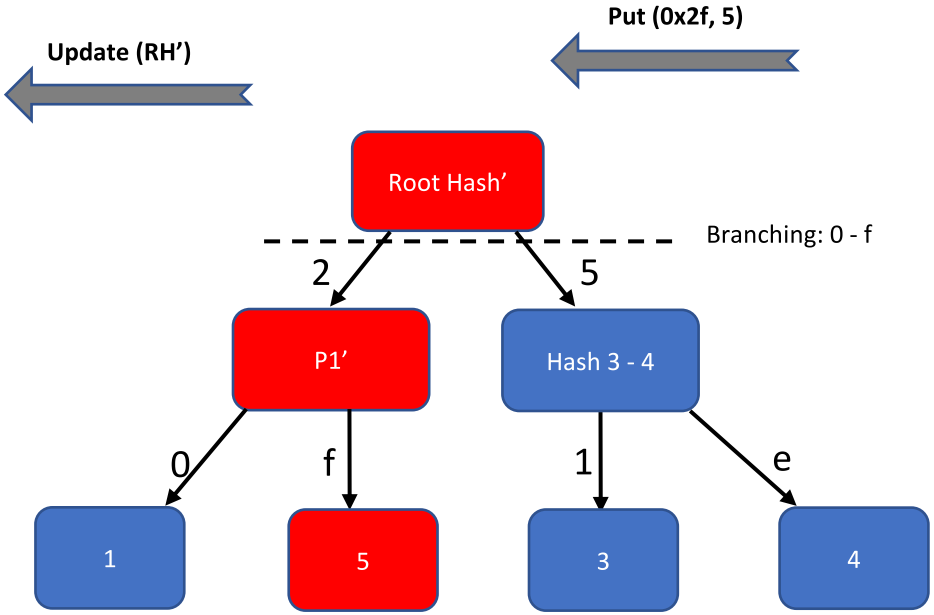
Write Amplification in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V5)	V5
Hash (V3)	V3
Hash (V4)	V4
Hash (P1’)	Hash (V1), Hash (V2)
Hash (P2)	Hash (V3), Hash (V4)
Hash (RH)	Hash (P1), Hash (P2)



Write Amplification in Ethereum

KEY	VALUE
Hash (V1)	V1
Hash (V5)	V5
Hash (V3)	V3
Hash (V4)	V4
Hash (P1’)	Hash (V1), Hash (V2)
Hash (P2)	Hash (V3), Hash (V4)
Hash (RH’)	Hash (P1’), Hash (P2)



Experimental Setup

- Private Ethereum network
- Importing first 1.6 M blocks of the real-world public block chain
- geth - Ethereum go client
- Machine
  - 16 GB of RAM
  - 2TB Intel 750 series SSD

IO Amplification in Ethereum

- State Trie – **7X IO Amplification**
- getBalance (addr)
  - Returns the amount of ether balance present in the account addr
  - 0.22M account addresses
  - 1.4M LevelDB gets

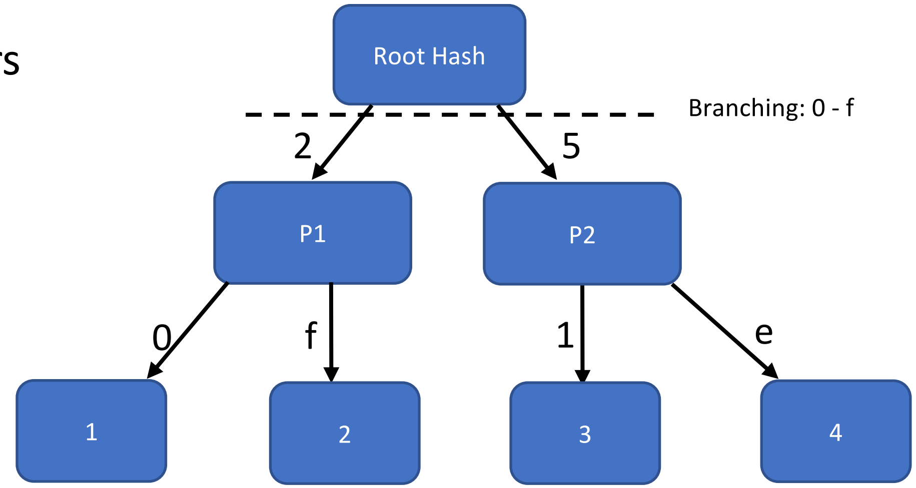
IO Amplification in Ethereum

- State Trie – **7X IO Amplification**
- Worst case – **64X IO Amplification**
  - Key : 256 bits
  - Every node : 4 bits
  - Depth of Trie : 64 in the worst case
- **Ignoring the IO Amplification introduced by underlying kv store**
- Considers the first 1.6M blocks of the block chain
  - **Current size of blockchain : 5.9M blocks**

Caching - Why doesn’t it work?

Caching key with value, proof

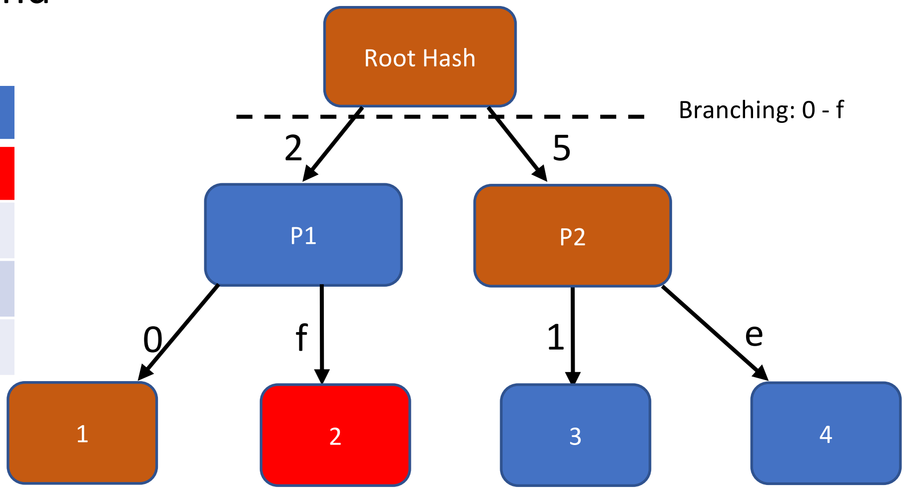
- Going back to our example
- For a 4 bit hex string key-value pairs
  - 0x20 – 1
  - 0x2f – 2
  - 0x51 – 3
  - 0x5e – 4



Caching key with value, proof

- For every key, we cache the value and the Merkle Proof

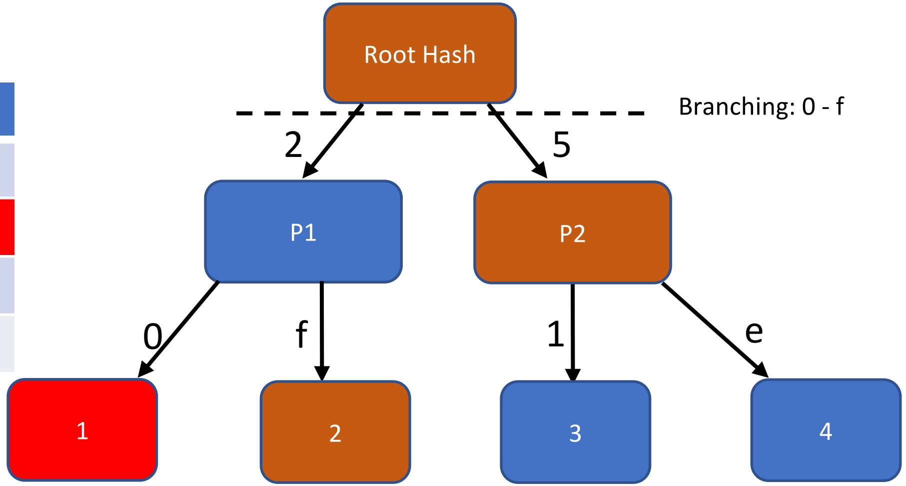
Key	Value	Proof
0x2f	2	[1, P2, Root Hash]



Caching key with value, proof

- For every key, we cache the value and the Merkle Proof

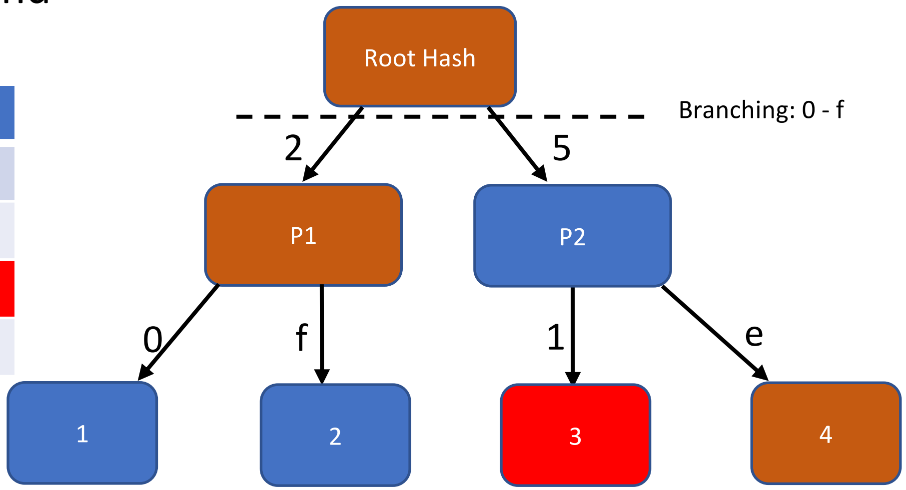
Key	Value	Proof
0x2f	2	[1, P2, Root Hash]
0x20	1	[2, P2, Root Hash]



Caching key with value, proof

- For every key, we cache the value and the Merkle Proof

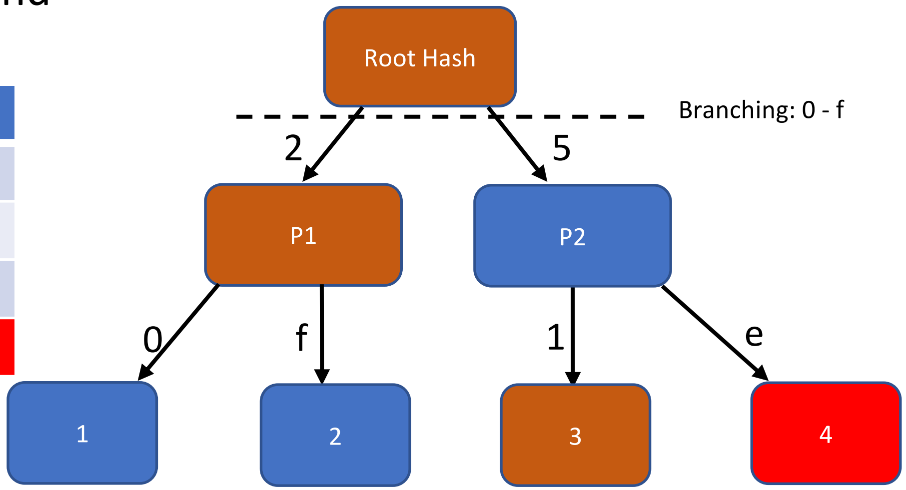
Key	Value	Proof
0x2f	2	[1, P2, Root Hash]
0x20	1	[2, P2, Root Hash]
0x51	3	[4, P1, Root Hash]



Caching key with value, proof

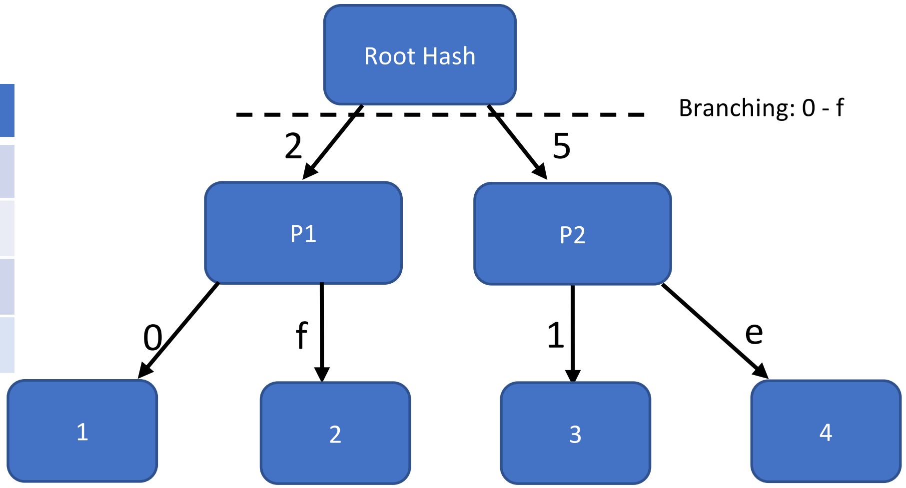
- For every key, we cache the value and the Merkle Proof

Key	Value	Proof
0x2f	2	[1, P2, Root Hash]
0x20	1	[2, P2, Root Hash]
0x51	3	[4, P1, Root Hash]
0x5e	4	[3, P1, Root Hash]



A single update invalidates the whole cache

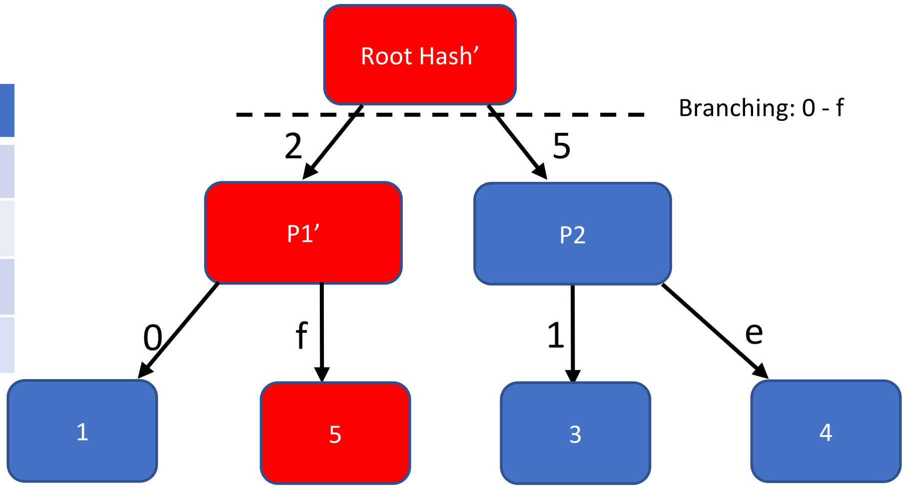
Key	Value	Proof
0x2f	2	[1, P2, Root Hash]
0x20	1	[2, P2, Root Hash]
0x51	3	[4, P1, Root Hash]
0x5e	4	[3, P1, Root Hash]



Reads can be served from the cache

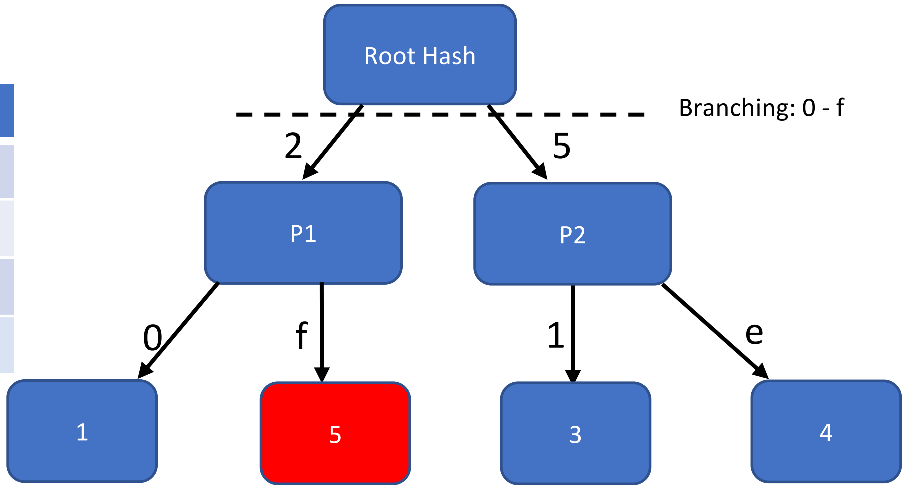
A single update invalidates the whole cache

Key	Value	Proof
0x2f	2	[1, P2, Root Hash]
0x20	1	[2, P2, Root Hash]
0x51	3	[4, P1, Root Hash]
0x5e	4	[3, P1, Root Hash]



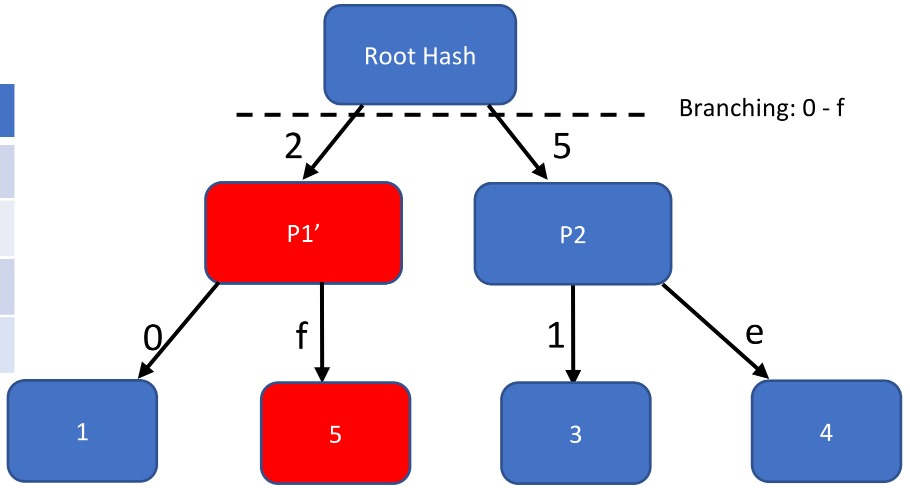
A single update invalidates the whole cache

Key	Value	Proof
0x2f	5	[1, P2, Root Hash]
0x20	1	[2, P2, Root Hash]
0x51	3	[4, P1, Root Hash]
0x5e	4	[3, P1, Root Hash]



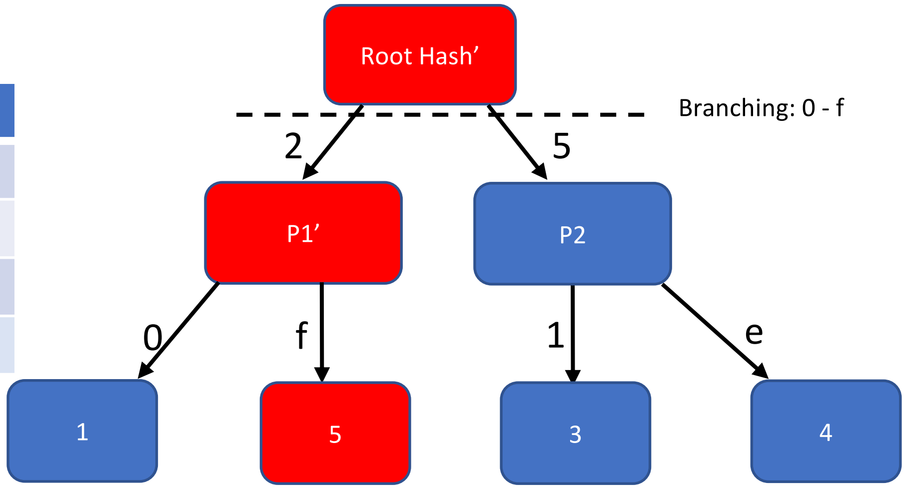
A single update invalidates the whole cache

Key	Value	Proof
0x2f	5	[1, P2, Root Hash]
0x20	1	[2, P2, Root Hash]
0x51	3	[4, P1’, Root Hash]
0x5e	4	[3, P1’, Root Hash]



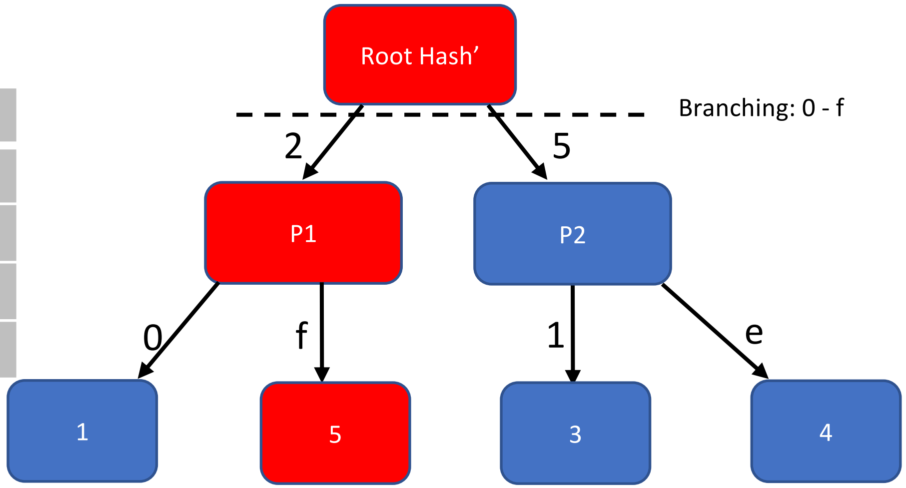
A single update invalidates the whole cache

Key	Value	Proof
0x2f	5	[1, P2, Root Hash’]
0x20	1	[2, P2, Root Hash’]
0x51	3	[4, P1’, Root Hash’]
0x5e	4	[3, P1’, Root Hash’]



A single update invalidates the whole cache

Key	Value	Proof
0x2f	5	[1, P2, Root Hash’]
0x20	1	[2, P2, Root Hash’]
0x51	3	[4, P1’, Root Hash’]
0x5e	4	[3, P1’, Root Hash’]



Works only for read-only workloads

Merkelized LSM

Why caching didn't work?

- Tight coupling between any two nodes in the tree
  - All nodes form a single tree under the same root node
- Tight coupling between Lookup and Authentication
  - Lookup for a value is done traversing the authenticated data structure

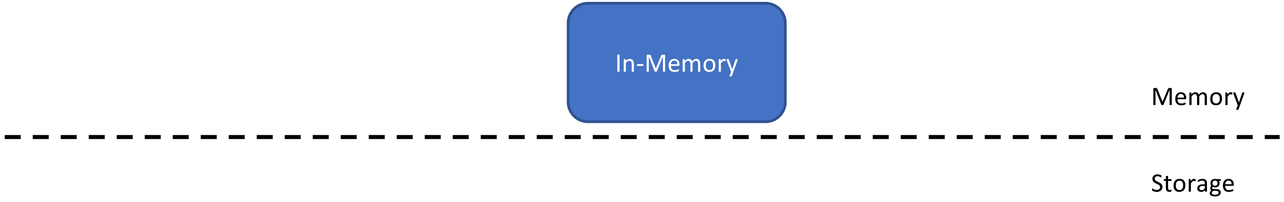
Insights behind mLSM

Maintaining Multiple Independent structures

Decoupling Lookup from Authentication

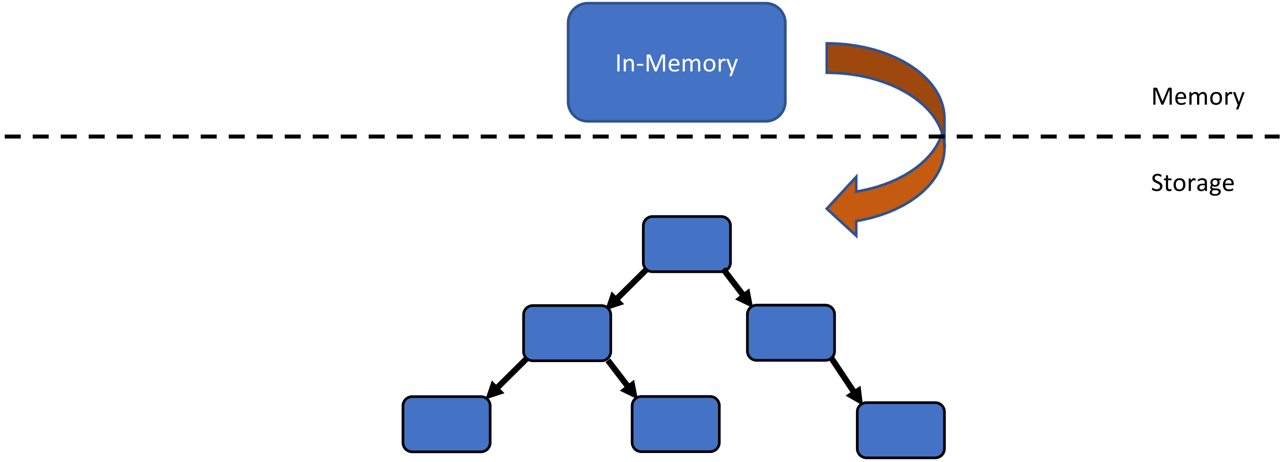
Maintaining multiple independent structures

Merkelized LSM : Design



In-memory and On-disk layers

Merkelized Log Structured Merge Tree (mLSM)

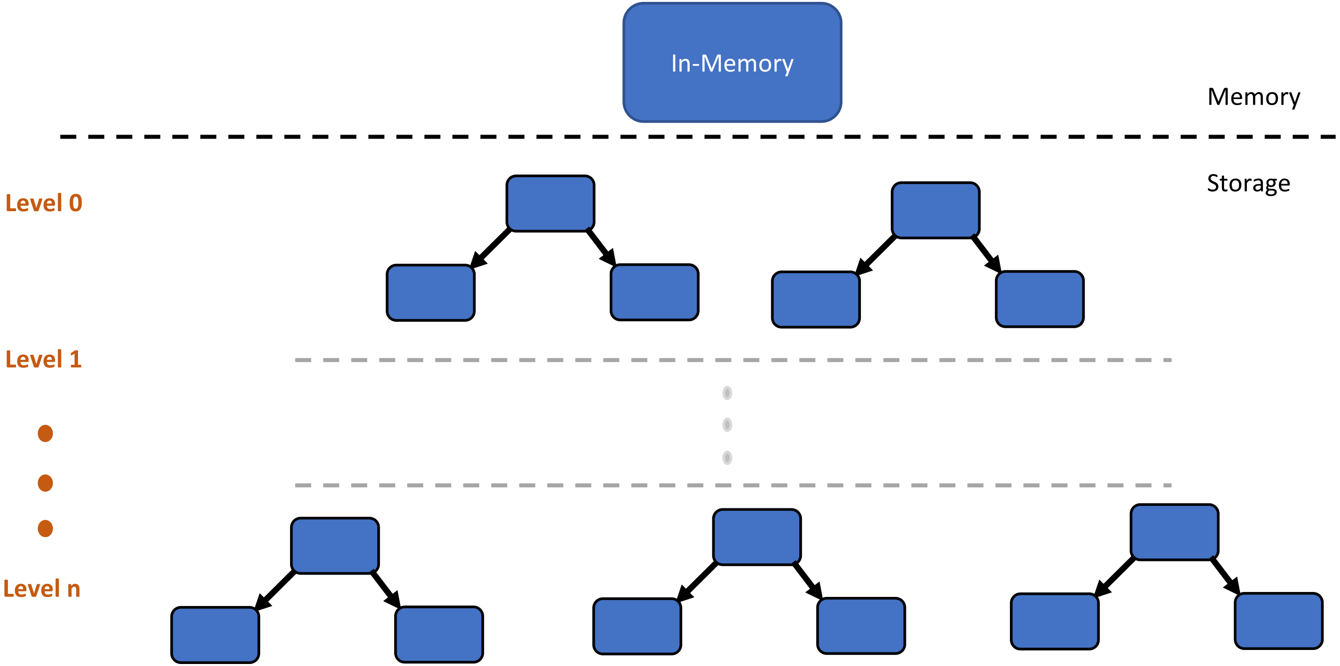


In memory data is periodically written as binary Merkle trees to storage

Merkelized LSM : Design

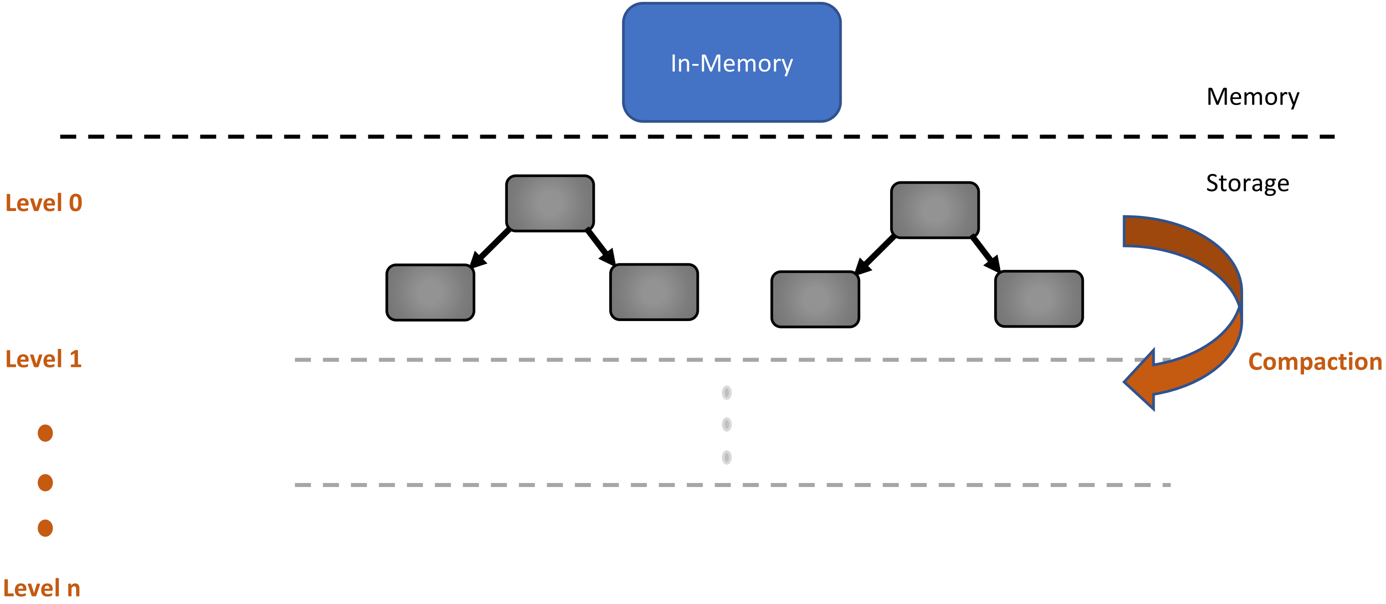
- Binary Merkle Trees
  - Reduce the size of the Merkle Proof
  - Balance data better than Tries

Merkelized Log Structured Merge Tree (mLSM)



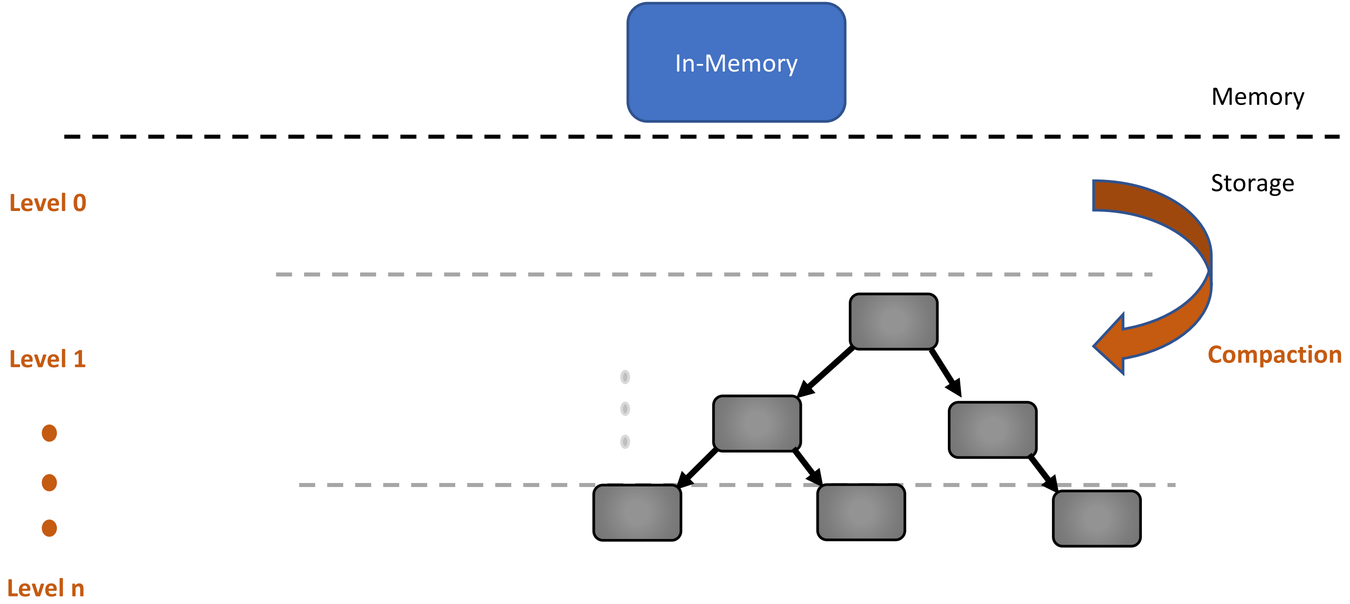
Merkle Trees on storage are logically arranged in different levels

Merkelized Log Structured Merge Tree (mLSM)



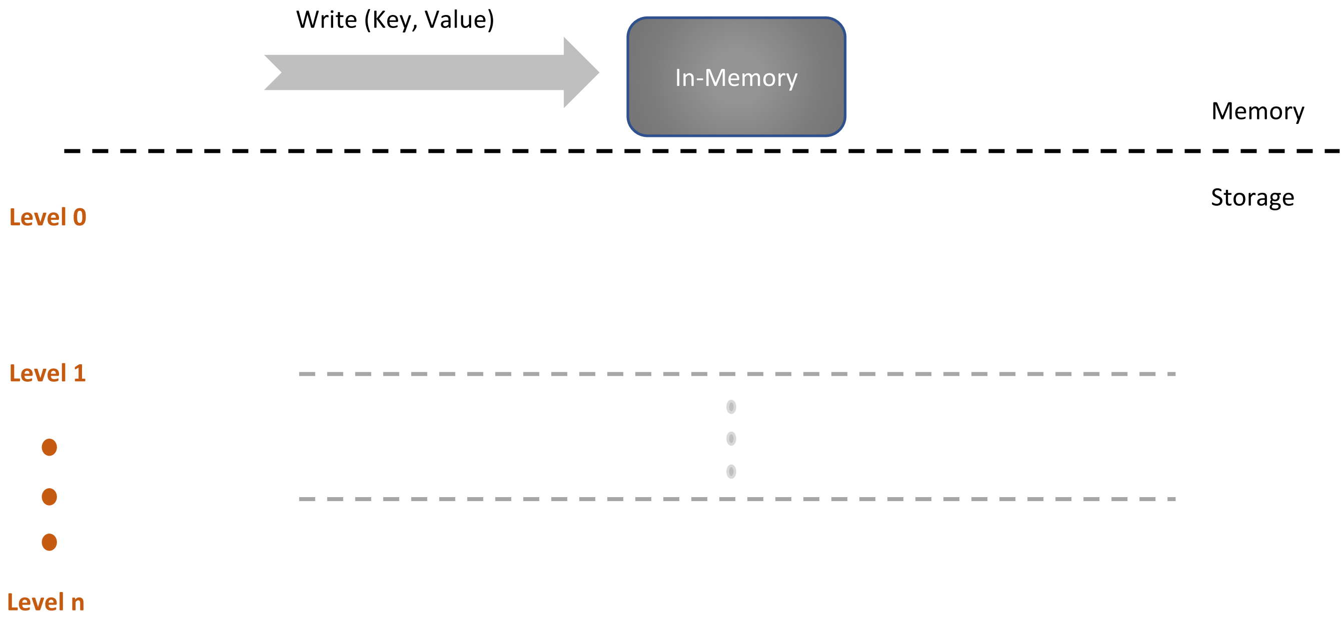
Compaction is performed once #Trees in a level reaches a threshold

Merkelized Log Structured Merge Tree (mLSM)



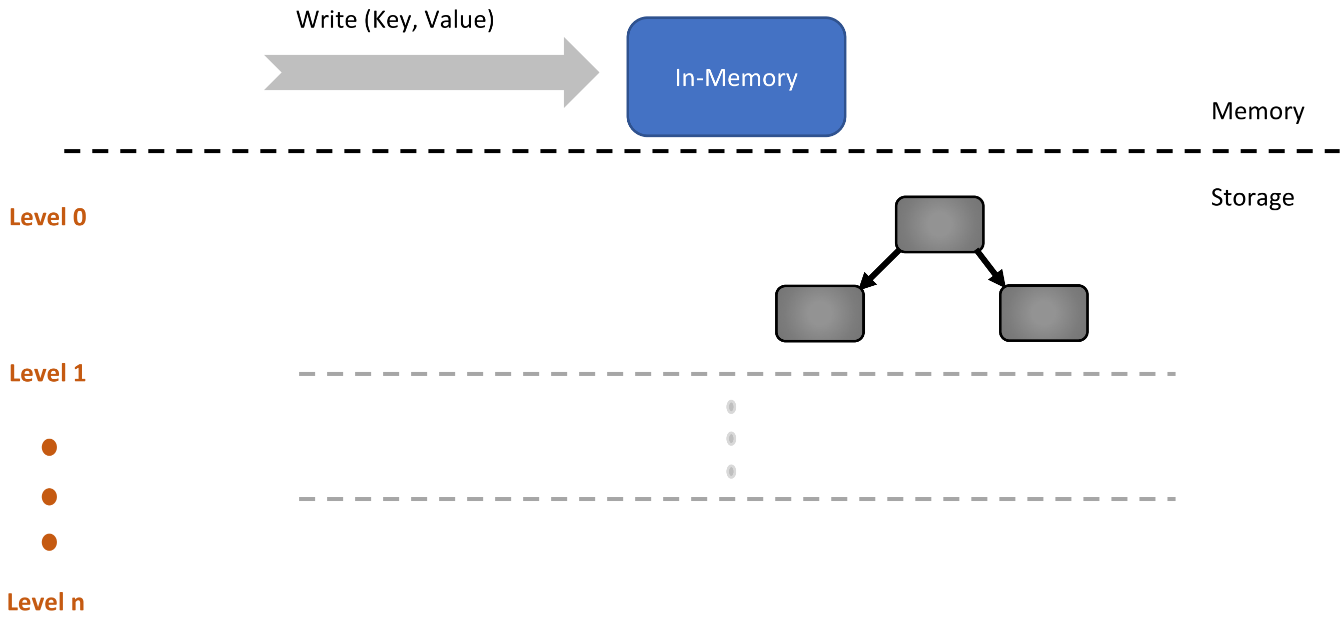
Compaction is performed once #Trees in a level reaches a threshold

Writes in Merkelized LSM



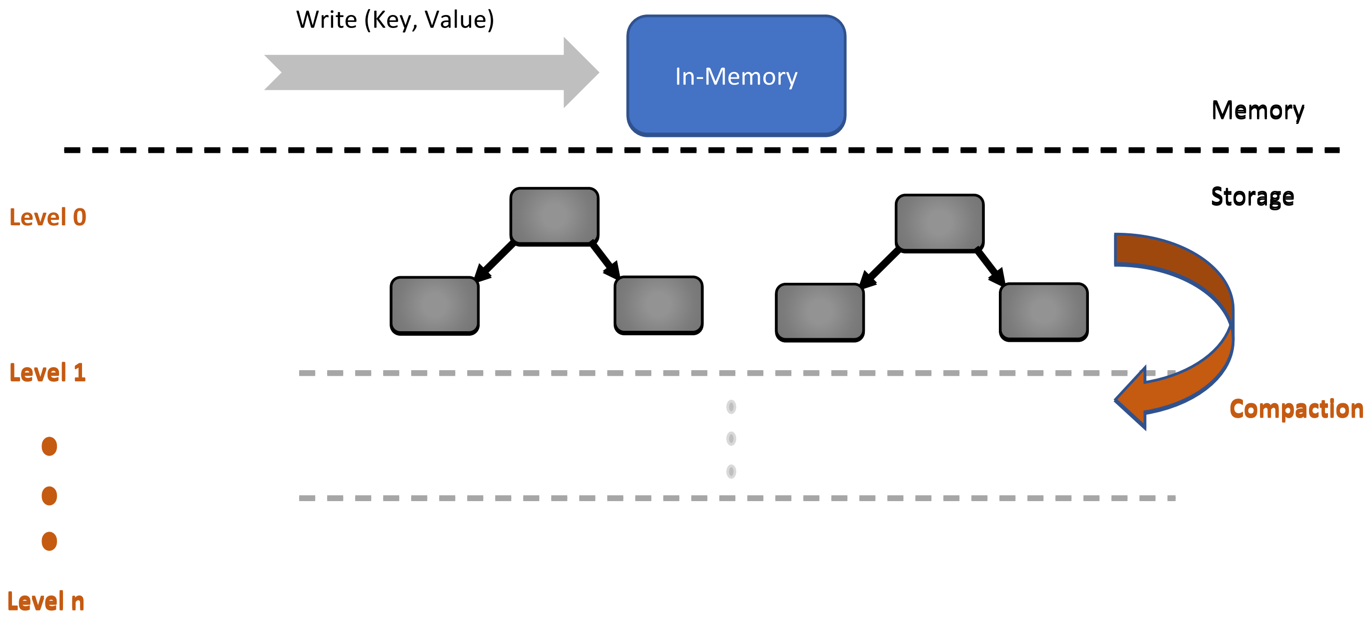
Writes are handled in-memory

Writes in Merkelized LSM



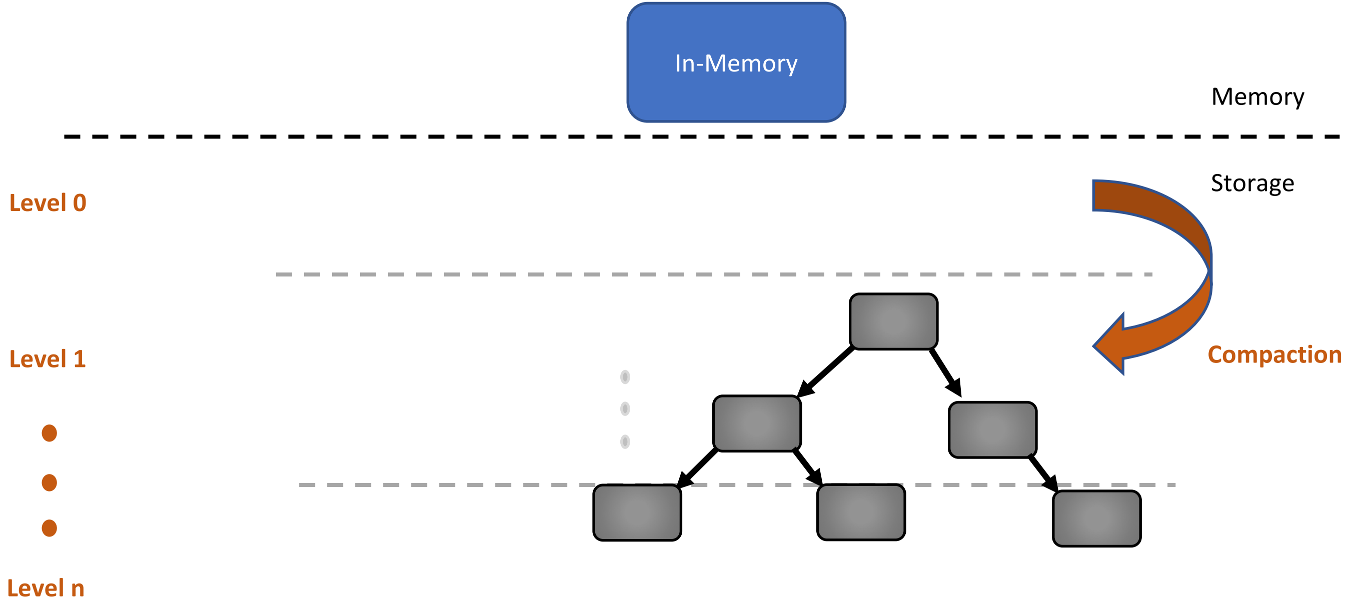
Writes are batched and written onto storage

Writes in Merkelized LSM



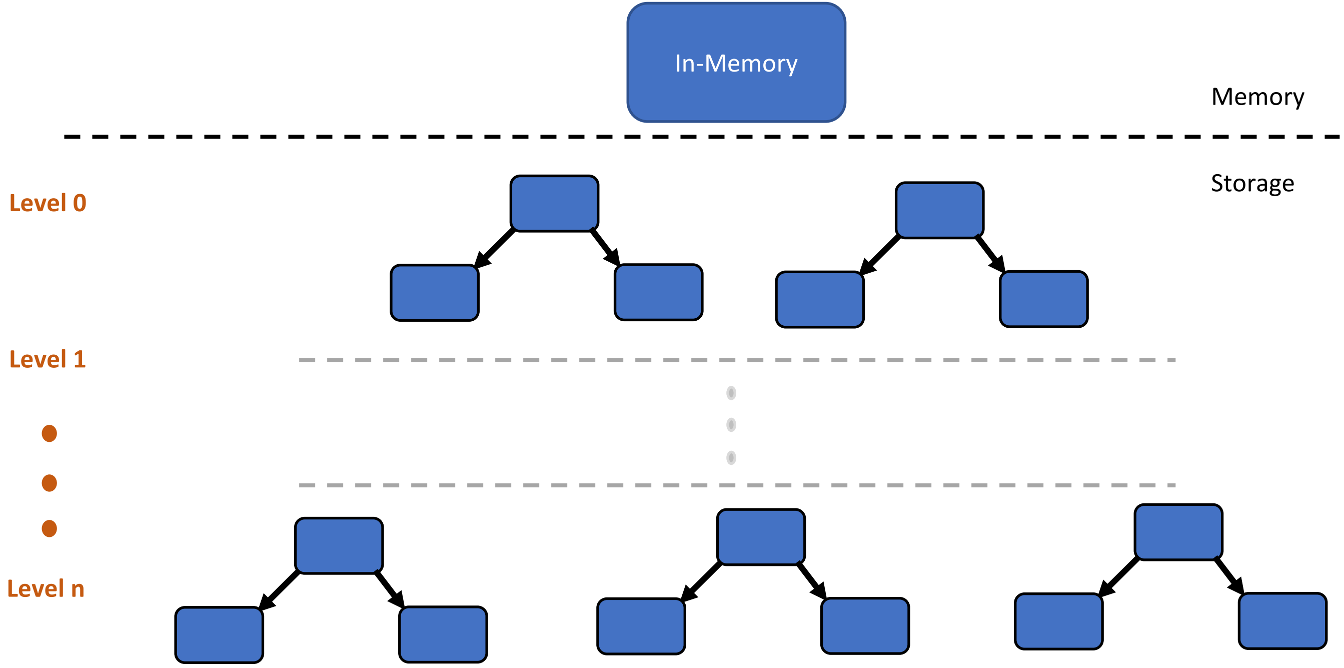
Numbers of files on reaching the threshold at the level

Writes in Merkelized LSM

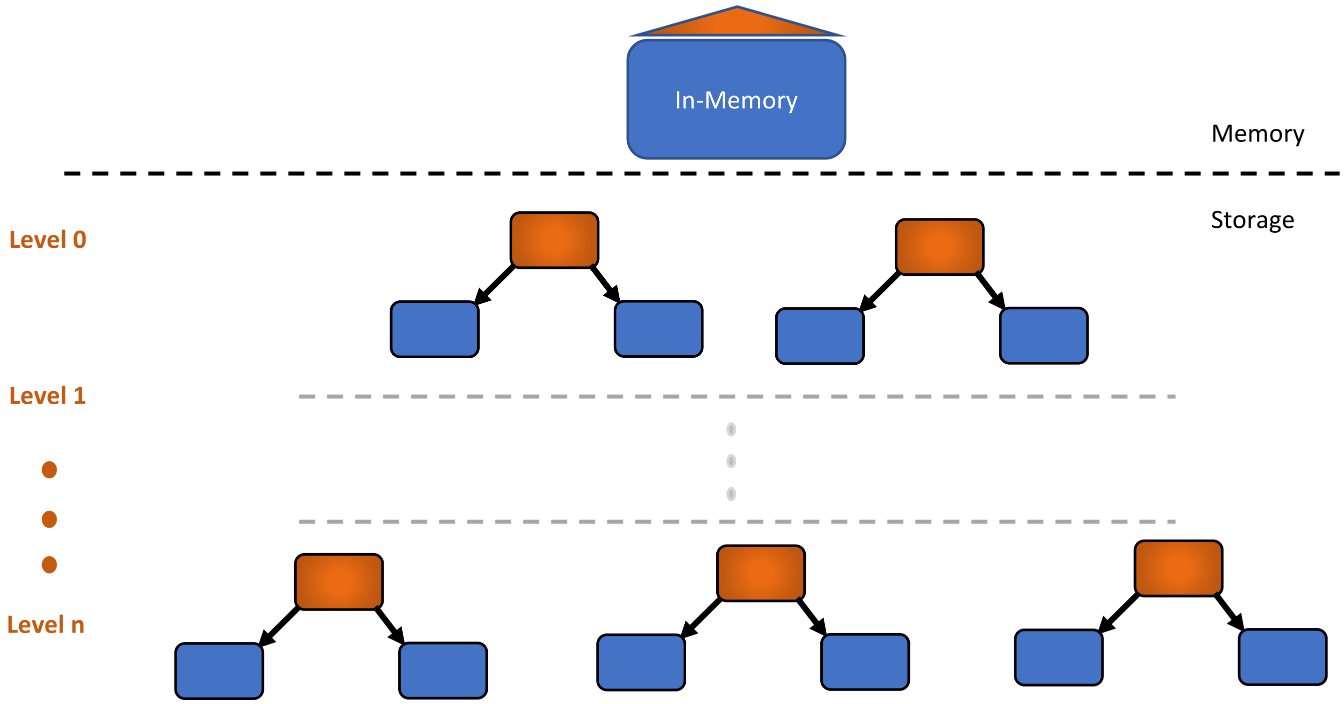


Compaction is performed from lower levels to higher levels

Authentication in mLSM

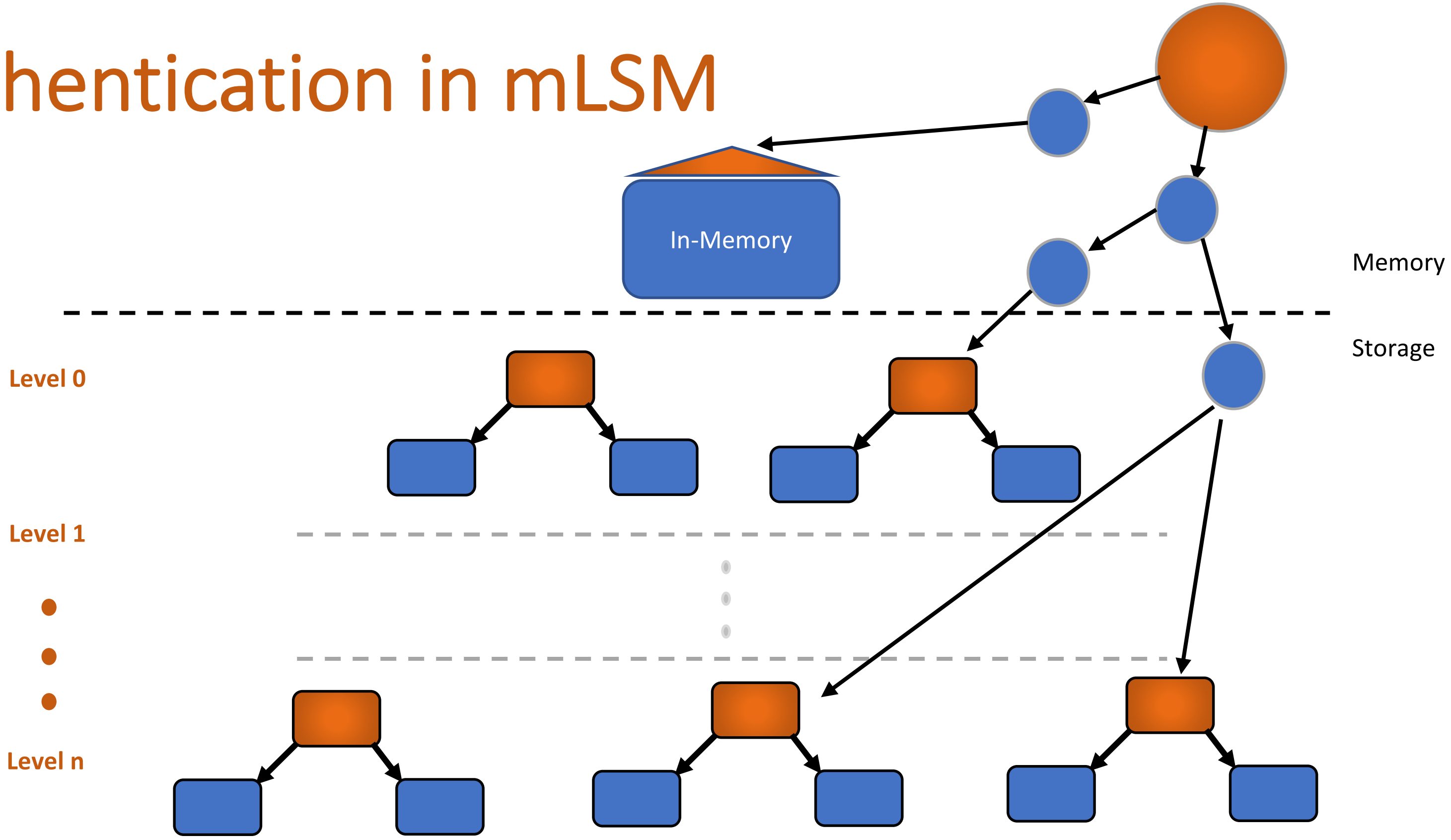


Authentication in mLSM



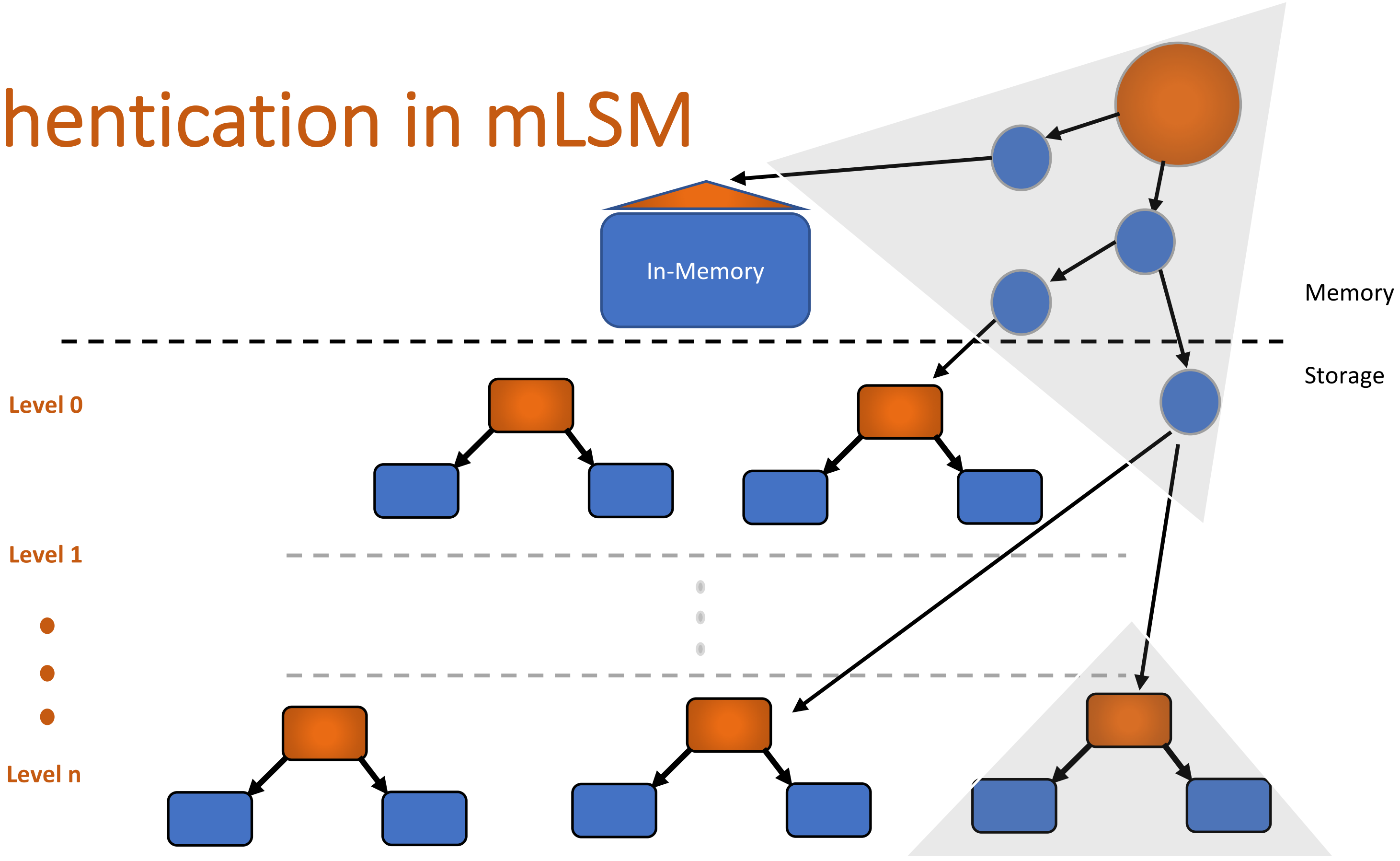
Every binary merkle tree on level has a local root

Authentication in mLSM



Global Master Root dynamically computes global Merkle Tree

Authentication in mLSM



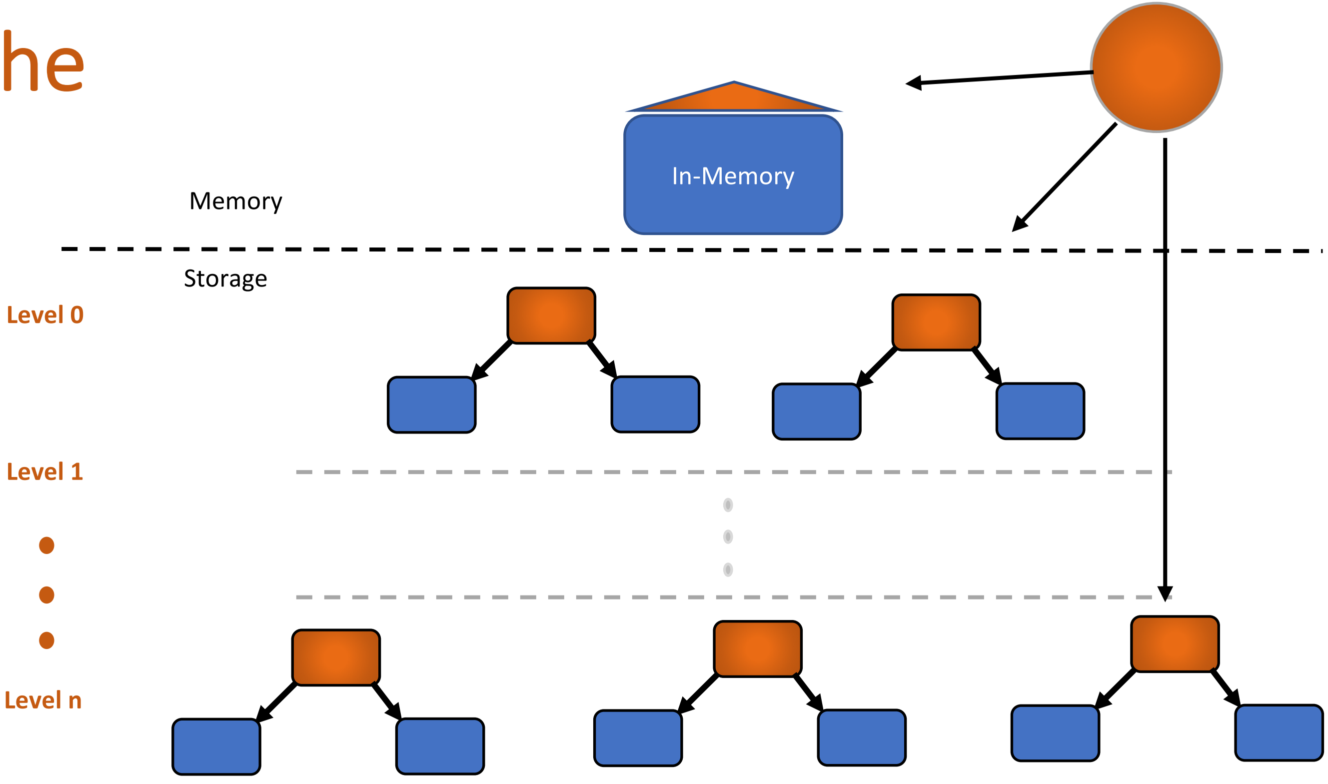
Merkle Proof includes the local and the global Merkle proofs

Decoupling lookup from Authentication

LevelDB Cache

LevelDB cache

Key, Level	Value, Proof

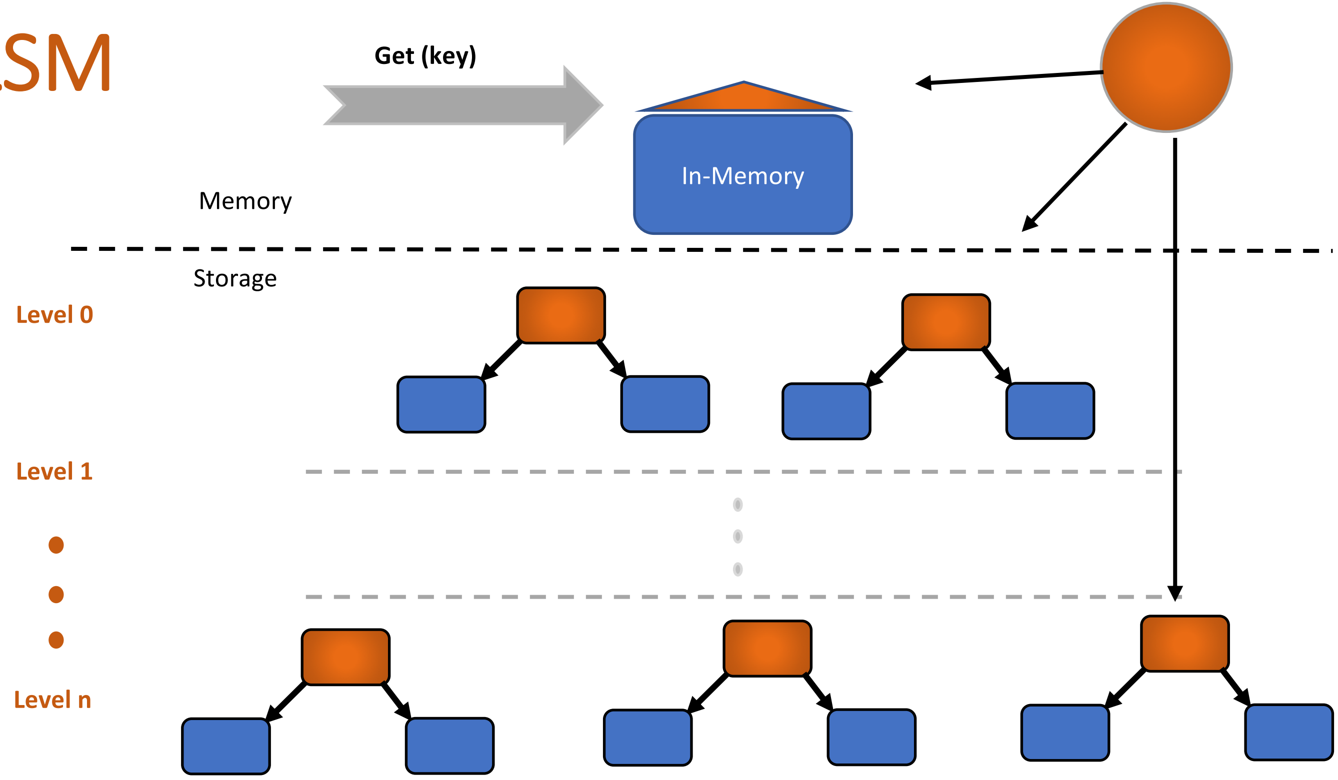


LevelDB cache to store (Key, Level : Value, Merkle Proof)

Reads in mLSM

LevelDB cache

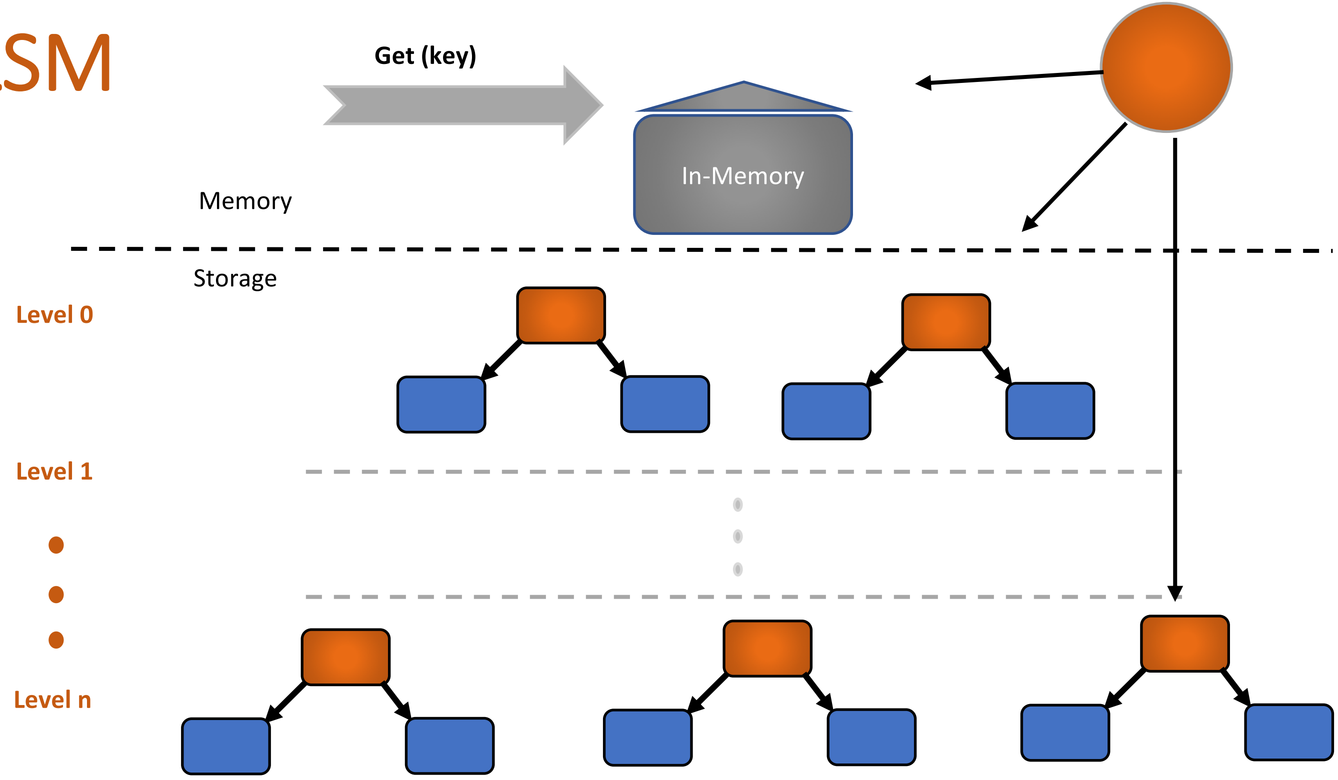
Key, Level	Value, Proof



Reads in mLSM

LevelDB cache

Key, Level	Value, Proof

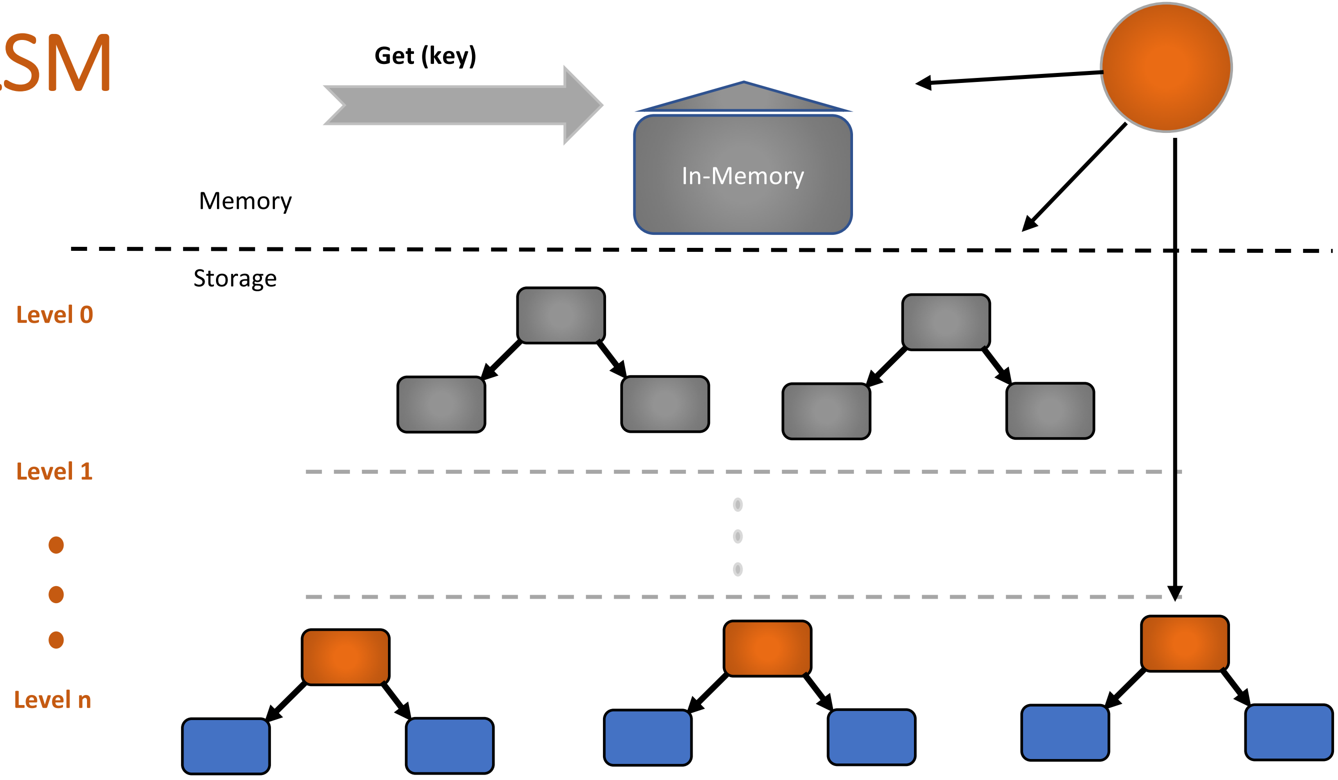


In-memory structure is searched for the value

Reads in mLSM

LevelDB cache

Key, Level	Value, Proof

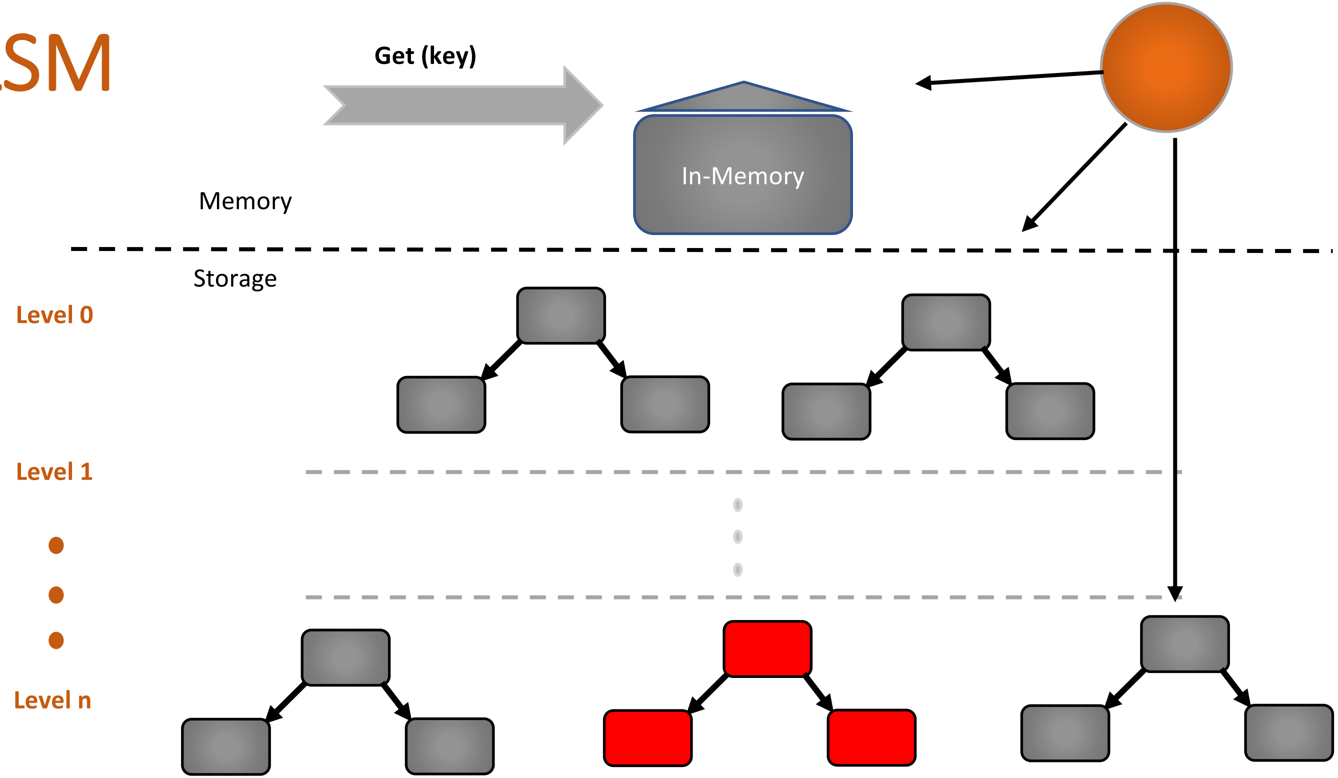


mLSM is traversed level by level in-order

Reads in mLSM

LevelDB cache

Key, Level	Value, Proof

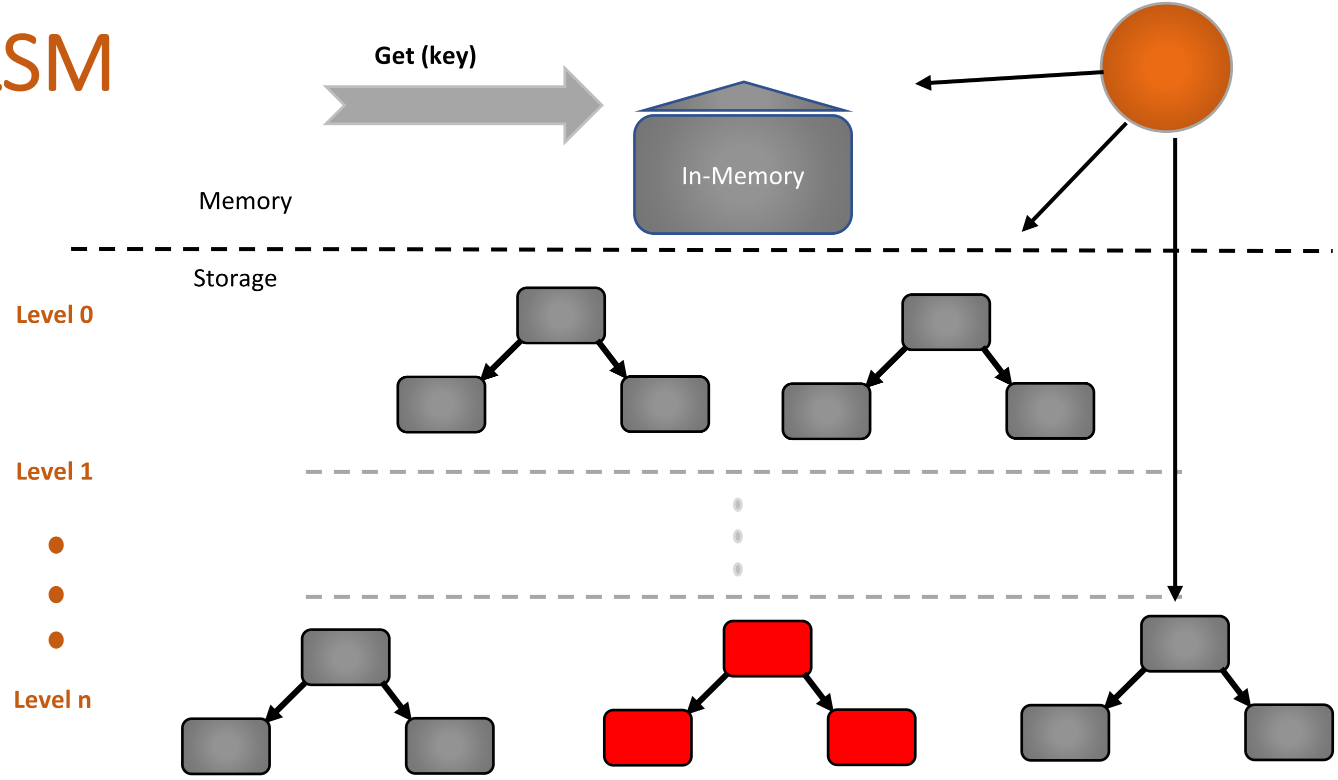


First occurrence of the key value pair is returned

Reads in mLSM

LevelDB cache

Key, Level	**Value, Proof**
key, Level	value, Local proof

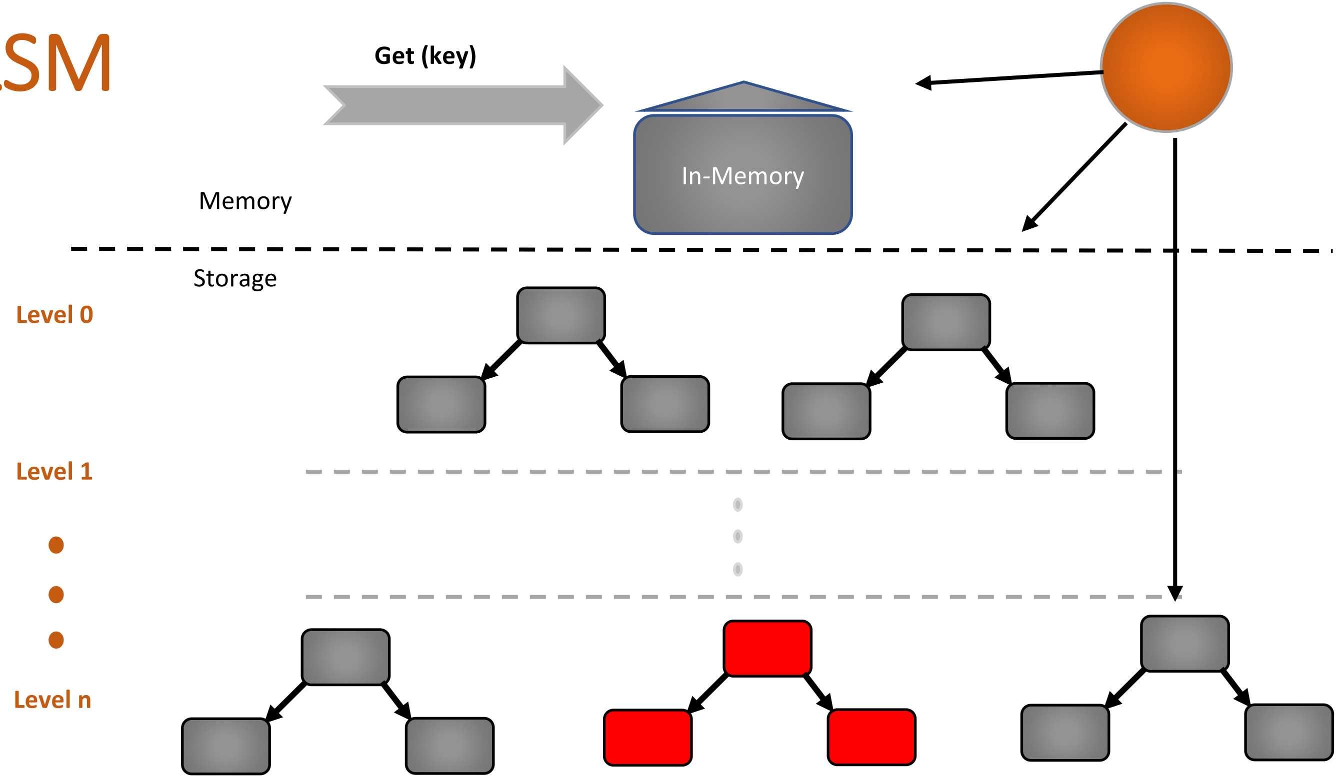


<Key, level : value, local Merkle proof> are cached

Reads in mLSM

LevelDB cache

Key, Level	Value, Proof
key, Level	value, Local Proof



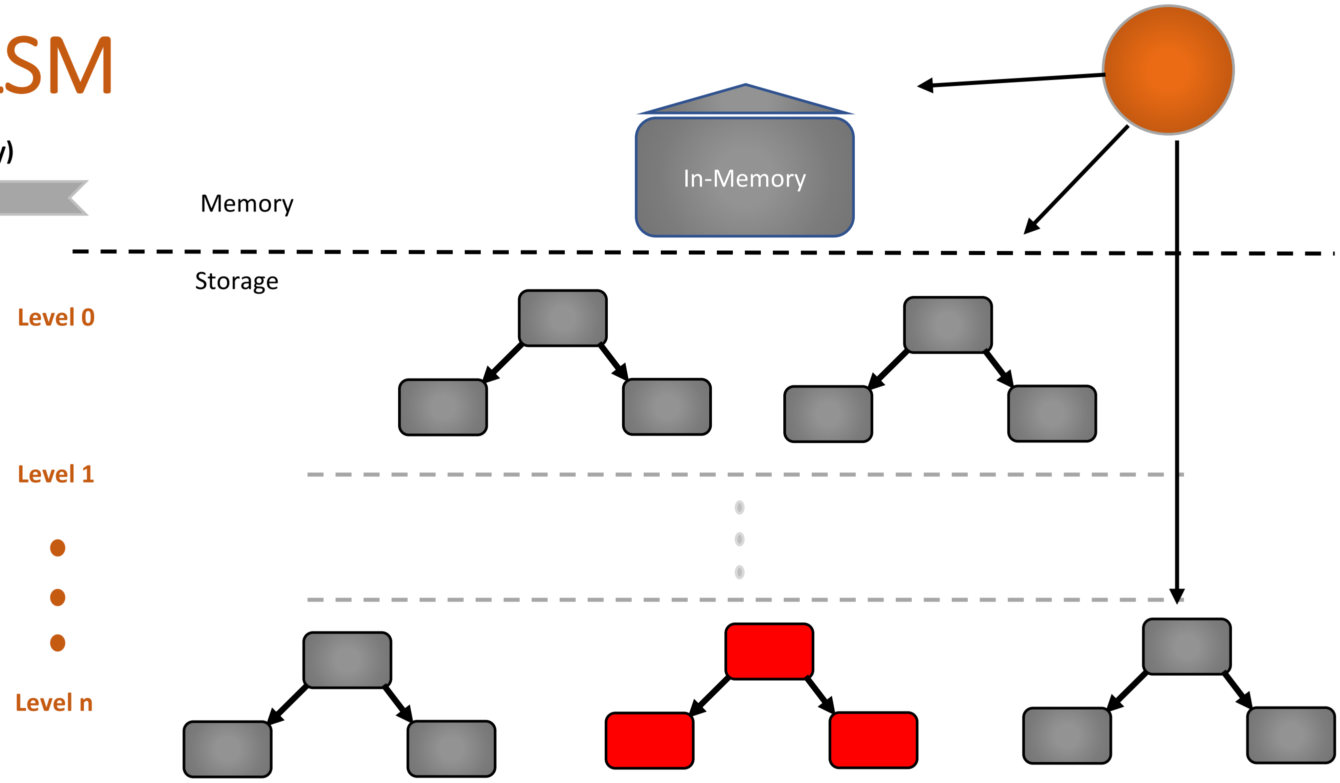
NOTE: Global Proof is not cached

Reads in mLSM

Get (key)

LevelDB cache

Key, Level	**Value, Proof**
key, Level	value, Local Proof

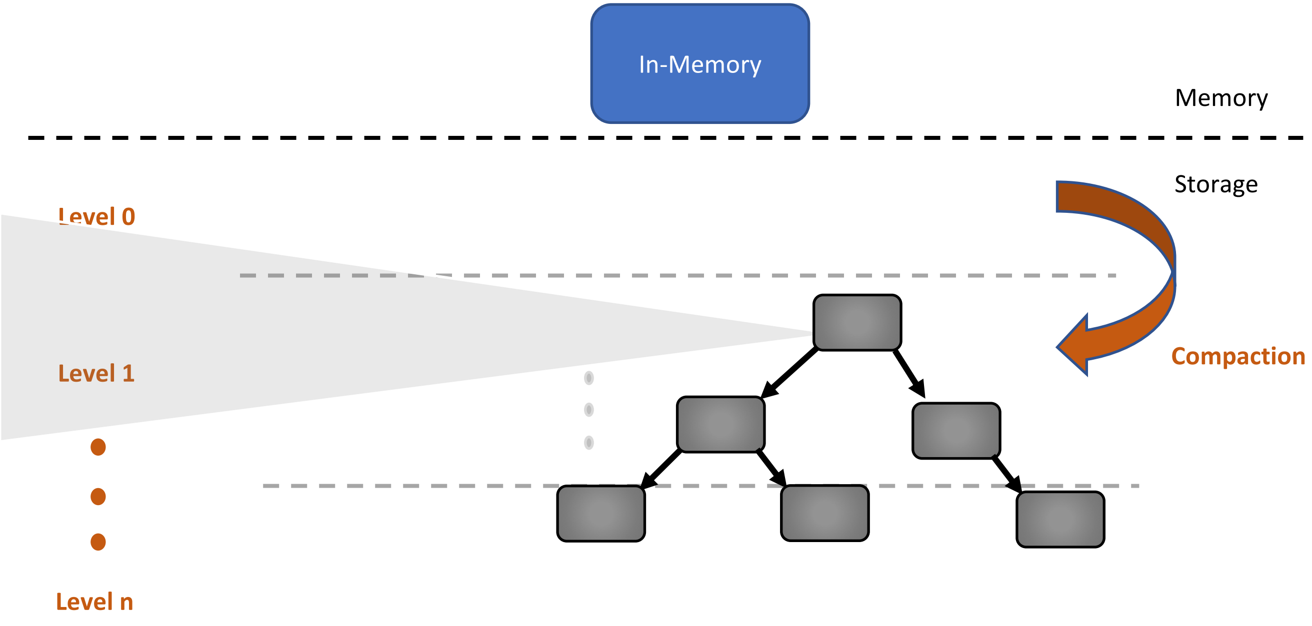


Subsequent reads are served from the cache

Reads in mLSM

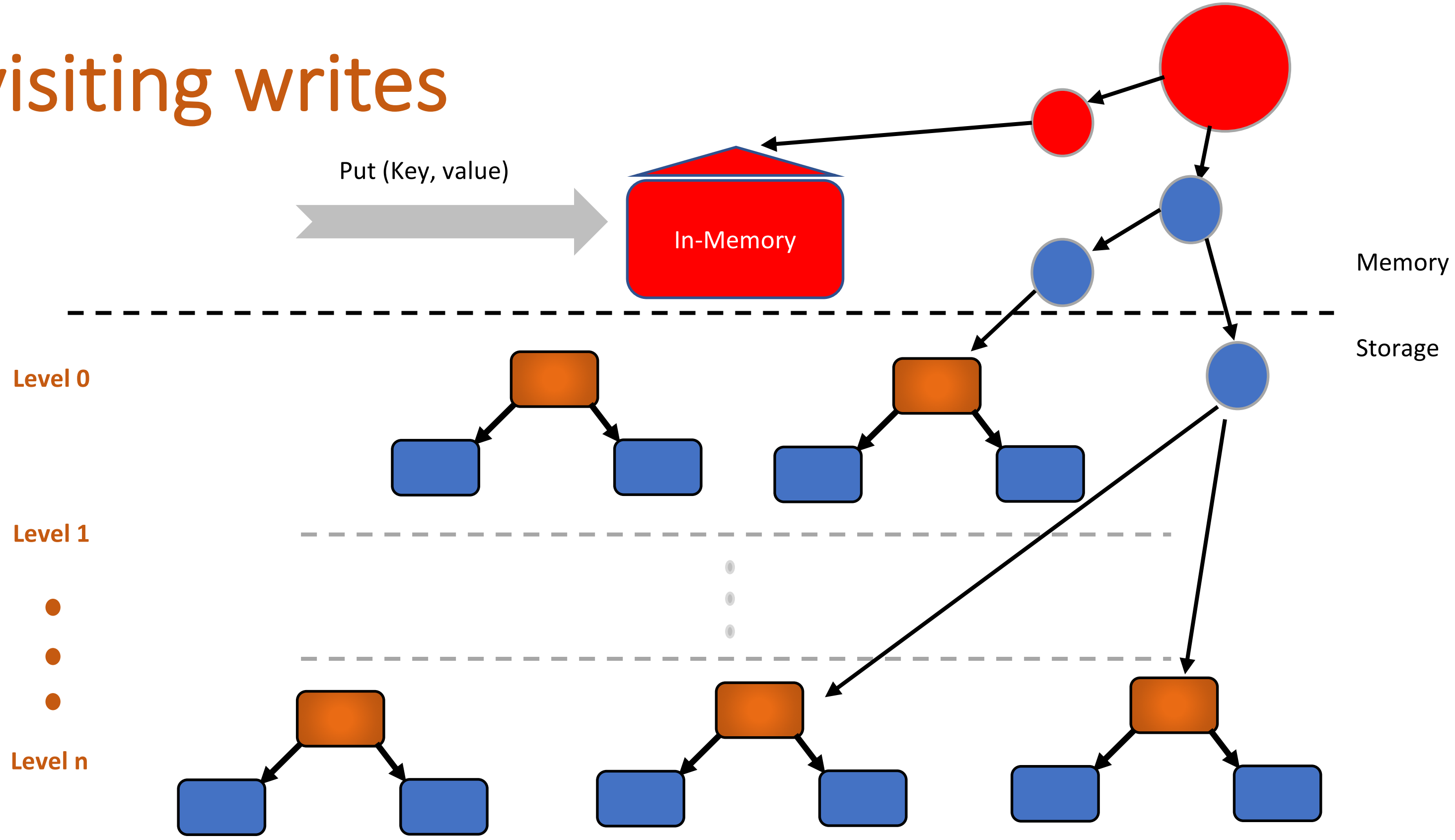
LevelDB cache

Key, Level	**Value, Proof**
key, Level	value, Local Proof



LevelDB cache can be populated once a new binary Merkle tree is created

Revisiting writes



Writes affect values in a single local tree and the global root

Would writes invalidate the whole cache?

- Global proofs are not cached
- Writes don’t invalidate any existing entries
- Keys at the same level are over-written when the binary tree is created
- Cache will not be invalidated on every write

Merkelized LSM : Reviewing the design

- Writes
  - Buffered in memory
  - Then written to storage
  - No in place updates
  - A write affects one tree and the master root
- Reads
  - Served from the cache
  - Or by traversing levels from lowest and till the first occurrence of key is found
  - Returns value and proof : <local proof, global proof>

Merkelized LSM advantages

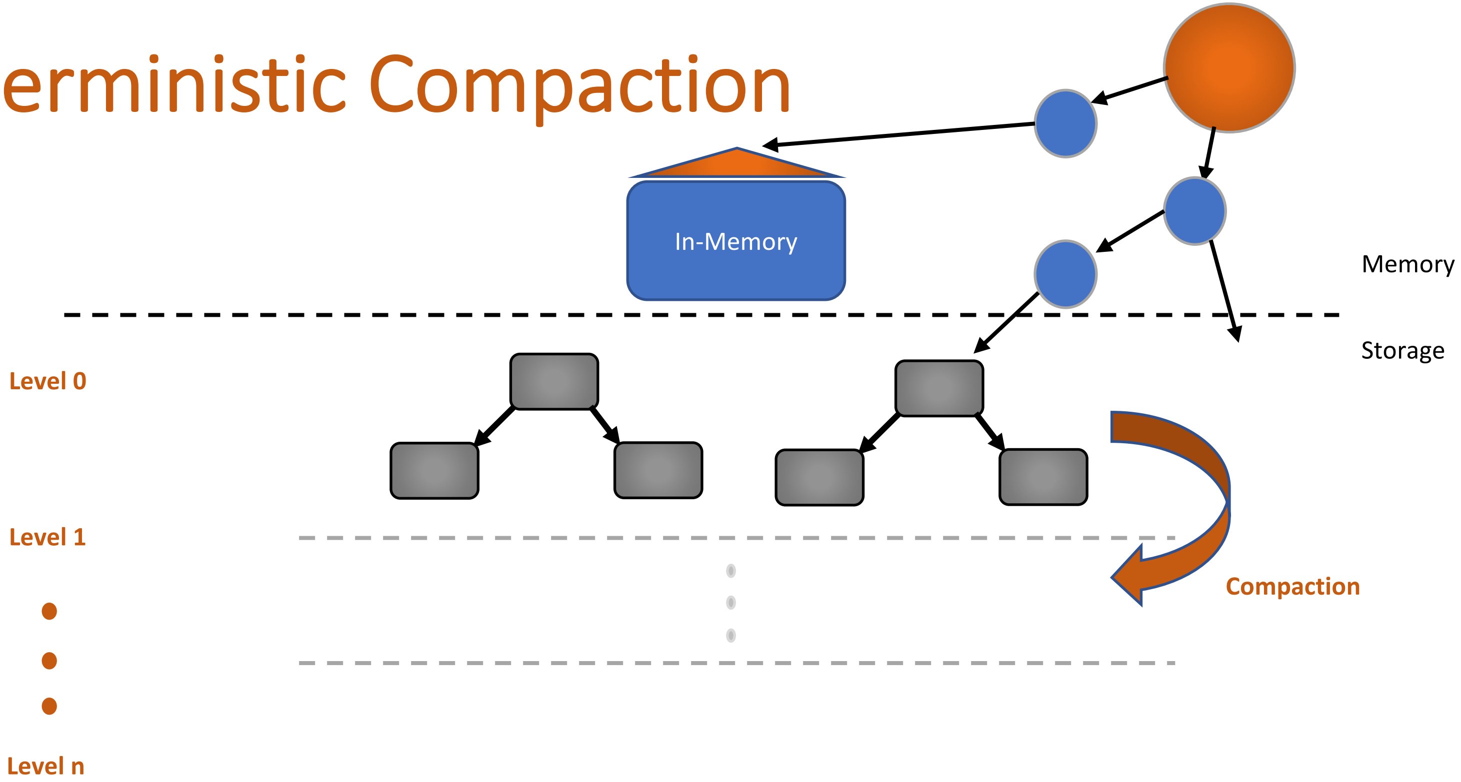
- Writes are handled in memory : O(1) complexity
- Reads :
  - Served from cache : O(levels in LevelDB cache)
  - Traversing the mLSM : O(height of mLSM * height of a binary Merkle tree)

Reads	Complexity	Served by
Cache Hit	O(Levels in Cache)	LevelDB cache
Cache Miss	O(Height of mLSM x Height of the binary tree)	Traversing mLSM

Merkelized LSM challenges

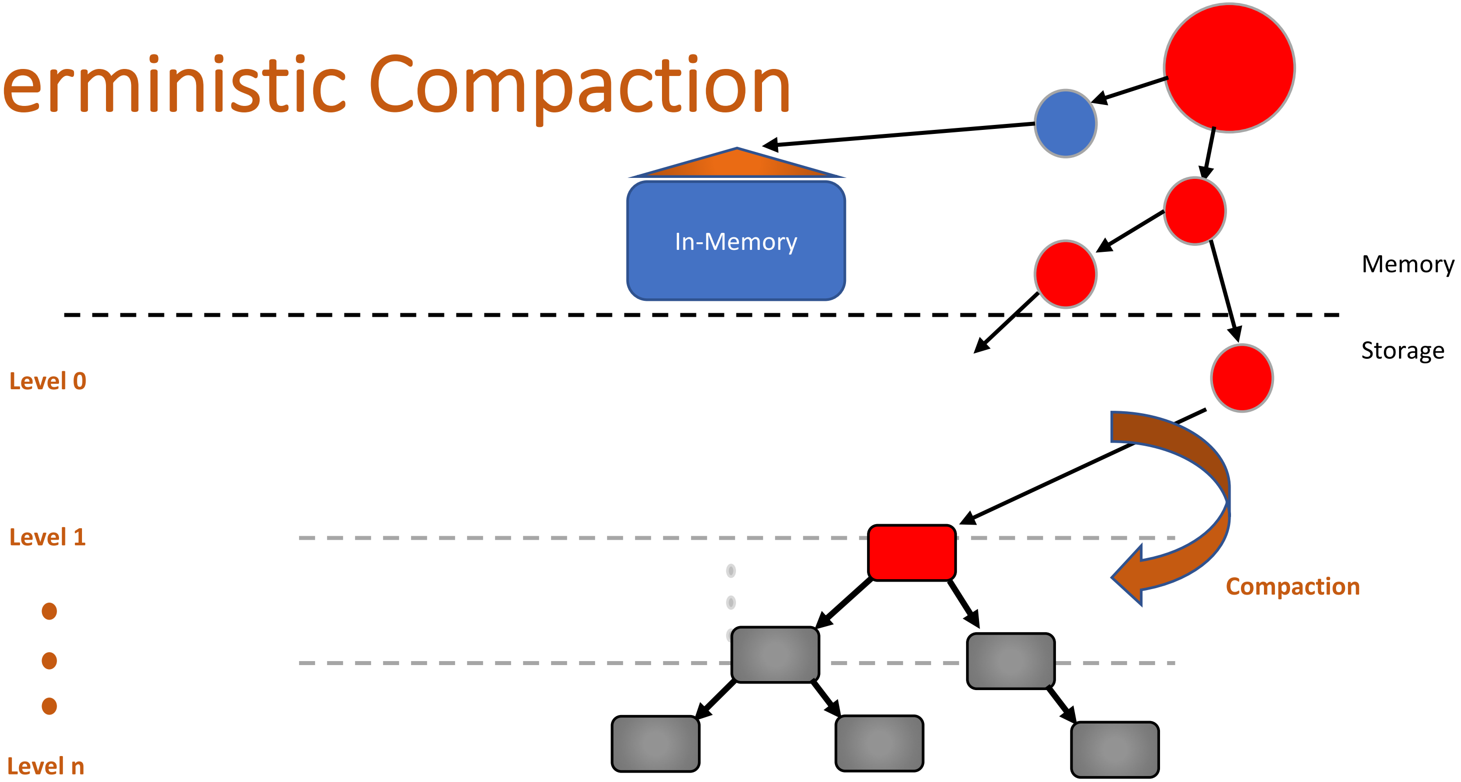
- Handling read amplification
  - Overhead of LSM structure is significant for applications with little data
  - LevelDB cache misses would result in read amplification
- Deterministic Compaction
  - Replicas : Multiple nodes storing data

Deterministic Compaction



Compaction changes the local roots

Deterministic Compaction



Compaction changes the local roots and the global root

mLSM : Authenticated Data Structure

- Minimizes IO Amplification
- Maintains multiple mutually independent binary Merkle trees
- Decouples lookup from authentication