# Avalution

## Authors:

Adyah Rastogi , Hao Lee , Jiahua Ren , Justin Lang , Wesley Truong

## Mentor:

Aaron Buchwald aaron.buchwald@avalabs.org

## Trello:

The link to join the Trello board is here. To see the tasks of a specific sprint, navigate to the respective “Spring X Done Tasks” column.

## Team/Project Title: Avalution

## Introduction:

### Problem Statement:

Merkle trees currently use databases such as LevelDB and RocksDB to store authenticated key value pairs. However, this adds another level of complexity to the system as the storage options are stacked on top of each other. This issue is significant as 80% of the time spent in blockchains are retrieving data. Therefore, any optimization on this part can greatly impact a cryptocurrency’s usage. Additionally, Golang is the preferred language in terms of blockchain companies, and even though this problem is solved via Rust, there are no implementations of this solution in Golang.

### Background

In the world of cryptocurrency, blockchain is the fundamental technology driving the existence and innovation of current cryptocurrencies such as Bitcoin and Ethereum. Blockchain is a decentralized, distributed digital ledger technology that stores data transparently and securely.

Conventionally, this data is stored in data structures known as Merkle Tries, where data is hashed and kept within the leaf nodes of a tree, and then combined with other sibling hashes recursively until reaching the Merkle root. Thus, by changing the data within one node, its hash will change and the change will propagate throughout the rest of the tree, making its way to the root and displaying a different hash. This property allows for fast verification of data by comparing hashes and identifying where changes occur, ensuring data integrity and security.

These Merkle trees are often stored on top of a generic database such as LevelDB or RocksDB, which have their own generic structure for how data is organized. Thus, the issue here is that these databases are different tree-like data structures themselves, and so further computation is required to transform the Merkle tree to fit the database.

In blockchains, low latency and fast verification are at the forefront of requirements to maintain the integrity of the network and prevent issues such as double-spending. If verification is slow, it would open the door to potential security vulnerabilities, such as malicious actors exploiting the delay to manipulate transactions. The amount of computation done is another factor as well, as oftentimes blockchain resides on light clients or mobile devices without the privilege of intense computation power.

To combat this, we recognize that Merkle trees have an implicit structure themselves, and so our goal is to circumvent the database portion entirely and store the Merkle tree on disk, leveraging the existing Merkle tree structure. On top of this, there are many potential speed-ups and optimizations that can be done when storing on-disk, and so we hope to further leverage these opportunities to create a low-latency blockchain platform.

## Project Specifics

Our project seeks to enhance the underlying architecture of the Avalanche blockchain by optimizing how key-value pairs of Merkle trees are stored directly on disk, moving beyond conventional database solutions like LMDB, LevelDB, and RocksDB. By developing and refining critical components such as the disk manager, free list, and revision manager, we aim to streamline the storage of recent Merkleized blockchain states, significantly reducing overhead and improving efficiency in state management.

## Science and Core Technical Advancements

Contrasting to databases used by Merkle Trees, if an on-disk implementation were to be created, there are multiple optimization options available for increasing performance and decreasing latency. Io_uring is a potential option to create multiple calls at once through their asynchronous read and write operations. Freelist can utilize a power of 2 algorithm for more efficient memory allocation and easier indexing. Our project plans to compete with current Merkle Trees/Databases to improve system performance and be faster at key-value requests.

## Team Goals/Objectives

As a team, we want to be able to:

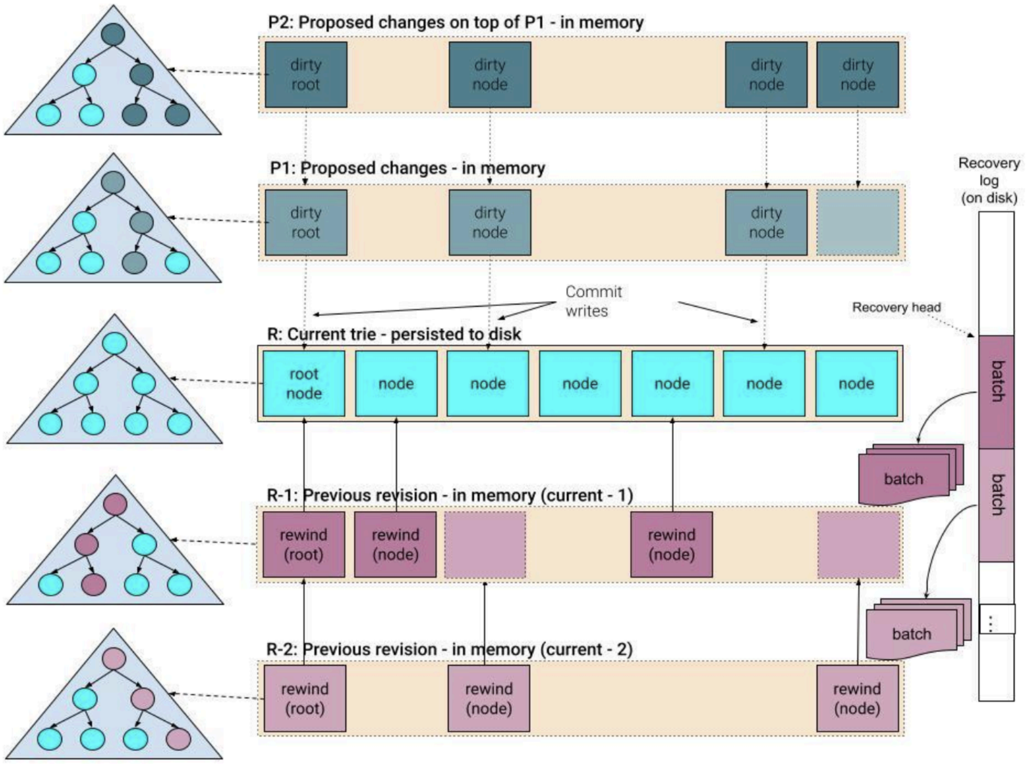
- Understand Blockchain
- Learn Golang, Merkle Patricia Trees

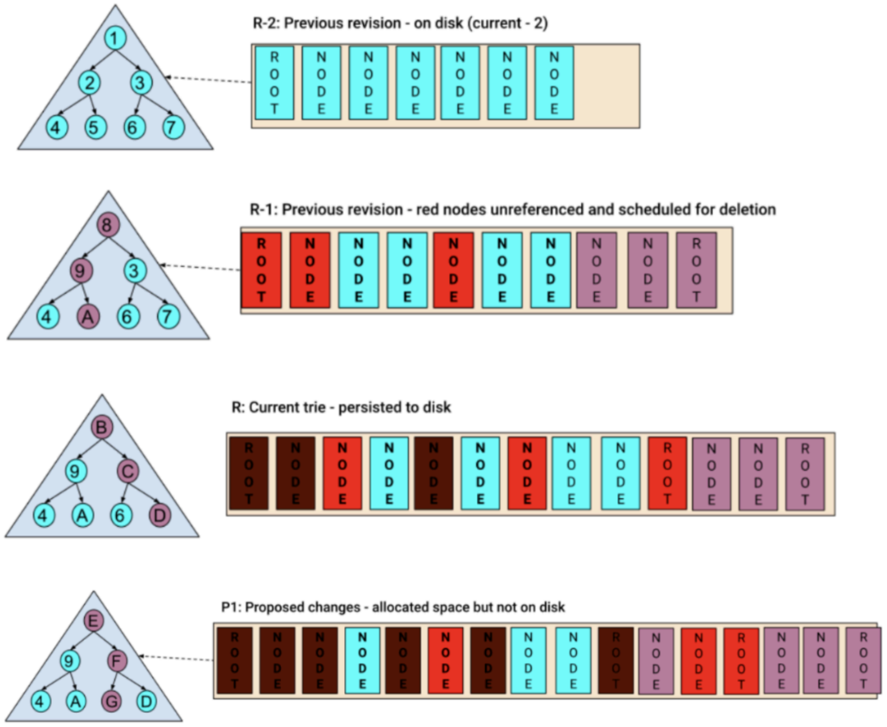
- Learn the Firewood project design and AvalancheGo MerkleDB design (golang merkle trie implementation writing to a generic key-value store that can be repurposed for this project)
- Write high-level spec for the individual components (serialization, free list, revision manager, and on-disk store)
- Implement on-disk serialization format and disk manager
- Implement revision manager to capture added/deleted nodes and write to disk manager
- Implement free list to re-use disk space from deleted nodes
- Update disk manager to utilize the free list
- Implement on-disk version of the free list
- Perform benchmarking at various database sizes (1GB to 1TB) to evaluate performance
- Optimize performance including memory allocations and write strategy (new-space vs. free list)
- Optional: plug directly into Avalanche C-Chain or Subnet-EVM to compare performance of executing the entire chain on an LSM Tree vs. Firewood
- Implement the state sync functionality, allowing the system to serve queries efficiently from previous revisions.
- Conduct benchmarking to identify and address bottlenecks for iteration

## Assumptions

As we work on the project, we have a core set of assumptions made about the project itself. We assume that Firewood is an implementation of our current issue, but since Golang is the preferred coding language for blockchain applications, we are modifying AvalancheGo (a similar version of Firewood in Go) to be on disk.

# System Architecture Overview:





## Requirements

As a developer, I can initialize a diskManager and either choose to pass in new metadata, read the existing diskManager metadata, or initialize an empty metadata, so that I can update and verify the persistance of a disk manager after closing.

- Trello: https://trello.com/c/gUoNDmsU
- Acceptance test: create diskManager without metadata, read metadata from existing diskManager, initialize diskManager with metadata
- Given: diskManager struct exists
- And: metadata can be passed in and read

As a developer, I can modify and see a new implementation of serialization which includes disk addresses of nodes, so that I can make changes and verify correctness of the new serialization format.

- Trello: https://trello.com/c/uQjR55VQ
- Acceptance test: serialization with disk address present when writing to disk
- Given: working existing on-disk serialization
- And: existence of disk address and serialization format of disk address
- When: working within the avalanchego/x/merkledb repository

As a developer, I can view a Go translation of Firewood so that I can get a better understanding of the general structure of what AvalancheGo should look like

- Trello: https://trello.com/c/jSmmltbv
- Acceptance test: there exists a non-gibberish translation of Firewood’s on-disk store/serialization/free list/revision manager code in Go
- Given: we have an open-source implementation of Firewood and AvalancheGo to use as the basis of our translation

As a developer I can write serializations onto disk, and deserialize them separately to check for persistence.

- Trello: https://trello.com/c/p9W2VvOX
- Acceptance test: we can write a serialization on disk, then deserialize by reading from the file on disk
- Given: We have a serialization format to serialize and unserialize our data
- Tasks:
    - Figure out how to write and read from disk efficiently
    - Figure out how to organize them using B+ tree structure

As a developer, I can view and run unit tests for each of the functions so that I can modify and change them while ensuring they still pass the tests.

- Trello: https://trello.com/c/2Ez4or1W

- Acceptance test: have working unit tests
- Given: a function of a feature
- When: running the tests, we can see whether the tests passed or failed
- And: for certain tests, we can see a log of the operations and changes to the file system

As a developer, I can find the root key of a Merkle Trie given a file that contains the raw disk.

- Trello: https://trello.com/c/TiMBAcB2
- Acceptance Test: Our function returns the root node successfully given a raw disk file
- Given: A raw disk file initialized with a merkle trie
- And: The correct root key address is stored in the metadata
- When: We read the root key address using disk manager, and compare if it is the correct rootkey

As a developer, I can accordingly pad disk address with 0 values to the next power of 2 size so that it will not overwrite another disk address.

- Trello: https://trello.com/c/U1jf8YkI
- Acceptance Test: WriteChanges should give us a correct output with no data being overwritten
- Given: A raw disk file and changes
- When: changes are passed into writechanges and written
- Then: Sizes of the disk addresses are read and added with 0 bytes
- And: They will not overwrite another disk address

As an investor, I can view a high level overview of the project and understand the project's design through UML diagrams and sequence diagrams.

- Trello: https://trello.com/c/2Qpp5LZU
- Acceptance Test: There are easy to understand diagrams representing the project's structure on the project GitHub readme.
- Given: Our project's design and code interface
- When: The project's GitHub repository is visited, there are straightforward diagrams to understand the project structure.

As a developer I can traverse our tree structure stored on disk using a key and return the node that matches it

- Trello: https://trello.com/c/sRQqSfzi
- Acceptance Test: We are able to retrieve a specified node from the merkle trie file and ensure that it is the same node as specified.
- Given: A key to a node
- And: Working functionality for getNode
- When: A specific key is entered, the matching node from the database will be returned.

As a developer, I have access to the DiskManager type so that I can keep code that interacts with the disk abstracted away and keep it cleaner.

- Trello: https://trello.com/c/ElJo6efK
- Acceptance Test: the disk manager writes and reads data correctly
- Given: constructed disk manager
- When: passing in the bytes to the write function, disk manager correctly writes to file
- When: passing in disk address to read, read the correct bytes back from file.

As a developer, I can put unused addresses / data buckets back into the freelist through the disk manager so that disk usage is efficient.

- Trello: https://trello.com/c/ZJFckrN8
- Acceptance Test: disk manager correctly utilizes free list and puts back address
- Given: constructed disk manager
- When: passing in the no longer used disk address to the put function, disk address is now free for use again for data of similar size.

As a developer, I can pass in changes to writeChanges without the freelist so that the function does not have any errors when trying to integrate disk into raw_disk.

- Trello: https://trello.com/c/r5f1Ne2c
- Acceptance test: abstraction of freelist for disk integration
- Given: a new version of writeChanges with an abstracted freelist
- When: a developer tries to utilize raw_disk instead of disk
- Then: There are no errors/function call changes

As a developer, I can add changes from the tree from the bottom up so that the children will have disk addresses and the parents can add disk addresses to its children.

- Trello: https://trello.com/c/foxuniKE
- Acceptance Test: Tree Bottom up changes
- Given: A new version of writeChanges with a different way of adding nodes to the tree.
- When: A new tree should be created
- Then: writeChanges should sort from the bottom layer nodes to the top layer
- And: Keep addresses/keys of children so that the parents know which diskaddresses should be added to which children.

As a developer, I can read the header of the diskManager which stores the information of the rootNode, so that I can eventually perform operations on the rest of the diskManager.

- Trello: https://trello.com/c/v7pV2C5f
- Acceptance test: correctly reads the header of the diskManager
- Given: a working implementation of initialized diskManager

- And: an existing header to compare reading to
- Then: check

As a developer, I have an overview of what the most important functionalities for implementing on disk Merkle trie capabilities are, and where to insert these into AvalancheGo.

- Trello: https://trello.com/c/4W5pBYHD
- Acceptance test: we are able to add functions into AvalancheGo and ensure that these functions have taken effect, given the functions that have been added, we can track the changes in the database
- Tasks:
    - Figure out how to run AvalancheGo
    - Find out where to add functions in specific files (step by step, add one function at a time)
    - Be able to test that these changes have made a difference internally

As a developer, I can utilize a freelist to manage the data on disk so that the data is stored efficiently.

- Trello: https://trello.com/c/d5STVgY9
- Acceptance test: the database with freelist implementation has the same functionality as one without freelist
- Given: A naive storage implementation (without freelist)
- And: basic functionality tests for both implementations
- When: tests are run, tests succeed
- And: tests on the efficiency of disk usage
- When: tests are run, the freelist implementation should be better than the naive implementation

As a company stakeholder, I can look and test benchmarks between this project implementation and Firewood to see that there is an improvement in speed/performance

- Trello: https://trello.com/c/ccdq1OFu
- Acceptance test: Benchmark produces favorable results for this implementation over Avalanche blockchain Merkle Tree index store (Firewood)
- Given: the Firewood implementation works
- And: our project implementation works
- When: the benchmark is ran against Firewood
- Then: our project performs better (whether that be latency-wise, overhead, etc.)

As an unfamiliar reader, I can get a quick overview of what the project is working on through the Github repository so that I can better understand this project that I am interested in.

- Trello: https://trello.com/c/I6YbBw9e

- Acceptance test: Outside observers understand the high level overview of what our project is trying to accomplish and why.
- Tasks:
    - Update the main README pages with information about Merkle Tries
    - Update the README pages with information about the motivation behind the project
    - Update the README pages with information about how we are going to achieve our goals

As a developer, I can communicate with the on-disk Go database so I can benchmark its performance.

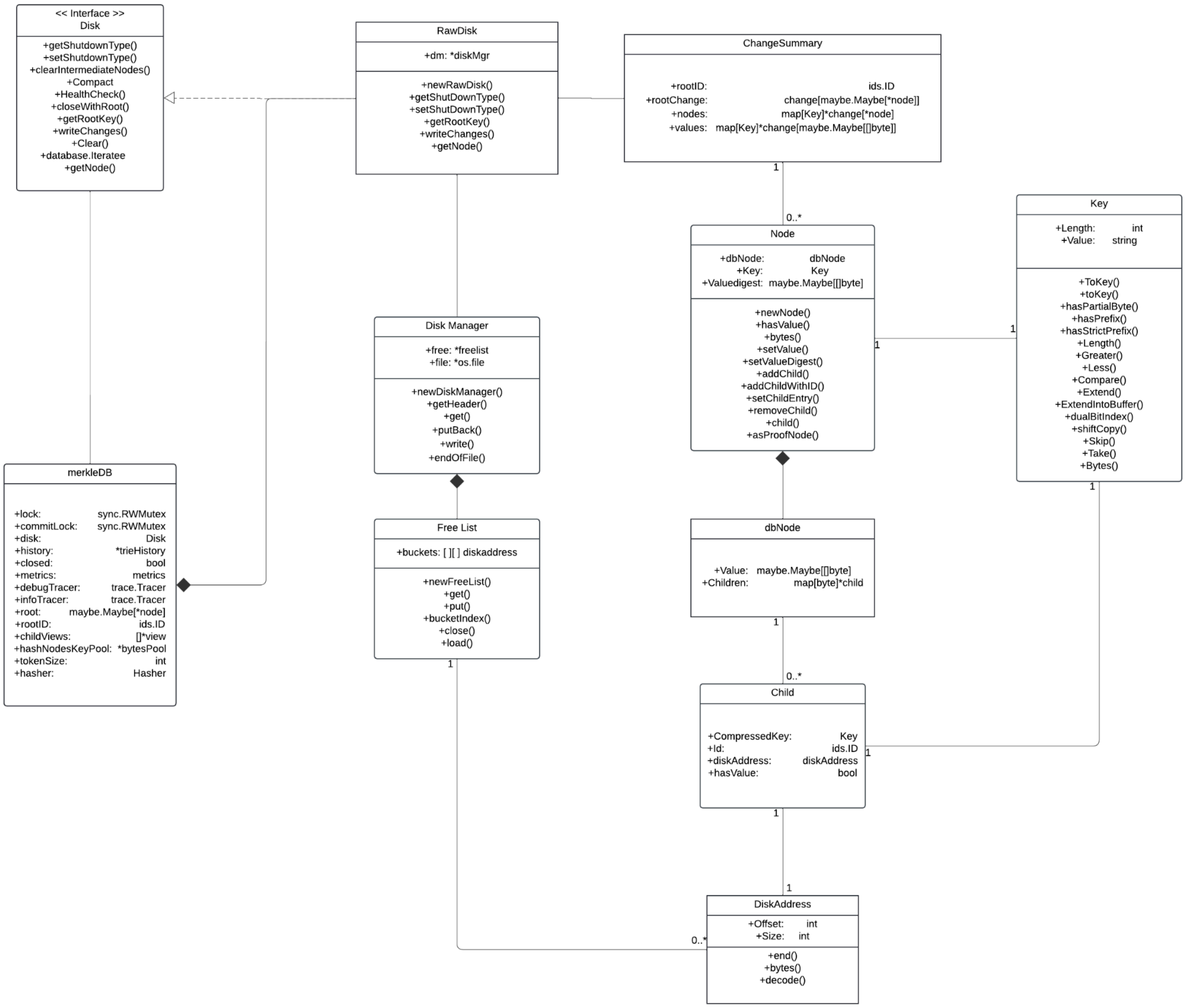
- Trello: https://trello.com/c/eqoxbZKr
- Acceptance test:
- Given: a benchmark database
- And: test input to database
- When: insert to our database, record relevant stats like its speed, memory usage, etc.
- Then: insert to the benchmark database and record stats
- Then: compare the two.

As a developer, I can find how the database interacts with debugging so that I can understand what components are used.

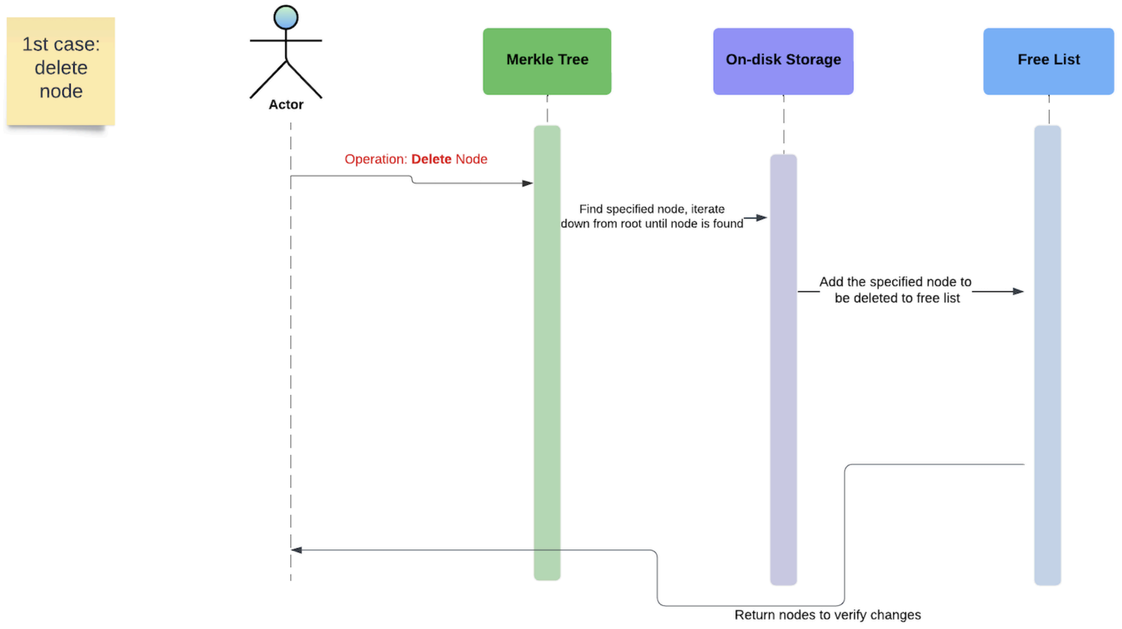
- Trello: https://trello.com/c/lNf1baBU
- Acceptance Test:
- Given: AvalancheGo interacts with LevelDB
- And: there are debugging tools and features that test these interactions
- When: the developer uses the debugging tools
- Then: they can find what components/functions are being used
- And: Communicate with rest of team

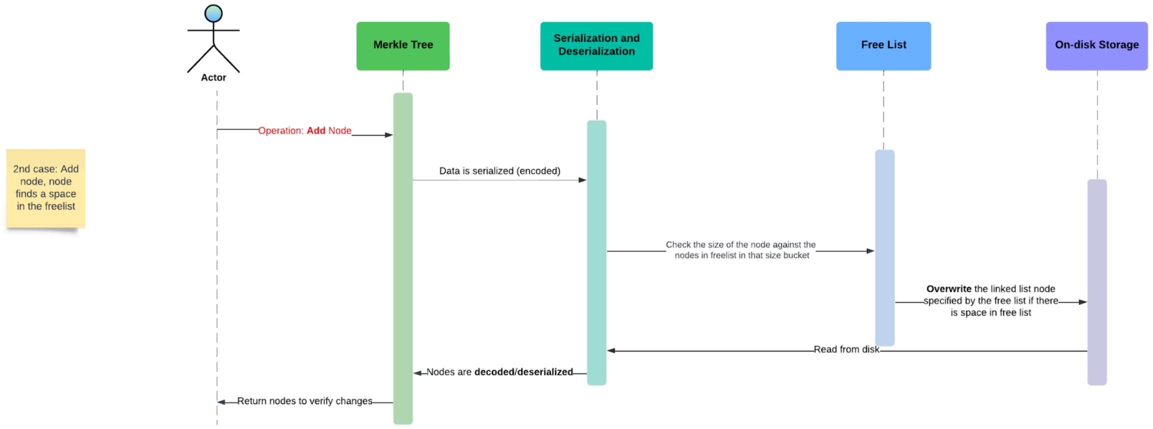
# System Models

## UML Diagram:

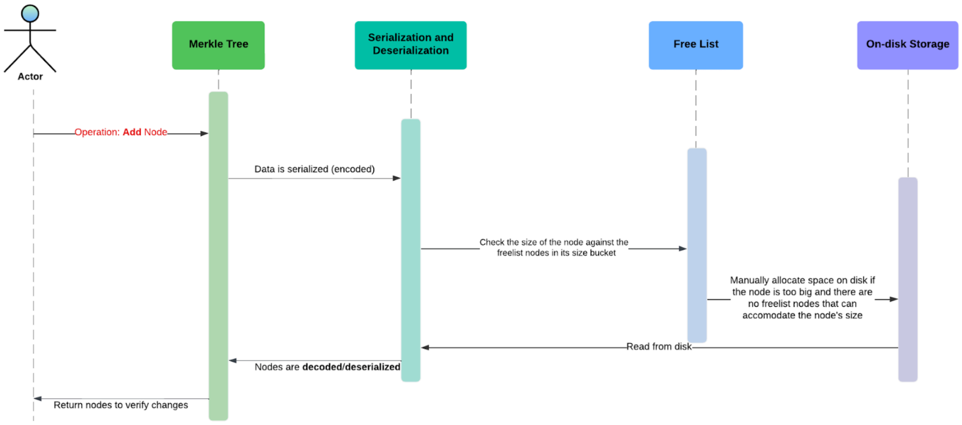


## Sequence Diagram:

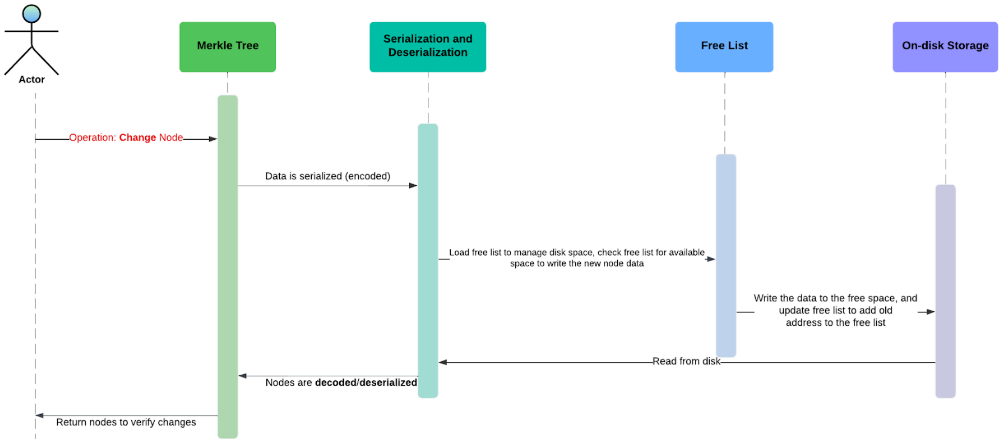

1st case: delete node
Actor
Merkle Tree
On-disk Storage
Free List
Operation: Delete Node
Find specified node, iterate down from root until node is found
Add the specified node to be deleted to free list
Return nodes to verify changes


Actor
Merkle Tree
Serialization and Deserialization
Free List
On-disk Storage
2nd case: Add node, node finds a space in the freelist
Operation: Add Node
Data is serialized (encoded)
Check the size of the node against the nodes in freelist in that size bucket
Overwrite the linked list node specified by the free list if there is space in free list
Read from disk
Nodes are decoded/deserialized
Return nodes to verify changes

3rd case: Add node, node doesn't find space in the freelist, need to manually allocate


Actor
Merkle Tree
Serialization and Deserialization
Free List
On-disk Storage
Operation: **Add** Node
Data is serialized (encoded)
Check the size of the node against the freelist nodes in its size bucket
Manually allocate space on disk if the node is too big and there are no freelist nodes that can accomodate the node's size
Read from disk
Nodes are **decoded/deserialized**
Return nodes to verify changes

4th case: Changing a node


Actor
Merkle Tree
Serialization and Deserialization
Free List
On-disk Storage
Operation: **Change** Node
Data is serialized (encoded)
Load free list to manage disk space, check free list for available space to write the new node data
Write the data to the free space, and update free list to add old address to the free list
Read from disk
Nodes are **decoded/deserialized**
Return nodes to verify changes

# Appendices

## Technologies Employed:

- Rust
- Golang
- Firewood
- AvalancheGo Merkle DB
- Merkle Trees