Adamnite Data Storage

The Adamnite Protocol, like other layer-1 blockchains, should leverage a cryptographic data storage protocol for the purpose of both creating proper block headers and efficient client verification of the actual chain. While much of the [initial discussion](https://github.com/Adamnite/POC_1/blob/main/protocol_research/Protocol%20Implementation%20(in%20progress)%20(1).pdf) around cryptographic data storage has centered on Merkle Trees, this proposal is meant to serve as a more specific overview of what Adamnite’s database (DB) structure could look like, and how both the commitment of new data and fetching of key-pair values can be handled. We propose a DB structure similar to the one formulated by Buterin and Ballet in [EIP-3102](https://eips.ethereum.org/EIPS/eip-3102), which envisioned a potential transition from hexary to binary for the Merkle Patricia Tries (MPT) commonly used by the Ethereum Protocol. This structure, with some changes made specifically for Adamnite’s requirements, is outlined below. Please note that the final[[1]](#footnote-0) DB structure for Adamnite will likely differ from this initial POC implementation. The final version of Adamnite will likely leverage Verkle Trees, which utilize Vector Commitments instead of hash functions to generate a commitment of new data.

However, for Adamnite’s POC implementation, a binary implementation of a Merkle Tree should be satisfactory, especially given that a majority of Adamnite’s innovation rest in its programming ecosystem and consensus protocol. Much like the implementations of other account-based blockchains, the DB leverages a tree structure called a “trie” for easy access. The structure is described below:

The [trie](https://en.wikipedia.org/wiki/Trie) is made up of nodes. A node N has the following structure:

N = (N\_R, N\_L, N\_P, N\_V), where the N\_R is the hash of the right child of the node, N\_L is the hash of the left child of the node, N\_P is an optional byte-prefix of all key-values in the subtries of the node, and N\_V is the actual value stored at this node (only for leaf nodes, or the nodes at the very bottom of the trie).

Adamnite Account values can be accessed using the following methodology:

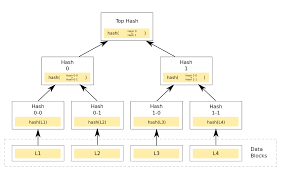
For an account X, where the values X\_A, X\_B, X\_N, and X\_S represent the account’s address, balance, nonce (the total number of transactions it has sent), and the root of the storage trie (for the POC, this can just be a regular byte value instead of the root of another trie) respectively, the values X\_A, X\_B, X\_N and X\_S should be stored as key values in the trie to be accessed later.

For example, the account balance can be found at key “hash(A\_A)[0-253] ++ 0b00, and is of type uint256;

The account nonce A\_N can be found at key hash(A\_A)[0..253] ++ 0b01 and is of type uint64;

The storage A\_S is an arbitrary-length byte sequence that can be found at key hash(A\_A)[0..253] ++ 0b10;

The diagram below displays a high–level version of what the trie could look like. It does not include, for example, the hashes that will be in each of the keys. Each of the nodes below can be empty. A proof that a value is in the tree will include all sister nodes leading up to th root. Each node in the diagram below not named value is considered to be hashed with a hashing function. These hashes are concatenated with their sisters repetitively until the root is formed.



Graph from Wikipedia

1. Final in this case means mainnet version. [↑](#footnote-ref-0)