Besu Bonsaï like storage: Specification

Table of Contents

1. Introduction

1.1. Scope of the document

This specification is the entrypoint for developers to contribute to the project and provides all the guidelines to organize the work efficiently. This document should be enough to work as a team with all the necessary explanations and overview of the whole project. This document will evolve over time.

1.2. Target of the document

This document should be a starting point and a reference for all the developers that contribute and a high level source of documentation about the project.

A technical expertise in Rust is required to understand the interfaces. The rest of the document is readable with limited knowledge of the technical principles involved.

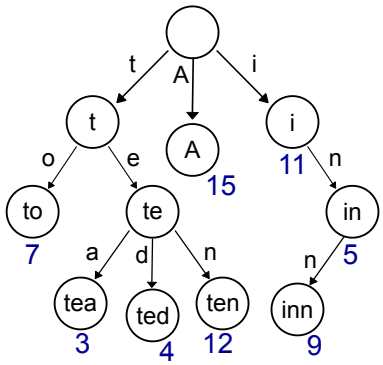
1.3. Concepts and Vocabulary

1.3.1. Besu Bonsai storage

Besu Bonsai storage is an advanced storage management system developed by HyperLedger for their Ethereum client, Besu. For a detailed specification, follow this link.

1.3.2. Trie

Unlike a conventional tree, a trie is designed so that each node (except the root) represents a byte. Each path descending the trie can symbolize a key in the form of a byte array. Each key is associated to an arbitrary value, making the trie a key-value data structure. To better understand, consider the following illustration:

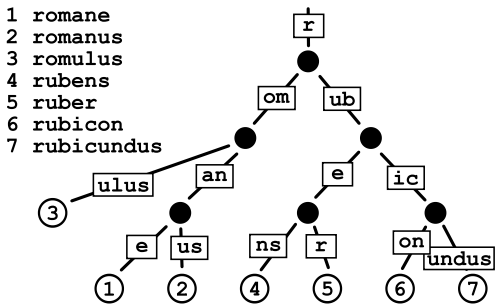


In this trie, connected nodes form keys by appending the key byte they represent. For instance, the trie contains keys like **A** - with a value of 15, **to** - 7, **tea** - 3, and so on. However, **t** or **te** aren't keys but merely prefixes.

For a deeper dive into tries, follow this link. The advantages of the trie data structure are detailed here.

1.3.3. Radix trie

Also known as a radix tree, compact prefix tree or compressed trie, a radix trie is data structure that represents a space-optimized trie in which each node that is the only child is merged with its parent. Here's a visual representation:



In this trie, the nodes **om**, **ub**, **ulus** ... are merged into a single node, as they are the only children of their parent. Instead of needing 7 nodes to represent the key **romulus**, each letter having its node, this space-optimized representation only needs 3, **r**, **om** and **ulus**.

For a deeper dive into radix tries, follow this link.

1.3.4. PATRICIA tree

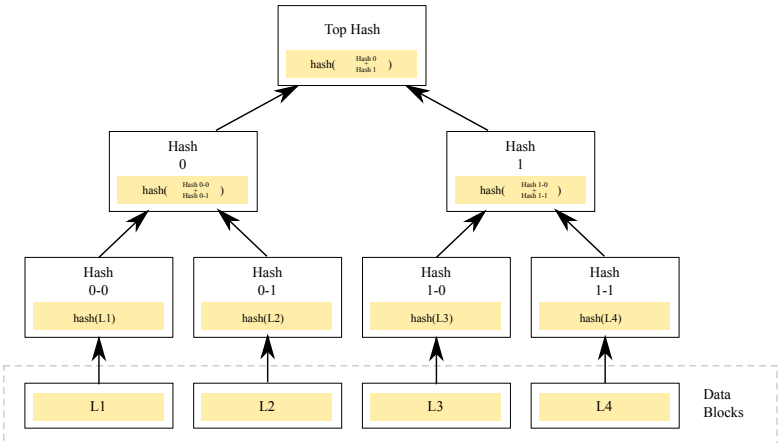
A PATRICIA tree, which stands for Practical Algorithm to Retrieve Information Coded in Alphanumeric, is a special variant of the radix trie where the radix is equal to 2. This implies that each node carries a 1-bit portion of the key and has at most two children (child 0 and child 1).

For a deeper explanation of the difference between PATRICIA tree and radix trie, you can explore here.

The PATRICIA tree in Ethereum also provides optimizations on the trie format that are detailed here.

1.3.5. Merkle Tree

A Merkle Tree is a specialized tree where every "leaf" or node is labeled with the cryptographic hash of a data block. Conversely, every non-leaf node, often referred to as a branch, inner node, or inode, carries the cryptographic hash of its child nodes' labels. Here's a visual representation:



In this depiction, hashes 0-0 and 0-1 represent the hash values of data blocks L1 and L2, respectively. Meanwhile, hash 0 is derived from the combined hashes 0-0 and 0-1.

For comprehensive information on the Merkle Tree, visit here. The benefits of the Merkle Tree data structure are outlined here.

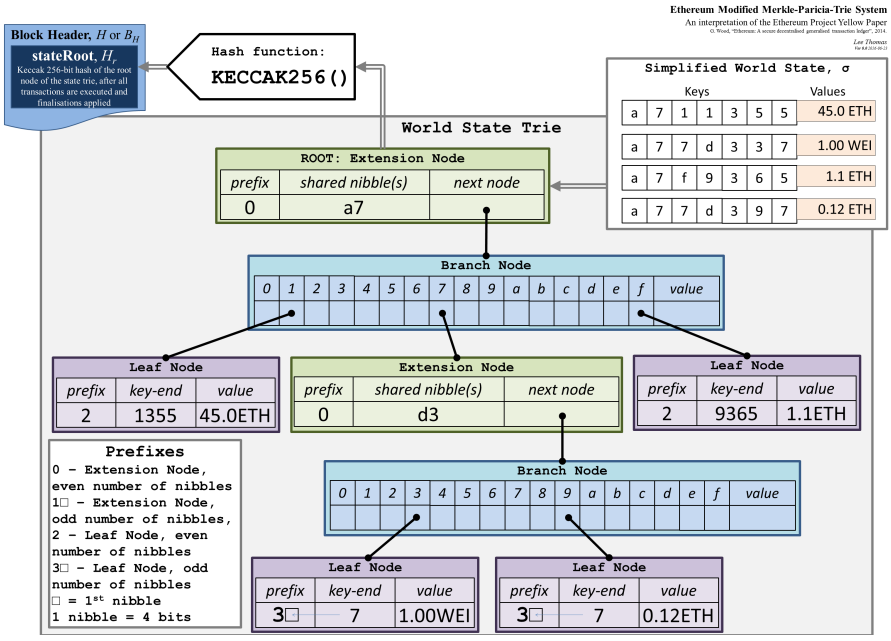
One important feature of Merkle Trees is that they allow exhibiting compact proofs of existence of an element in the tree. The proof size is logarithmic in the number of elements in the tree.

1.3.6. Merkle-Patricia Trie

A Merkle-Patricia Trie (MPT) is a combination of a Merkle Tree and a Patricia Tree. This data structure is famous because it is being used by Ethereum to store the state of an Ethereum blockchain. The Ethereum version of the MPT is composed of 3 types of nodes:

- **Branch:** A node with up to 16 child links, each corresponding to a hex character.
- **Extension:** A node storing a key segment with a common prefix and a link to the next node.
- **Leaf:** An end-node holding the key's final segment and its value.

Here's a visual representation:



In this depiction, the key a77d397, having the value of 0.12 ETH, is stored using 5 nodes (a7 - extension node, 7 - branch node, d3 extension node, 9 - branch node, 7 - leaf node).

1.4. External Resources

Substrate

Substrate is a Rust framework developed by ParityTech, designed to facilitate the creation of blockchain nodes.

1.5. Versions

Version	Date	Author	Description
0.1	2023-11-16	Aurélien FOUCAULT	Initial version
0.2	2023-11-20	Aurélien FOUCAULT	Rework interfaces

2. Function Description

2.1. Actors

2.1.1. Caller

The library is designed for callers who need a key-value data structure with efficient data management operations for retrieval, storage and deletion, while maintaining a global fingerprint (hash) of the whole structure that allows for compact proofs of element existence. Callers can either use the library directly or through another intermediary library. The primary advantage is that callers can utilize the library without delving into its underlying implementation and can choose a database implementation that suits their needs.

2.1.2. High-Level Interface

This interface serves as the library's main entry point and the primary interface for callers. It simplifies interactions with the library, ensuring that the caller only engages with this interface, abstracting away the complexities of the underlying processes.

2.1.3. Accumulator

The accumulator plays a pivotal role in the management of the state of the data structure. It facilitates the addition of new states and retrieves states at specific point in time. The high-level interface leverages the accumulator for these tasks.

2.1.4. Trie

Trie is the chosen data structure for data storage within the library. Both the accumulator and the high-level interface utilize the Trie for data operations.

2.1.5. Trie Logs

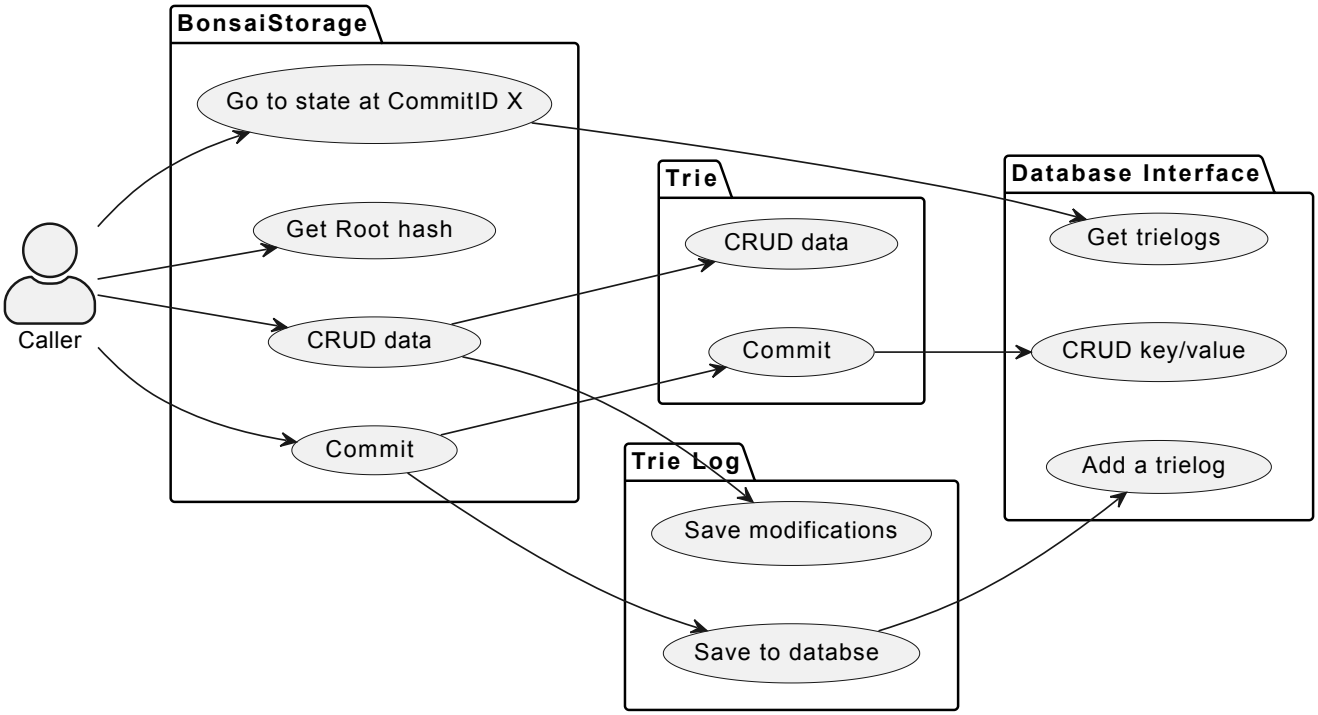
Trie logs capture batches of modifications, detailing every change made during the processing of a "commit" which is an atomic batch of modifications that can for example represent the changes caused by the execution of block in a blockchain. These log are required to the accumulator when it needs to roll back or roll forward to a particular state.

2.1.6. Database Interface

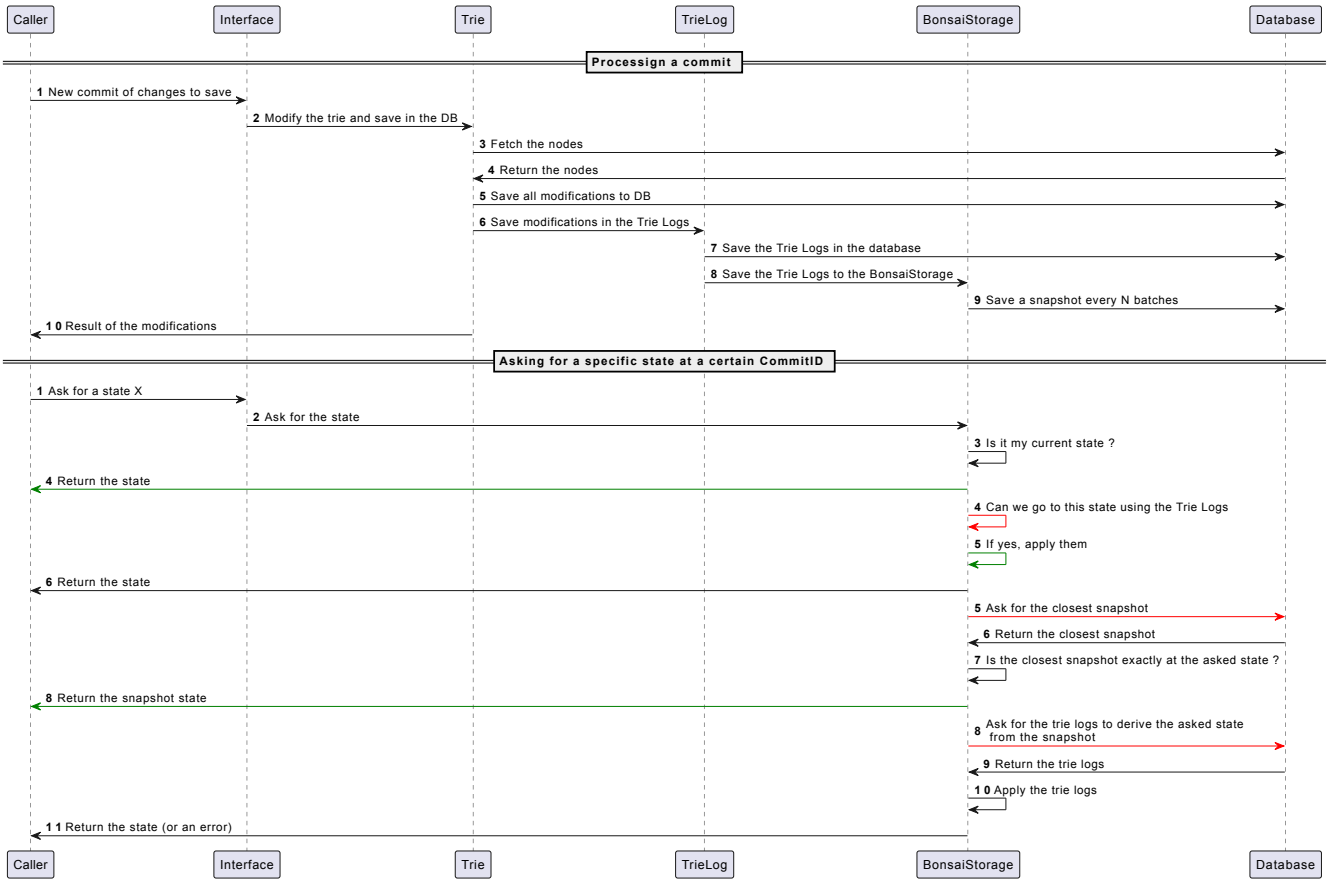
Serving as the underlying persistent storage mechanism, the database ensures data longevity by saving it to the disk. While various library components rely on the database for low-level data management, its interface is implemented by the caller. This design choice maximizes abstraction and portability, allowing the library to be adaptable across different database implementations.

2.2. Diagrams

2.2.1. Use case



2.3. Sequence



3. Technical Description

3.1. Trie

The Trie is the central component of the Besu storage system. To avoid reimplementing a PMT, we use the crate Trie from paritytech which provides a standard PMT. This choice was made for multiple reasons:

- It avoids re-implementing a PMT.
- It provides the flexibility to create Tries formatted for different blockchains.
- The code has a lot of generics and gives us the possibility to make modifications to the PMT structure easily.

We only use the sub-crate `trie-db` and override its keys, database, and layout implementations. However, this sub-crate only solves part of the problem: - In a Bonsai Trie, we store nodes directly by their location, while the Trie crate stores them by hash - The Trie crate does not allow the implementation of trie logs, which require some modifications to the crate code

Given those constraints, we forked the Trie crate while minimizing the changes to the code. Our modifications make the crate more generic and are being proposed to the maintainers as an upstream PR.

3.1.1. Attributes

No attributes need to be defined. All traits are detailed below.

3.1.2. Traits/Implementations

Trie interface

The `TrieDBMut` structure of the `trie` crate is used. Definition can be found here

3.1.3. Database trait

The Trie crate from paritytech already provides a database trait described here. We reuse the same trait for this part of the project.

3.1.4. Child encoding

In the sub-crate `trie-db`, the children of a branch node are either inline or referenced by hash. In a Bonsai Trie, as the nodes are stored by their locations, we still want to save the hash to have it cached when parent hashes need to be computed (up to the trie root hash).

3.1.5. Node encoding

The sub-crate `reference-trie`` gives the implementations of `trait NodeCodec` (that manages the serialization/deserialization of nodes to store them in DB) with the same behavior as the one used on Substrate. We re-use it as it corresponds to our needs and gives us more compatibility with existing Substrate code.

3.2. Accumulator

The accumulator is an optimization component that is used to register all the reads and changes made to provide optimizations for future actions. For example when reading a value, in a second thread, the path in the trie will be loaded in the accumulator in case the value is modified in the same batch of changes

3.3. Trie logs

Trie logs store a batch of modifications to the trie to be applied to a state.

3.3.1. Attribute

- In each trie log, we should save all keys/values that are modified within `HashMap<ID, Vec<(Option<Vec<u8>>, Option<Vec<u8>>)>`. The value is represented as `Vec<(Option<Vec<u8>>, Option<Vec<u8>>` to be able to hold the old value (if existed) and the new one (if we are not in a remove operation) for each key.
- An ID

3.3.2. Traits/Implementations

```
impl TrieLog<CommitID>
where:
  CommitID: Id {
    // Initialize a trie log with a batch of modifications
    fn new(id: CommitID, modifications: Vec<(Option<Vec<u8>>, Option<Vec<u8>>)>) -> Self

    // Get the associated CommitID
    fn get_commit_id(&self) -> CommitID;
}
```

3.4. Database

The database implementation is generic on the underlying database. We provide a first implementation of the database using RocksDB.

All methods take an optional transaction type that allows making transactional modifications to the database if the database type allows it. If the transaction object is provided, we don't update the DB directly but accumulate the changes into the provided TX, and commit it afterwards.

3.4.1. Attributes

- A connector to the database

3.4.2. Traits/Implementations

```
pub enum DatabaseError {
    // All errors related to the db, omitted here
}

pub trait BonsaiDatabase {
    fn new(path_to_database: &str) -> Self

    // Insert an entry in the trie
    fn insert(&mut self, key: &[u8], value: &[u8]) -> Result<(), DatabaseError>;

    // Remove an entry from the trie
    fn remove(&mut self, key: &[u8]) -> Result<(), DatabaseError>;

    // Get a value in trie
    fn get(&self, key: &[u8]) -> Result<Vec<u8>, DatabaseError>;

    // Check if the key is in trie
    fn contains(&self, key: &[u8]) -> Result<bool, DatabaseError>;

    // PUT operation in TRIE_LOG column
    fn put_trie_log(&mut self, key: &[u8], value: &[u8]) -> Result<(), DatabaseError>;

    // GET operation in TRIE_LOG column
    fn get_trie_log(&self, key: &[u8]) -> Result<Vec<u8>, DatabaseError>;

    // Generate a snapshot
    fn generate_snapshot(&mut self) -> Result<u64, DatabaseError>

    // Get a snapshot
    fn get_snapshot(&self, snapshot_id: u64) -> Result<BonsaiDatabase, DatabaseError>
}
```

3.5. User-facing interface

This is the main interface that the Caller interacts with.

3.5.1. Definition

```
// Error type for the interface (elided)
pub enum BonsaiStorageError {}

// Configuration
// None = unlimited
pub struct BonsaiStorageConfig {
    pub max_trie_log_size: Option<usize>,
    pub max_snapshot_saved: Option<usize>,
    pub max_trie_logs_in_memory: Option<usize>,
    pub snapshot_interval: usize,
}

pub trait BonsaiStorage<'a, CommitID>
where:
  CommitID: Id {
    // Create a new bonsai storage instance
    fn new(db: &'a mut KeyValueDB<DB, CommitID>, root: &'a mut BonsaiTrieHash, config:
BonsaiStorageConfig) -> Self;

    // Insert a new key/value in the trie, overwriting the previous value if it exists
    // If the value already exists it will overwrite it and return the previous value if any
    fn insert(&mut self, key: &[u8], value: &[u8]) -> Result<Option<Vec<u8>,
BonsaiStorageError>;

    // Remove a key/value in the trie
    // If the value doesn't exist it will do nothing
    fn remove(&mut self, key: &[u8]) -> Result<Option<Vec<u8>>, BonsaiStorageError>;

    // Commit all the changes to the trie
    // This is in charge of saving the in-memory `insert`/`remove` operations in the database
otherwise
    fn commit(&mut self, id: CommitID) -> Result<(), BonsaiStorageError>;

    // Get a value in the trie
    fn get(&self, key: &[u8]) -> Result<Option<Vec<u8>>, BonsaiStorageError>;

    // contains checks if the key exists
    fn contains(&self, key: &[u8]) -> Result<bool, BonsaiStorageError>;

    // Go to a specific commit ID using trie logs and snapshots.
    // If insert/remove is called between the last `commit()` and a call to this function,
    // the in-memory changes will be discarded
    fn go_to(&mut self, commit_id: CommitID) -> Result<(), BonsaiStorageError>;

    // Get the root hash of the trie, at the state of the latest commit
    fn root_hash(&self) -> Hash;
}
```

3.5.2. Usage example

For example you will be able to use the interface like this:

```
fn main() {
  let mut db = KeyValueDB::<RocksDB, BasicId>::new(RocksDB::new("./rocksdb")); // add empty
node
  let mut root = BonsaiTrieHash::default();
  let mut bonsai_storage = BonsaiStorage::new(&mut db, &mut root,
BonsaiStorageConfig::default());
  let mut id_builder = BasicIdBuilder::new();
  bonsai_storage.insert(&[1, 2, 3, 4, 5, 6], &[4, 5, 6]);
  bonsai_storage.insert(&[1, 2, 3, 4, 5, 7], &[4, 5, 8]);
  bonsai_storage.commit(id_builder.new_id());
  bonsai_storage.insert(&[1, 2, 2], &[7, 5, 6]);
  let go_to_id = id_builder.new_id();
  bonsai_storage.commit(go_to_id);
  bonsai_storage.remove(&[1, 2, 2]);
  bonsai_storage.commit(id_builder.new_id());
  println!("root hash: {:#?}", bonsai_storage.root());
  println!(
      "value for key [1, 2, 3, 4, 5, 6]: {:#?}",
      bonsai_storage.get(&[1, 2, 3, 4, 5, 6]).unwrap()
  );
  bonsai_storage.go_to(go_to_id).unwrap();
  println!("root hash: {:#?}", bonsai_storage.root());
  println!("value for key [1, 2, 2]: {:#?}", bonsai_storage.get(&[1, 2, 2]).unwrap());
}
```

3.6. Tests

We provide the following tests:

- A test with a simple set of key/value
- A test with a big set of key/value
- A test with different types of `TrieValue`
- A test with behavior closest to the one of Madara.