Accelerating Prolly Trees: Simplified Chunking for Rapid Updates

Abhimanyu Rawat[1,*], Tarun Kumar Vangani[2], Hanno Cornelius[3] and Vanesa Daza[1]

[1]*Department of Information and Communication Technologies, Universitat Pompeu Fabra, 08018 Barcelona, Spain*
[2]*BITS Pilani, India*
[3]*Status Research and Development, Singapore*

Abstract

Amidst the growing reliance on Conflict-free Replicated Data Types (CRDTs) for synchronizing data across distributed networks, our research introduces a novel implementation of Prolly trees. This innovative approach overcomes the limitations of existing Prolly tree models by simplifying design complexities and enabling faster implementation. Our design strategically mitigates existing performance bottlenecks and operational intricacies by introducing an advanced architecture that curtails complexities associated with chunk splitting and minimizes the necessity for updating tree nodes. Furthermore, our implementation incorporates an ingenious Anchor Nodes design and supports batch insertions, substantially elevating the structure's proficiency in managing updates, insertions, and deletions. Developed using Golang, to leverage its execution speed, and Python, for its readability and straightforward adaptability, our version presents significant improvements over conventional Prolly tree configurations. Through exhaustive evaluation and comparative analysis against existing benchmarks, our model underscores a pronounced enhancement in performance metrics. Our method significantly improves both creation and insertion operations, achieving a 30% to 50% enhancement for Dolthub and a multiple-fold increase in performance for Canvas.

Keywords

Prolly Tree, CRDT, Merkle Tree, Blockchain, Distributed, Data Structure

1. Introduction

With the continuous growth of decentralized technologies, the necessity for scalable and efficient data structures has intensified. Conflict-free Replicated Data Types (CRDTs) [1] have gained prominence within the decentralized computing arena. CRDTs are specifically engineered to allow replication across multiple nodes within a network, facilitating convergence towards a consistent state without necessitating a centralized authority [2].

One notable example of a CRDT is the Prolly (Probabilistically Balanced) tree, a distributed data structure optimized for replicating data across multiple nodes within a network, particularly beneficial in the context of State Machine Replication systems such as blockchains [3, 4]. Here, Prolly trees play a crucial role in assisting client software [5], operating atop communication layers like libp2p [6] or devp2p [7], to synchronize states. These states may encompass sets of transactions, for example, facilitating a seamless state synchronization process. The Prolly tree [8, 9], a variant of the Merkle tree [10], utilizes a tree-like structure to store the hashes of individual data blocks within a large dataset. This arrangement facilitates efficient verification of the dataset's contents. By calculating a set difference between two Prolly trees in logarithmic time, it is possible to synchronize the state between multiple parties. Designed for environments requiring weak consistency, the Prolly tree offers both scalability and efficiency, enabling the synchronization of large datasets under eventual consistency paradigms.

The key performance metrics for a Prolly tree are multifaceted, including the chunking algorithm which significantly influences the tree's structure, the capability for inserts to minimally impact the

DLT'24: 6th Distributed Ledger Technology Workshop, May 14–15, 2024, Turin, Italy
[*]Corresponding author.
abhimanyu.rawat@upf.edu (A. Rawat); f2013817@pilani.bits-pilani.ac.in (T. K. Vangani); hanno@vac.dev (H. Cornelius); vanesa.daza@upf.edu (V. Daza)
https://abresting.github.io/ (A. Rawat)

existing structure, and the efficiency of random reads amidst continuous insertions, among others. Existing designs [11, 12, 13] have underscored the necessity for optimizing the Prolly tree to enhance its performance and scalability. In production environments to synchronize the extensive datasets, the goal is to develop a Prolly tree that not only processes large volumes of data effortlessly but also scales efficiently in response to a high influx of messages. The performance of Prolly trees is significantly affected by the order of message or data insertion. However, within the framework of causal broadcast, data tend to follow a linear sequence, mitigating the complexities associated with random insertions and thereby enhancing the operational efficiency of Prolly trees. Nevertheless, designing a system that accommodates random insertions with minimal disruption to the existing tree structure remains paramount.

In this paper, we introduce an optimized iteration of the Prolly tree, specifically engineered for enhanced efficiency and scalability, particularly under conditions of sequential insertion. This refined version of the Prolly tree has been developed and tested using Python and Golang, with extensive performance evaluations conducted to ascertain its efficacy and scalability. The findings from our analysis indicate that the optimized Prolly tree exhibits superior capability in managing large datasets and demonstrates potential to significantly improve scalability across an extensive network of nodes compared to prior implementations.

Our contributions are manifold: (1) We propose a novel chunking algorithm that markedly influences the height and structure of the Prolly tree. (2) We unveil a design modification that ensures random insertions into the Prolly tree exert minimal impact on its structural integrity. (3) We introduce an innovative tree backbone design aimed at accommodating a sequence of insertions without impacting the tree's existing structure or its performance. (4) We provide an open-source implementation of the optimized Prolly Tree, complemented by comprehensive performance testing to evaluate its efficiency and scalability.

In our experiments with the Prolly Tree implementation on commodity hardware, we observed a minimum time improvement of around 30% compared to existing implementations. The structure of this paper is organized as follows: Section 2 provides an overview of the CRDTs focused on Prolly tree-related data structures. In Section 3, we introduce our proposed design for the optimized Prolly tree, detailing how it diverges from existing models. Section 4 details the implementation of the optimized Prolly Tree in Python and Golang, accompanied by a discussion of various performance metrics. Section 5 reviews relevant prior work on CRDT solutions akin to Prolly trees. We conclude the paper in Section 6, where we summarize our findings and outline directions for future research.

2. Background

In this section, we briefly explain the fundamentals of the CRDTs in the context of Prolly tree, a brief introduction to the tree-based technology that facilitates data indexing and integrity preservation.

2.1. Conflict-free Replicated Data Types (CRDTs)

Conflict-free Replicated Data Types (CRDTs) are a class of data structures designed to facilitate reliable data replication across multiple nodes in distributed systems, without necessitating central coordination [1]. The fundamental idea of CRDTs is to permit updates at various nodes independently, ensuring that such updates can be consolidated in a manner that deterministically resolves any conflicts, thereby achieving eventual consistency throughout the system. CRDTs are particularly valuable in settings where network partitions are common or where latency and continuous availability are critical concerns, such as in decentralized applications. A set difference is calculated between two instances of a CRDT, and the resulting data can be applied to synchronize the original data layers.

2.1.1. B-Tree and N-ary Tree

B-tree [14, 15] is a self-balancing tree data structure that maintains sorted data, allowing for efficient insertion, deletion, and search operations, even in large datasets. They scale gracefully with the size of the dataset, and are widely utilized in the implementation of databases and filesystems. Each node in a B-tree contains a number of keys that are in a specific, sorted order and can have a variable number of child nodes within predefined limits. This structure not only ensures that the tree remains balanced, with all leaf nodes at the same depth, but also minimizes disk accesses—making B-trees particularly efficient for storage systems that rely on sequential block access. Each internal node has at most 2 children in the case of B-tree.

Similarly, n-ary trees share the same foundational properties as binary trees, with the key difference being their capacity to have at most n children, as opposed to the binary tree's limit of two. This significantly reduces the height of the tree but impacts the width of the tree. For B-tree operations such as search, insert, and delete have $O(\log_2 N)$ complexity where N is the number of keys in the tree. For n-ary trees, operational complexity is $O(\log_n N)$.

2.1.2. Merkle Tree

Merkle trees, serve as a cryptographically secure and efficient structure for verifying the contents of large datasets [10]. They are essentially binary trees with same operational complexity in which every leaf node represents a data block's cryptographic hash, and each non-leaf node is the hash of its children. This hierarchical hashing strategy ensures that any alteration in a single data block can be quickly detected by observing changes in the root hash, facilitating efficient and secure verification processes. The sequential arrangement of leaf data within a Merkle tree is of paramount importance, as altering the position of the same data can lead to a different root hash, fundamentally changing the tree's integrity. The construction of a Merkle tree exhibits a complexity of ***O(N)***, while the verification process benefits from a more efficient complexity of ***O(log N)***, where ***N*** represents the total number of data blocks.

In the realms of blockchains and cryptocurrencies, Merkle trees play a crucial role for light clients in verifying data inclusion within a tree structure. Notably, in leading cryptocurrencies like Bitcoin and Ethereum, Merkle trees are instrumental in confirming the presence of transactions within a block. This verification process is remarkably efficient, requiring only a logarithmic number of data nodes from the Merkle tree to ascertain the inclusion of a specific transaction. Additionally, a specialized variant known as the Merkle Patricia trie [16] is extensively utilized in Ethereum, serving a pivotal function in maintaining the blockchain's state.

2.1.3. Prolly Tree

Prolly trees data structures are designed to optimize storage and retrieval in distributed databases and version control systems. First introduced by Noms [9], Prolly tree structure is a variant of the B-tree and Merkle tree, distinguished by its probabilistic approach to node splitting and merging, which enhances performance in distributed environments. Prolly trees are particularly adept at handling large datasets that require efficient synchronization across multiple clients or servers, due to their efficient data deduplication and conflict-free replication capabilities. Each node in a Prolly tree can contain a mix of data and hashes, facilitating both direct data access and verification of large data blocks without necessitating complete traversal of the tree. In the design the number of children each node can have is probabilistically deterministic.

Prolly Trees comprise of the following notable components:

- **Node:** It is the simplest unit of data storage in Prolly tree. It represents a key-value pair at Level 0 of the Prolly tree. At non-zero levels, only the key is used to traverse the tree. Each Node also stores a Merkle hash of the children nodes below it. In this paper, Node is sometimes referred to as a key-value pair or a key.

- **Level:** It is the level of the tree at which the node is present. Level 0 is the leaf level of the tree where Nodes have both key and value filled in them. At Non-zero levels, only the key is used to traverse the tree.
- **Boundary Node:** It is any node in the Prolly tree whose *node_hash* attribute falls below a certain threshold, leading to its promotion to the subsequent level in the tree. This node contains the rolling Merkle-hash of a group of non-boundary nodes present in the level just below it until the first boundary node or None is seen when moving from right to left. It is also called a Promoted Node.
- **Tail/Anchor Node:** It is a special concept where a Node (non-data) is inserted at each level of the Prolly Tree. It can also be termed as a Fake node. It acts as a backbone for the Prolly tree. It is always a boundary node by default. This is a design put forward by Canvas [11], where the position of Anchor Node is on the left side of the Tree, other Prolly tree implementations such as Dolthub [8] and IPLD [12] do not have this concept. In our proposed Prolly tree, the anchor Node is positioned on the rightmost side of each level, so that for the most recent incoming Nodes that are inserted in the tree and happen to be non-boundary nodes, their Merkle hash state is tracked with such Node.
- **Bucket/Chunk:** It is a collection of nodes that are present at the same level in the Prolly tree and on either end there are boundary nodes (left side boundary node excluded). The bucket/chunk boundary starts from the last seen boundary node in a sequence until the first seen Boundary node. The chunk is also represented using the rolling Merkle hash of all the nodes that are present in the chunk/bucket. This Merkle hash is subsequently stored in the boundary node positioned at the immediate upper level, thereby cryptographically encapsulating the chunk's state.

Throughout the rest of the document, the terms "bucket" and "chunks" are used interchangeably to refer to the same concept.

3. Optimized Prolly Tree

In this section, we first elaborate on the design of our optimized Prolly tree as per the different components involved, then examine its distinctive properties that enhance its efficiency compared to other implementations.

3.1. Chunking and Height of Prolly Tree

Before commencing the construction of the Prolly tree, it is essential to define the mechanisms for controlling the tree's width and height. The width pertains to the number of children each Node in the tree can accommodate, essentially the chunk or bucket size. The height, conversely, is determined by this width in relation to the total number of Nodes, reflecting the overall structure's vertical dimension. Width and height are crucial as they introduce a level of determinism to the Prolly tree's structure.

Adopting a probabilistic methodology for tree structure, wherein the sizes of chunks or buckets are determined by the hash of key-value pairs—essentially, the content of the tree Nodes—we put forward a model to estimate the chunk size and anticipated height of the tree. The hash of each Node (content-defined structure) is utilized to ensure that, even within a probabilistic framework, any entity in the network can reconstruct the identical Prolly tree. This model relies on the division of the hash value space into Q equally sized segments and identifies boundary nodes according to specific criteria that involve analyzing the last few hexadecimal digits of each node's hash value.

The model operates under the following assumptions:

- **Uniform Distribution:** The hash function is assumed to uniformly distribute values across its range. This assumption ensures that any segment of its output, including the last few hex digits, exhibits a uniform distribution.
- **Independent Selection:** Each node's determination as a boundary node is considered an independent event, with the probability P_b of being a boundary node defined as $P_b = \frac{1}{Q}$.

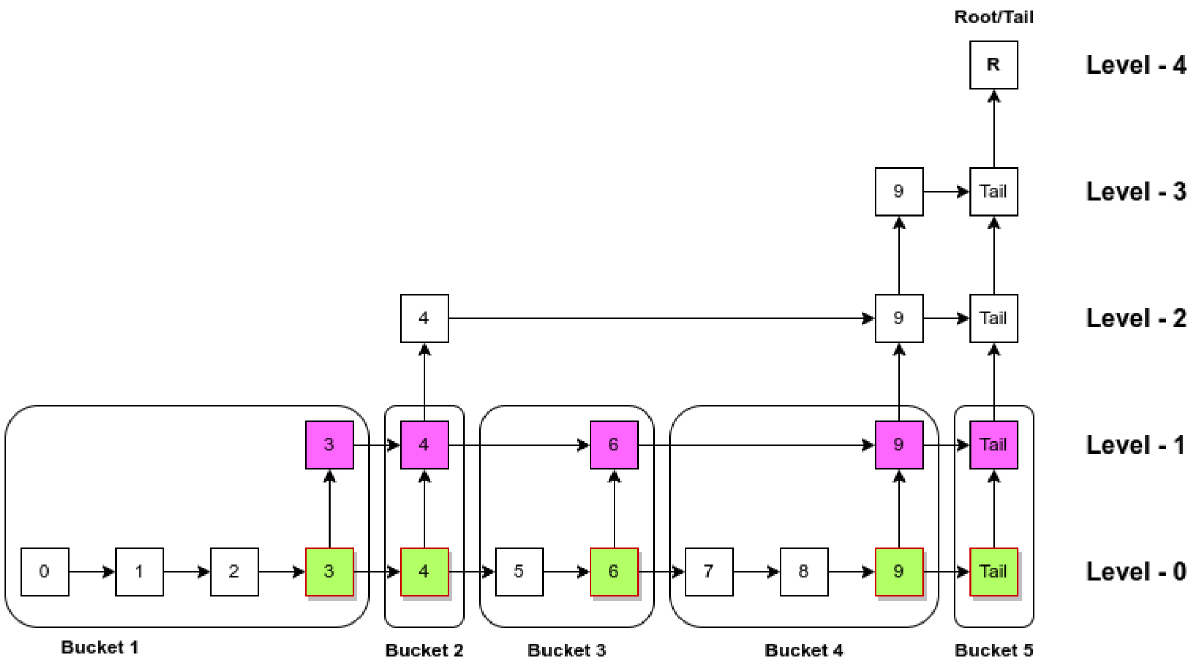

Figure 1: Proposed Prolly Tree

3.1.1. Model Description

Under these assumptions, the expected height H of the tree, for a total of N nodes, can be estimated by acknowledging the relationship between the number of nodes and the frequency at which nodes become boundary nodes:

$$P_b = \frac{1}{Q} \tag{1}$$

Given P_b, the expected height H of a tree with N number of unique Nodes can be approximated by:

$$H \approx \log_Q(N) \tag{2}$$

3.2. Construction of Prolly Tree

The construction of the Prolly tree is pivotal, serving as the foundational step to initialize the tree using existing key-value data. This data, irrespective of its origin, is seamlessly integrated into the tree's framework, demonstrating the implementation's flexibility. The process of building the Prolly tree adheres to two methodologies outlined in Base Level construction Algorithms 1 and Other Levels construction Algorithm 2. These two algorithms play a vital role in bootstrapping a Prolly tree.

3.2.1. Base Level Construction

As detailed in Algorithm 1, it begins with the incorporation of key-value data store to form the tree's base level. Each piece of data undergoes hashing with the SHA-256 algorithm, and the resultant hash is assigned as the *node_hash* attribute in a Prolly tree Node. A Node is elevated to the next level if the hexadecimal digit furthest to the right of *node_hash* falls below a predefined threshold. This elevation process is iteratively applied until all data is allocated to level 0, with a probabilistic selection of nodes ascending to level 1. Promoted nodes encapsulate the rolling hash of their child nodes from the lower level within the *merkle_hash* attribute, facilitating the identification and comparison of subtree or chunk contents. As depicted in Figure 1, an Anchor node is introduced at the extremity of level 0,

Algorithm 1 Prolly Tree Construction - Base Level

```
procedure BASE_LEVEL(data)
    Initialize the Data Store in key-value format
    Level 0 ← $\Phi$                                          ▷ No Node at initialization
    Anchor_Node ← Initialized
    while Data Store is not empty do
        Node ← key-value
        Node(node_hash) ← sha-256_hash(key-value)
        Level 0 ← Node                                          ▷ Node Added to Level 0
        if Node(node_hash) $<$ threshold then
            if Level 1 does not exist then
                Level 1 ← $\Phi$                                ▷ Level 1 Initialized
            end if
            Node(merkle_hash) ← Rolling Hash of child chunk
            Level 1 ← Node                                      ▷ Node promoted to Level 1
        end if
    end while
    Level 0 ← Anchor_Node                                       ▷ Append Anchor Node
    if Level 1 exist then
        Anchor_Node(merkle_hash) ← Rolling Hash of child chunk
        Level 1 ← Anchor_Node
    end if
end procedure
```

on the rightmost side, further elaborated upon in Section 2.1.3. The Nodes in the pink box are the boundary/promoted nodes, they are forming a chunk of their own starting from the previously observed boundary Node.

3.2.2. Other Levels Construction

For levels exceeding zero, as detailed in Algorithm 2, each Node, except the Anchor Node, situated at these elevated levels is subjected to rehashing based on the value contained in its *node_hash* attribute. The decision to promote a Node to the subsequent level again depends on whether its *node_hash* meets a specified threshold value. This promotion process is repeated until the arrangement at a given level is reduced to a single Node, in conjunction with an Anchor Node. At the final stage of this iterative process, the Anchor Node is designated as the tree's root, thereby establishing the ultimate hierarchical structure of the Prolly tree.

3.3. Updating the Prolly Tree

Updating a Prolly tree is similar to updating a conventional Merkle or B tree [17], with the additional intricacy of chunking. The update operations include inserting, deleting, or modifying a key-value pair (Node at Level 0) within the tree. As delineated in Algorithm 3, the update process entails three main steps: (1) locating the position of the key-value pair to be updated, (2) executing the update operation, and (3) updating the Merkle hash values of the impacted chunks or boundary Nodes.

Search operations play a crucial role in identifying the position of the key-value pair that needs updating. The search algorithm, denoted as *SEARCH_POSITION(...)*, operates recursively, navigating from the root node to the leaf node along the path of the key-value pair designated for updating. As specified between lines 4-8, the search process, particularly during insertion, begins at the root and proceeds downward to the leaf level to find the key that is immediately smaller than the target key. This is the reason the algorithm evaluates at intermediate Nodes whether the left Node is higher. Upon

Algorithm 2 Prolly Tree Construction - Other Levels

```
procedure OTHER_LEVELS(Level N)
    Anchor_Node ← Initialized
    while While Current Level N has Nodes do                       ▷ Iterate Over Each Node
        New_Node(node_hash) ← sha-256_hash(Node(node_hash))
        if New_Node(node_hash) < threshold then
            Promote to next level
            if Level N+1 does not exist then
                Level N+1 ← Φ                                      ▷ New Level Initialized
            end if
            New_Node(merkle_hash) ← Rolling Hash of child chunk
            Level N+1 ← New_Node                                   ▷ Node Promoted to Level N+1
        end if
    end while
    if Level N+1 exist then
        Anchor_Node(merkle_hash) ← Level N [Anchor_Node](merkle_hash)
        Level N+1 ← Anchor_Node                                    ▷ Append Anchor Node
    end if
    if No new Level (N+1) Added then                               ▷ No Node Got Promoted
        Level N+1 ← Φ                                              ▷ Final Root Level
        Anchor_Node(merkle_hash) ← Rolling Hash of child chunk
        Prolly Tree Root ← Anchor_Node                             ▷ Anchor Node of N+1 is the Root
    end if
end procedure
```

locating the target leaf Node, the algorithm inserts the Node, corrects the pointers, assesses the extent to which the inserted Node will ascend, and then refreshes the Merkle hash values of the impacted chunks. The Merkle hash update algorithm, *UPDATE_HASH(...)*, also a recursive function, starts from the leaf Node and moves up to the root Node, updating the Merkle hash values of the affected chunks (maximum two in our case). Upon insertion or any other update, Merkle hash of the affected chunks are updated first and then it moves towards the right side of the Prolly tree to update the chunks till the root node. Line 17 *Find next boundary Node* is a procedure that finds the boundary Node of each chunk on the right side of the updated site in the tree moving up in a reverse waterfall manner to efficiently update the parts of the tree. During sequential updates within the tree, only the Anchor Nodes, extending all the way up to the Root Node, are affected by the Merkle hash updates(if inserted Node is non-boundary). Conversely, only the directly impacted chunk undergoes a structural update in scenarios involving random updates.

When updating content, the previously described algorithm operates similarly, except for the insertion step. In such instances, only the Merkle hash of the impacted chunk is updated. This update process is notably efficient because it ensures no other chunk is affected. On the other hand, during a deletion, the search mechanism functions in the same manner. However, with deletion, the Node is removed from all Levels, and alongside the pointers to adjacent nodes, the Merkle hash of the affected chunk is updated. In the case of deletion, there exists a probability of $P\left(\frac{1}{b}\right)$ (where b is the chunking factor) that an existing chunk will be removed and merged with an adjacent chunk.

3.4. Features of The Optimized Prolly Tree

Highlighted below are the distinguishing features of the optimized Prolly tree, contributing to its efficiency compared to other implementations:

Algorithm 3 Prolly Tree Update

```
procedure SearchPosition(Root, Node_to_find)
    Node ← Root
    Get to the level 0 right boundary Node of the subjected Node
    while $Node.down \neq None$ do
        if $Node.left \neq None$ and $Node.left > Node\_to\_find$ then
            Node ← Node.left
        else
            Node ← Node.down
        end if
    end while
    Move left until the element or lower
    while $Node.left \neq None$ and $Node.left >$ Node_to_find do
        Node ← Node.left
    end while
    return Node                                ▷ Target position is left to this Node
end procedure
procedure UpdateHashes(Node)
    if Node has any Node above then
        Update the Node's hash
    else
        Find next boundary Node
        if No Node found then
            Update operation complete
        end if
        UpdateHashes(Next Boundary Node)
    end if
end procedure
```

3.4.1. Right Side Anchor Node Design

Anchor Nodes within the Prolly tree serve as a foundational framework, effectively integrating node data pending inclusion in a chunk. This architecture significantly enhances efficiency over other designs, as illustrated in Figure 2. Here, each insertion or update influences the entire pre-formed tree through updates to the *merkle_hash*. Additionally, Figure 3 demonstrates that sequential insertions leave the tree's existing structure unaffected; changes are confined to the upgrade path, where Merkle hashes are updated following any node modification. It is a notable advantage over other designs where new changes necessitate adjustments throughout the entire tree. In contrast, in our design, only the Anchor Nodes across various levels are subject to modifications, thereby streamlining the update process and maintaining the integrity of the tree's structure.

3.4.2. Subtree Segmentation

The proposed Prolly tree design features an innovative approach for segregating pre-established tree chunks, enabling these segments to operate independently as subtrees. This capability is crucial, considering that a node's height remains uniform across the network, thereby allowing these chunks to autonomously function. As shown in Figure 1, bucket 4 exemplifies how a chunk or subtree can serve as a standalone unit of comparison. Such functionality becomes particularly advantageous for lightweight client applications unable to store large sized Prolly trees, exemplified by clients within Internet of Things (IoT) blockchain ecosystems [18]. This segregation feature significantly enhances the efficiency of synchronization activities across Prolly trees with varying heights. As detailed in Section 3.4.1 and Figure 3, in scenarios involving sequential updates, existing tree chunks can be segregated

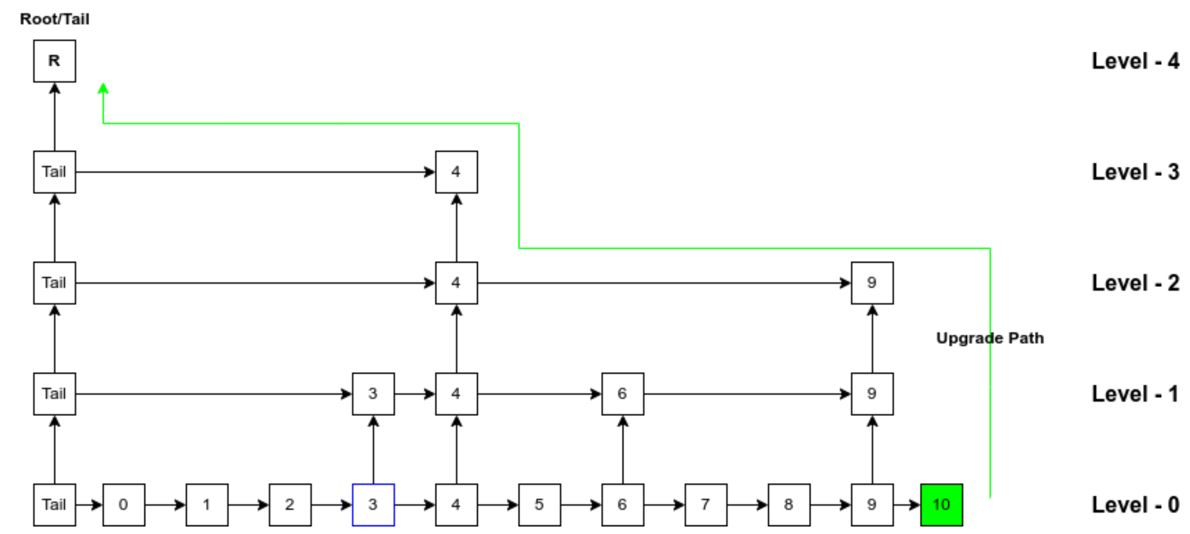

Figure 2: Left side Anchor Node

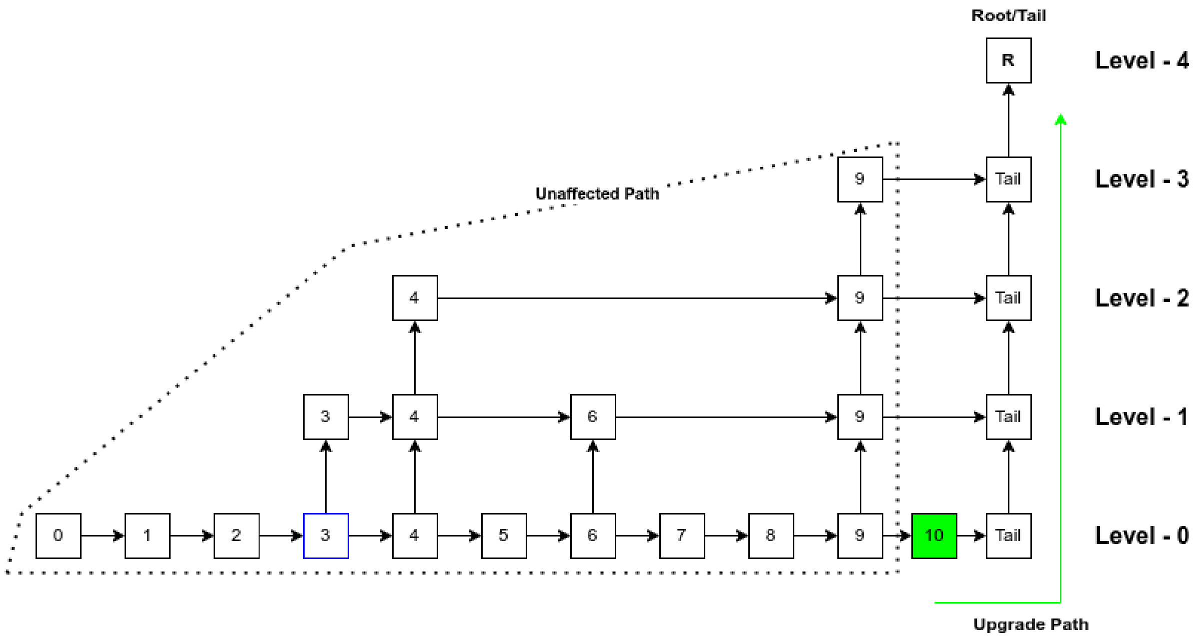

Figure 3: Right side Anchor Node

from the primary tree structure. This allows for updates to be confined to Anchor Nodes or newly developed subtrees.

3.4.3. Batch Insertion

A crucial innovation of the proposed Prolly tree is its support for batch insertion, a mechanism that allows for the simultaneous insertion of multiple key-value pairs. This method is akin to embedding a miniature subtree within the larger tree structure, markedly enhancing efficiency by diminishing the requisite number of updates. The implementation of a sophisticated chunking algorithm alongside a thoughtfully designed Anchor node system underpins this capability. This approach, favoring the collective insertion of a subtree over individual additions, streamlines the entire process. Notably, this feature proves invaluable for the synchronization of Prolly trees; insertions made between boundary nodes obviate the need for the intricate restructuring typically necessary in other models, thereby facilitating a more seamless integration process.

4. Implementation

In this section, we provide the implementation details of the proposed Prolly tree along with a few metrics over which it outperforms the existing designs.

4.1. Python and Go Implementation

Our proposed Prolly tree design, as detailed in Section 3, is available through an open-source implementation. For this project, we selected Go due to its efficiency and speed, and Python for its simplicity, readability, and ease of modification. We have structured our implementation into three fundamental components of the Prolly tree, aiming to optimize both performance and accessibility in the development process.

4.1.1. Node Class

This entity signifies a Node within the Prolly tree, characterized by attributes including key, value, pointers to other Nodes within the tree, merkle_hash, node_hash, among others. Each Node serves as a self-contained unit, enabling traversal throughout the entirety of the tree by leveraging these interconnected components.

4.1.2. Level Class

This component embodies a specific level within the Prolly tree, distinguished by attributes like the level number and the tail node of that level. The Level functions as a repository for Nodes situated at a distinct tier within the tree, acting as a modular unit that can expand or contract according to the tree's structural needs.

4.1.3. ProllyTree Class

This constitutes the operational core of the Prolly tree, integrating both Node and Level components to construct the tree structure. It is equipped with utility functions facilitating essential operations such as traversal, insertion, deletion, and the updating of Merkle roots. Due to its component-based modularity, this unit is highly programmable, allowing for swift adaptations in the tree's behavior to meet various requirements.

For testing and demonstration purposes, we have introduced a Message class designed to generate message objects for insertion into the tree. This class is characterized by attributes such as key and value, facilitating the creation of message objects. Additionally, we have implemented a sample diff algorithm tasked with comparing two Prolly trees to identify discrepancies between them. This feature not only serves to verify the tree's integrity but also showcases its capacity to manage updates efficiently. We are making our codebase for the proposed Prolly tree publicly available[1] for anyone to recreate and verify benchmarks, and to further extend capabilities.

4.2. Performance Metrics

* Prolly tree initialization data used for Canvas in Table 1 is taken directly from the publicly available figures[2] by Canvas, it is because the benchmarking code[3] by Canvas does not support the Tree initialization benchmarks. While the Dolthub and proposed Prolly tree benchmarks were executed. In the

[1]Available at: https://github.com/ABresting/Prolly-Tree-Waku-Message
[2]https://joelgustafson.com/posts/2023-05-04/merklizing-the-key-value-store-for-fun-and-profitempirical-maintenance-evaluation
[3]https://github.com/canvasxyz/okra

Table 1
Time to create different Prolly trees

Tree Type	# Entries	Min Time (ms)	Max Time (ms)	Avg Time (ms)	Std Dev (ms)
Dolthub	1,000	3.66	11.53	5.47	1.44
	100,000	247.25	325.52	285.07	22.31
	1,000,000	2841.58	3614.93	3012.03	301.99
	10,000,000	34943.06	49207.39	41398.00	5510.00
	17,000,000	47232.56	73075.68	61987.00	11016.00
Canvas*	65,536	-	-	478	-
	16,777,216	-	-	111494	-
Proposed	1,000	1.28	4.39	2.79	0.70
	100,000	184.56	241.66	210.77	18.21
	1,000,000	1964.63	2171.78	2049.41	67.67
	10,000,000	19847.38	24040.66	21623.00	1528.00
	17,000,000	30143.14	38940.05	34238.00	2985.00

case of Dolthub, data showcases a trend that the proposed Prolly tree implementation outperforms the existing solutions by around 30-45% for creating the Prolly tree. While it is around 3 times faster than the Canvas Prolly tree.

We tested the performance of the existing Prolly trees against our proposed implementation by utilizing the publicly available metrics and open-source codebases of Canvas[4] and Dolthub[5], evaluating across various metrics. Operations such as creation/initialization of a tree and insertions were performed on Prolly trees of varying capacities to assess their efficiency. The following benchmarks were conducted on hardware equipped with a 13th Gen Intel i7-13700H processor with 20 cores, 64 GB of main memory, and a 1 TB M2 SSD, all running on Ubuntu 22.04 LTS. Specifically, our approach accelerates the tree bootstrapping process from 30% to multiple-folds during creation time and demonstrates significantly faster performance in handling insertions, achieving these results on average with commodity hardware. Benchmarks from Canvas and Dolthub are also very interesting since they are running atop real-world production environments with libp2p as a network layer in the case of Canvas. The performance metrics are summarized in Table 1 and 2.

Table 2
Time to insert entries in Prolly trees

Tree Type	# Insertions	Min Time (ms)	Max Time (ms)	Avg Time (ms)	Std Dev (ms)
Dolthub	1,000	2.26	6.56	3.00	1.00
	100,000	436.79	582.03	483.00	51.00
	1,000,000	4910.55	11217.01	8632.00	2090.00
	10,000,000	61720.77	65942.95	63828.00	1723.00
Canvas	1,000	8.39	20.97	12.79	4.52
	100,000	658.71	719.643	679.25	10.49
	1,000,000	11658.26	13265.09	12268.73	552.03
	10,000,000	153112.30	159895.32	154708.51	1608.02
Proposed	1,000	1.95	4.13	2.00	1.00
	100,000	234.34	293.89	264.00	16.00
	1,000,000	4178.30	5130.71	4661.00	373.00
	10,000,000	47975.28	49631.20	48804.00	676.00

The benchmarks presented in Table 2 were generated through the execution of the publicly available benchmark codebase of both Dolthub and Canvas. In the table, the same number of unique entries was inserted into an existing tree that already contained an identical(unique also) number of entries. For instance, inserting 1,000 entries into a tree already holding 1,000 Nodes, similar to other numbers of

[4]https://github.com/canvasxyz/okra/blob/main/benchmarks/main.zig
[5]https://github.com/dolthub/dolt/blob/main/go/performance/kvbench/prolly_store.go

insertions.

5. Related Work

In this section, we explore tree-based Conflict-free Replicated Data Types (CRDTs), including designs akin to the Prolly tree that have been introduced in previous research.

CRDTs are pivotal for achieving eventual consistency within distributed systems, ensuring that datasets synchronize with minimal communication and computation overhead. Auvolat et al. [19] introduced the Merkle Search Tree (MST), an n-ary Merkle tree that shares similarities with the Prolly tree design. Their approach utilizes a chunking algorithm that hashes data (key-value pairs); the number of leading zeros in the resultant hash dictates the node's height within the MST. Hashes without a leading zero remain at the leaf node level, while those with one or more leading zeros are elevated to higher levels. Their design is reminiscent of the B-tree's node tracking methodology, albeit with a distinct structural configuration. Importantly, this construction maintains the order of transactions, facilitating efficient remote comparisons or differences with other MST instances.

The Prolly tree, as developed by Jose et al. at Canvas [11], adopts a nuanced method for determining node promotion, specifically concerning boundary node selection. This method entails assessing whether certain bytes from the data node's resultant hash fall below a predefined threshold. If so, the node is elevated to the next hierarchical level. For nodes situated above the initial level, the evaluation involves the aggregated hashes of their child nodes, proceeding until a new boundary node is identified. To enhance the stability of the Prolly tree, a unique anchor node is introduced at the beginning of every level. This node, invariably the first child of its parent, ensures structural consistency and provides a backbone to the tree, as defined in Section 2.1.3. Notably, random updates within the tree can potentially alter its structure both horizontally and vertically, extending up to the root. Such modifications, which may result in the splitting of node buckets, add a layer of complexity to the tree's architecture. Unlike traditional data structures, the Prolly tree utilizes a content-addressable order, thereby not maintaining lexicographical sequence.

The Prolly tree implementation by Son et al. at Dolthub [13] exhibits considerable similarity to the Prolly tree framework developed at Canvas. A primary distinction, however, lies in the utilization of a rolling hash for the incoming data at the leaf level to determine the initiation point for boundary node creation. Unlike the Canvas model, the Dolthub approach does not incorporate the concept of an anchor node. Instead, it features a continuous bucket of nodes that receive incoming data nodes, from which no boundary node is derived. Comparable to the Canvas Prolly tree, the Dolthub Prolly tree encounters similar complexities in tree restructuring due to random updates or insertions. With each insertion certain parts of the Prolly tree needs to update their hashes.

The Prolly tree implementation by IPLD [12] shares notable similarities with those developed by Canvas and Dolthub. It too employs a rolling hash mechanism for determining the formation of node chunks and the subsequent creation of boundary nodes. The process of splitting existing chunks aligns with the methodologies observed in both Canvas and Dolthub implementations. A distinctive feature of the IPLD implementation is its high programmability and adaptability to changes, facilitated by a commitment to component modularization. This approach enables a flexible architecture, allowing for easier modifications and updates in response to evolving requirements.

Our Prolly tree design is strategically oriented towards minimizing both the complexity inherent in the tree's structure and the necessity for its restructuring while seamlessly preserving the established properties and standards outlined in Section 2.1.3. The design decisions employed facilitate simplified updates to the tree, thereby reducing the impact on the tree's overall structure and enhancing the efficiency of parallel read and write operations. Although, the mechanisms by which two Prolly trees are compared and the methodology for calculating differences between them fall beyond the scope of this paper, but we have provided a sample execution code in the provided codebase and a brief explanation in the Appendix A.1

6. Conclusions

In conclusion, our study presents an innovative approach to optimizing Prolly trees, a pivotal structure in distributed database systems and version control applications. Building on our enhanced and optimized design, which surpasses a variety of existing approaches, we introduce several key features that significantly enhance performance and efficiency. Notably, avoiding horizontal chunk splitting, the unique Anchor Nodes design reduces the complexity of updates and synchronizations, streamlining operations while maintaining high levels of data integrity and system resilience, batch insertion, etc. Our open-source implementation, crafted in Go for its performance and Python for its ease of use, underscores the practicality and adaptability of our design.

Empirical evaluations, benchmarked against publicly available metrics and the codebases of Canvas and Dolthub, validate the efficiency of our proposed Prolly tree implementation, which adheres to the established standards and essential properties of a Prolly tree. Our approach demonstrates a 30% to 50% improvement in tree initialization time for Dolthub and multiple-fold increases for Canvas, with similar gains in insertion efficiency, all on commodity hardware. These findings underscore the robustness and efficiency of our design and its potential as a foundational model for future enhancements in distributed computing environments.

Our research contributes to the ongoing dialogue in distributed systems design, offering a practical solution that balances efficiency, flexibility, and performance. In future, we sought to develop a multi-threaded Prolly tree design towards faster operations and cutting down additional time.

References

[1] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, in: Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13, Springer, 2011, pp. 386–400.

[2] K. Jannes, Secure and resilient data replication for the client-centric decentralized web, in: Proceedings of the 23rd International Middleware Conference Doctoral Symposium, 2022, pp. 1–4.

[3] V. Buterin, Ethereum: A next-generation smart contract and decentralized application platform, Ethereum project yellow paper (2014 (accessed March, 2024)). URL: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[4] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, Cryptography Mailing list at https://metzdowd.com (2009).

[5] O. Thorén, S. Taheri-Boshrooyeh, H. Cornelius, Waku: A family of modular p2p protocols for secure & censorship-resistant communication, in: 2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW), IEEE, 2022, pp. 86–87.

[6] P. Labs, Libp2p, 2021 (accessed March, 2024).

[7] G. Wood, Devp2p wire protocol, 2014 (accessed March, 2024).

[8] T. Sehn, Prolly trees, 2024 (accessed March, 2024). URL: https://www.dolthub.com/blog/2024-03-03-prolly-trees/.

[9] A. Boodman, Prolly trees: Probabilistic b-trees, 2021 (accessed March, 2024). URL: https://github.com/attic-labs/noms/blob/master/doc/intro.md#prolly-trees-probabilistic-b-trees.

[10] R. C. Merkle, A digital signature based on a conventional encryption function, in: Conference on the theory and application of cryptographic techniques, Springer, 1987, pp. 369–378.

[11] J. Gustafson, Merklizing the key/value store for fun and profit, 2023 (accessed March, 2024). URL: https://docs.canvas.xyz/blog/2023-05-04-merklizing-the-key-value-store.html.

[12] Ipld prolly trees specification, 2022 (accessed March, 2024). URL: https://github.com/ipld/ipld/blob/prolly-trees/specs/advanced-data-layouts/prollytree/spec.md#structure.

[13] A. Son, How dolt stores table data, 2020 (accessed March, 2024). URL: https://www.dolthub.com/blog/2020-04-01-how-dolt-stores-table-data/#prolly-trees.

[14] R. Bayer, E. McCreight, Organization and maintenance of large ordered indices, in: Proceedings

of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, 1970, pp. 107–141.
[15] G. M. Adelson-Velskii, E. M. Landis, An algorithm for organization of information, in: Doklady Akademii Nauk, volume 146, Russian Academy of Sciences, 1962, pp. 263–266.
[16] H. S. de Ocáriz Borde, An overview of trees in blockchain technology: merkle trees and merkle patricia tries, University of Cambridge: Cambridge, UK (2022).
[17] D. Comer, Ubiquitous b-tree, ACM Comput. Surv. 11 (1979) 121–137. URL: https://doi.org/10.1145/356770.356776. doi:10.1145/356770.356776.
[18] S. Popov, The tangle, 2018 (accessed April, 2023).
[19] A. Auvolat, F. Taïani, Merkle search trees: Efficient state-based crdts in open networks, in: 2019 38th Symposium on Reliable Distributed Systems (SRDS), 2019, pp. 221–22109. doi:10.1109/SRDS47363.2019.00032.

A. Appendix A

A.1. Synchronizing Prolly Trees

The synchronization process initiates with a diff protocol, involving two Prolly tree root nodes from different peer-to-peer clients. For effective comparison, these roots—or trees—must be at the same height. If one tree is taller than the other, the taller tree's root node descends to match the height of the shorter tree. Notably, in our design, Anchor Nodes serve as the roots of the Prolly tree. Utilizing this design, each level's Anchor Node can act as a root for the tree below, ensuring uniform tree heights. Consequently, the taller tree selects an Anchor Node at the same level as the shorter tree's root. As illustrated in Figures 4 and 5, both trees are adjusted to the same height, facilitating the commencement of the comparison. The key node missing in this scenario is the node with key 7.

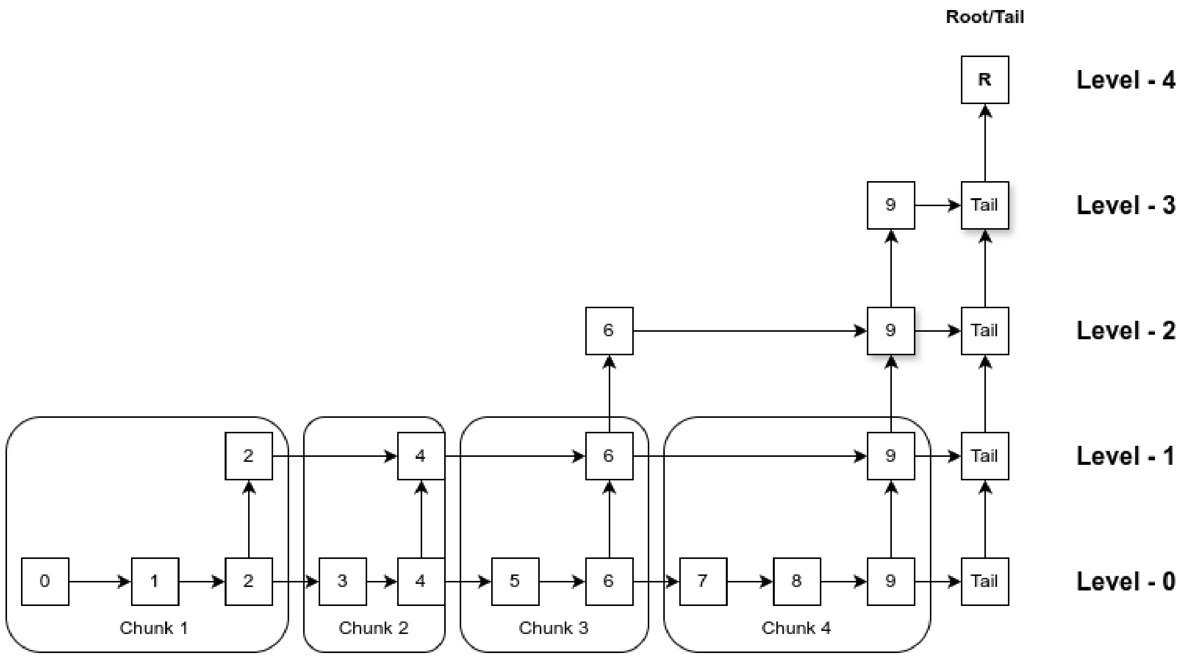

Figure 4: Prolly Tree with More Data

The synchronization process involves comparing the Merkle Hash of each root nodes. If a match is found, the algorithm concludes, as it indicates the trees are identical. If the Merkle hashes do not match, the intermediate nodes beneath the root node are retrieved. Each node at this stage represents a subtree. For example, at level 2, nodes 6 and 9, along with the Tail/Anchor Node, represent subtrees encapsulating

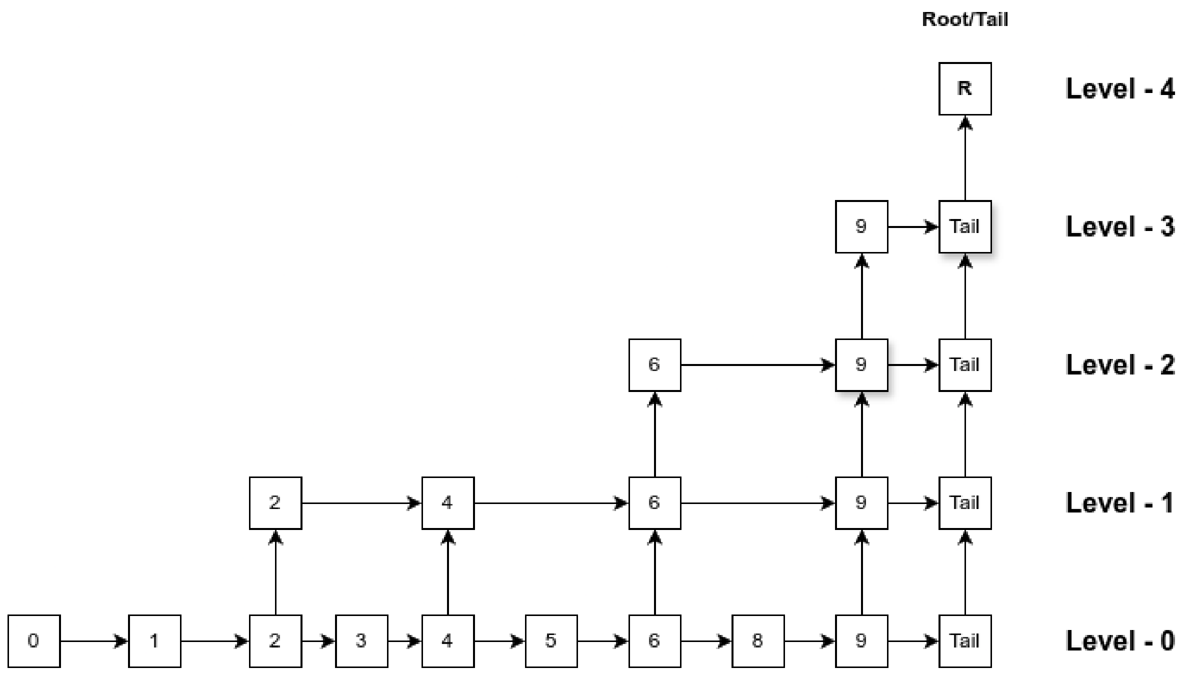

Figure 5: Prolly Tree with Missing Data

the state of all underlying nodes. Subtrees where the Merkle hashes match are not explored further. Ultimately, the algorithm isolates the specific segment of leaf nodes where the differences manifest. These nodes can be retrieved using a method tailored to the bandwidth and resources of the clients. As shown in Figure 4, node 7, located within chunk 4, exemplifies a missing node. This process may identify multiple differing nodes across several chunks, which are then synchronized accordingly.