Optimization of Merkle Tree Structures: A Focus on Subtree Implementation

Prasad Ayyalasomayajula
Department of Computational Enginering
Mississippi State University
pda54@msstate.edu

Mahalingam Ramkumar
Department of Computer Science
Mississippi State University
ramkumar@cse.msstate.edu

***Abstract*—A Merkle hash tree is an efficient data structure for generating a cryptographic commitment to a dynamic repository of records. For a database with N entries, the Merkle tree includes $2N-1$ cryptographic hashes as the nodes of the tree. With the rising prominence of such structures in contemporary applications, especially blockchain ledgers, optimal strategies for storage and retrieval of Merkle tree nodes become pivotal. This paper juxtaposes two primary mechanisms - the *linear tree* and the *subtree* - emphasizing databases potentially containing billions of records. Our discourse underscores the compelling advantages of the *subtree* method.**

I. Introduction

A Merkle hash tree is a data structure that facilitates efficient computation of a cryptographic commitment to a dynamic database of records. In such a tree, each leaf node represents the cryptographic hash of a record[15]; every non-leaf node represents the cryptographic hash of its two "child nodes." For a database with N records, the corresponding hash tree will consist of $2N-1$ hashes (nodes), including the "root node", which serves as the commitment to the entire database.

In broader terms, the leaves of a Merkle hash tree can be interpreted as independent data-blocks that require independent "processing" (i.e., read, write, insert, delete). The primary advantage of using a Merkle tree to provide a succinct, dynamic commitment to all blocks is its efficiency. Each incremental operation necessitates only $\log_2 N$ hash operations. For instance, in a database with a billion records, the verification of any record against the commitment (root hash), or updating the commitment corresponding to a write, insert, or delete operation, would only require 30 hash operations.

Merkle trees have found utility in a wide range of practical applications due to their ability to provide efficient and secure verification of the contents of large data structures. Some of these applications include:

1) **Blockchain ledgers:** Merkle trees are a fundamental component of blockchain ledgers. These ledgers record information regarding blockchain transactions in a way that makes it nearly impossible to alter or remove them. Blockchain ledgers are typically duplicated and disseminated across an entire network of computer systems. In these ledgers, a Merkle tree is utilized to summarize all transactions within a block [20].
2) **File Systems:** Certain cloud-based file systems utilize Merkle tree-based verification for data integrity [16].
3) **Application States in Blockchain Networks:** In blockchain networks like Ethereum, the state of each application executed on the network is captured by Merkle tree based structures to maintain consensus on the state of the application, by maintaining consensus on the dynamic cryptographic commitment to the application states.
4) **Distributed Systems:** Merkle trees are beneficial in distributed systems where the same data should exist in multiple places. They can be used to check for inconsistencies and to ensure data is synchronized across replicas. For instance, Apache Cassandra uses Merkle trees to detect inconsistencies between replicas of entire databases [19].
5) **Peer-to-Peer Systems:** Merkle trees are also useful in peer-to-peer systems. The root hash is obtained from a trusted source before a file is downloaded from a peer-to-peer source, enabling the retrieval of lower-level nodes from untrusted peers [21].

Given their widespread use, it is crucial to understand how to efficiently store and retrieve the nodes of a Merkle hash tree. This paper investigates two possible implementations of Merkle hash trees and compares the time required for I/O accesses and the number of page accesses needed. The two different implementations to create and store Merkle trees that are examined in this paper are:

- Linear Tree Implementation
- SubTree Implementation

A. Overview of Binary Merkle Tree and its representation

Figure 1 illustrates a Merkle Tree. For convenience, each node in a Merkle Tree is labeled with horizontal

2833-8898/23/$31.00 ©2023 IEEE
DOI 10.1109/CyberC58899.2023.00021

and vertical coordinates throughout this document. In addition, we shall refer to as the "index" of a node, it's position when the tree is scanned from top to bottom. The node with index 0, the root of the entire Hash Tree, is labelled (0,0); the node with index 1 is labelled (1,0), index 2 is node (1,1), and so on. Given that this is a binary Merkle Tree, the number of nodes in any row is a power of 2. The index of the node with coordinates (r,p) (row r with position p in the row) is

$$\text{Index}_{r,p} = 2^r + p - 1. \tag{1}$$

If the tree is stored as scanned, the "Index" is a function of the position of the node in memory. Since each node in a Merkle tree is a hash value, the Index needs to be multiplied by the number of bytes used by the Hash value for a single node, depending on the chosen Hash algorithm. For SHA2, for example, we would need to multiply the Index by 32, indicating that 32 bytes are used for each Hash Value of a node.

B. Properties of Binary Merkle Tree

The following are some of the important properties of the Merkle tree.

1) **Merkle root:** The top of the tree is the root node. $v_{0,0}$ is the Merkle root of the entire tree and represents a commitment to all nodes, and is computed as $v_{0,0} = \text{Hash}(v_{1,0} \text{ and } v_{1,1})$.
2) **Ancestor Nodes:** Every node except the root node has direct ancestors that include the node's parent node, the parent's parent, and so on, till the root of the tree. A node at row r has $r-1$ ancestors.
3) **Non-leaf Nodes:** A node can be leaf-node or a non-leaf node. A non leaf-node at row r is a "parent" of two "sibling" nodes at level $r+1$. For instance, in Figure 1, $v_{2,0}$ is the parent of left-sibling $v_{3,0}$ and right-sibling $v_{3,1}$. The value of the parent node is computed as the hash of its children, viz., $v_{2,0} = \text{Hash}(v_{3,0} \text{ and } v_{3,1})$. While hashing it is important to place the left-sibling to the left of the right-sibling as $\text{Hash}(v_{3,0}, v_{3,1}) \neq \text{Hash}(v_{3,1}, v_{3,0})$.
4) **Non-leaf Nodes:** A **leaf** node at level r does **not** have children at level $r+1$. Leaf nodes are computed as hashes of leaves.
5) **Leaves of the Tree:** The leaves of the tree represent the actual data captured by the Merkle tree. It is convenient to think of each leaf as representing a record of a database. There exists a unique leaf node for every leaf. Thus, if a node with coordinates (r,p) is a leaf node, we also represent the corresponding leaf by it's leaf node coordinates (r,p).
6) **Complementary Nodes:** For any node at row r there exists an r-tuple of nodes that are considered as complementary. The complementary nodes include the sibling of the node, and the siblings of it's $r-1$ ancestors. In Figure 1 the complementary nodes for leaf-node $v_{4,5}$ are shown in red. This includes the set of nodes ($v_{4,4}$, $v_{3,3}$, $v_{2,0}$, $v_{1,1}$). Nodes complementary to a leaf node are also often seen as nodes complementary to the leaf represented by the leaf node. Thus, the 4-tuple ($v_{4,4}$, $v_{3,3}$, $v_{2,0}$, $v_{1,1}$) is considered as complementary to the leaf at position $(4,5)$. Note that complementary nodes can be right or left siblings ($v_{4,4}$ and $v_{2,0}$ are left-siblings, while $v_{3,3}$ and $v_{1,1}$ are right siblings).
7) **Existence of a Leaf:** Applications that use Merkle-tree employ two parties - prover and the verifier. The prover maintains the entire Merkle Tree and its associated leaves. The verifier only maintains the current root of the Merkle Tree. To prove the existence of a leaf, the prover provides the leaf along with all its complementary nodes. The verifier computes the Hash of each parent node using the leaf Hash and its complementary nodes. Only if the leaf truly exists in the tree will the resulting root hash from the computation match the current root of the Merkle tree.

In general, the most important Merkle tree related process, for proving existence of leaf $L_{r,p}$ (with corresponding leaf node $v_{r,p}$), can be represented as

$$\tilde{v}_{0,0} = f_{mt}(v_{r,p}, \mathbf{c}_{r,p}) \tag{2}$$

where $\mathbf{c}_{r,p}$ is the r-tuple of nodes complementary to $v_{r,p}$, and $\tilde{v}_{0,0} = v_{0,0}$ only if $L_{r,p}$ exists in the tree with root $v_{0,0}$.

This operation involving a sequence of r hash operations can also be used for updating the value of a leaf, for inserting a leaf, or for deleting a new leaf.

For updating a leaf from it's current value $L_{r,p}$ to (say $L'_{r,p}$, or equivalently, to modify the value of the corresponding leaf node from $v_{r,p}$ to $v'_{r,p}$, the new root is computed as

$$v'_{0,0} = f_{wt}(v'_{r,p}, \mathbf{c}_{r,p}) \tag{3}$$

using the same set of nodes complementary to $v_{r,p}$. More specifically, the function $f_{wt}()$ updates all r ancestors of $v_{r,p}$ (one of which is the root of the tree).

Given the existence of leaf $L_{r,p}$ (or leaf node $v_{r,p}$), a new leaf can be inserted by pulling down $v_{r,p}$ to row $r+1$ and inserting a new leaf node as it's sibling. More specifically, if $tmp = v_{r,p}$ and $v = Hash(L)$ is the leaf node corresponding to a new leaf, the new leaf L is inserted as the right sibling of the node that leaf that was pulled down. Now the new value $v_{r,p}$ is

$$v'_{r,p} = \text{H}ash(tmp, v), \tag{4}$$

and the updated root is $f_{mt}(v'_{r,p}, \mathbf{c}_{r,p})$ (computed using the same complementary nodes). The newly inserted leaf

has coordinates $(r+1, 2p+1)$. On the other hand, if the newly inserted leaf is a left sibling then

$$v'_{r,p} = \mathrm{H}ash(v, tmp), \quad (5)$$

and the inserted leaf has coordinates $(r+1, 2p)$.

In other words, inserting a leaf involves converting a leaf node to a parent node. Coversely, deleting a leaf involves converting a parent node to a leaf node. The nodes updates required for insertion/deletion of a leaf is the same as updating the leaf node that is pulled down for adding a sibling, or updating the leaf node that is being pushed up for purposes of deletion of a sibling.

C. Organizing and Storing Merkle Tree Nodes

In practice, efficient implementation of Merkle tree prover-verifier protocols requires the prover to efficiently access complementary nodes. Specifically, for a tree with N leaves this would call for accessing $\log_2 N$ complementary nodes. Furthermore, scenarios involving updates to leaves or insertion/deletion of leaves, also calls for updates to $\log_2 N$ ancestor nodes.

Efficient mechanisms for reading / writing nodes will need to consider the number of memory pages that need to be accessed for each operation. This is especially true in scenarios involving very large trees where it may not be possible to store all $2N - 1$ nodes in memory, and may need to be accessed from non volatile storage (like a disk).

In this paper we assume that nodes are stored in memory with a page size of 4K, and memory pages may be flushed to the disk as and when deemed necessary by the operating system.

Alternately, as records (or leaves) are typically stored in a database, it may also be useful to store the nodes in a database too. In this case each node (32-byte SHA2 value) can be the value of a record with (r, p) position used as the record key. However, to minimize the number of database queries, it may be beneficial to store multiple nodes in each 'blob'/record. In the latter case we once again assume that up 128 nodes in record blobs of size 4K ($128 \times 32 = 4096 = 4K$).

This paper outlines two methods, viz., linear method and sub tree method, to organize the nodes within a page/blob and compares these two methods with respect to memory usage and I/O access times.

II. Implementation of the Linear Tree

In the implementation of the linear tree, all nodes are placed sequentially into memory. Considering the utilization of SHA-2, each hash value would occupy 32 bytes of memory. Consequently, node(0,0) would occupy bytes 0 to 31, followed sequentially by node(1,0), node(1,1), and so forth.

A. Memory Mapping in Linear Tree

For simplicity, it is assumed that memory is allocated in 4K pages. These pages are assumed to be contiguous blocks, even if the operating system allocates pages from different locations. In the context of a Linear Merkle Tree, nodes occupy contiguous locations across different pages, as illustrated in Figure 3. Once a page is fully allocated to nodes, the next page is used. For instance, after allocating node 126 with (r,p) coordinates (6,63) as the penultimate node, node 127 with (r,p) coordinates (7,0) occupies the final 32 bytes of page 0. The following node, number 128 with (r,p) = (7,1), is allocated on Page 1, occupying the first 32 bytes of this page.

B. Inserting a Node in the Linear Tree

In the context of a Linear Merkle Tree, node insertion is done sequentially. Here, we detail the steps and formulas used for this insertion method.

Insertion Process:

1) **Determine Position for New Node:** Initially, determine where the new node should be placed. If the tree is empty, it starts at $(0, 0)$. If not, it will be based on the last node's position.
2) **Pull Down Existing Node:** Before adding the new node, the existing node at the identified position is "pulled down" to the next level $(r+1)$. The p value gets doubled.
3) **Insert New Node:** The new node takes the position to the right of the pulled down node. Given that it's placed to the right, the r value will be the same as the pulled down node, and the p value will be incremented by 1.
4) **Check Page Overflow:** Given the constraints of page size and node size, verify if the addition causes the current page to overflow. If so, the system must move to a new page or utilize an overflow handling mechanism.

C. Accessing Complementary Nodes in the Linear Tree

Given that a 4K page can accommodate 128 nodes, identifying the complementary nodes of a leaf node involves finding the leaf node's parents up to the node in row 1. As illustrated in Equation II.I, a node number (starting from 0) can be represented by the following formula where r is the row number and p is the position of the node on row r. As the row number increases (r becomes larger), more pages are needed to fit all nodes that belong to a row. This increases the distance between a leaf node and its parent node, meaning that the child and parent will not be on the same page. This is reflected in multiple page accesses when complementary nodes are read or when leaf nodes are updated.

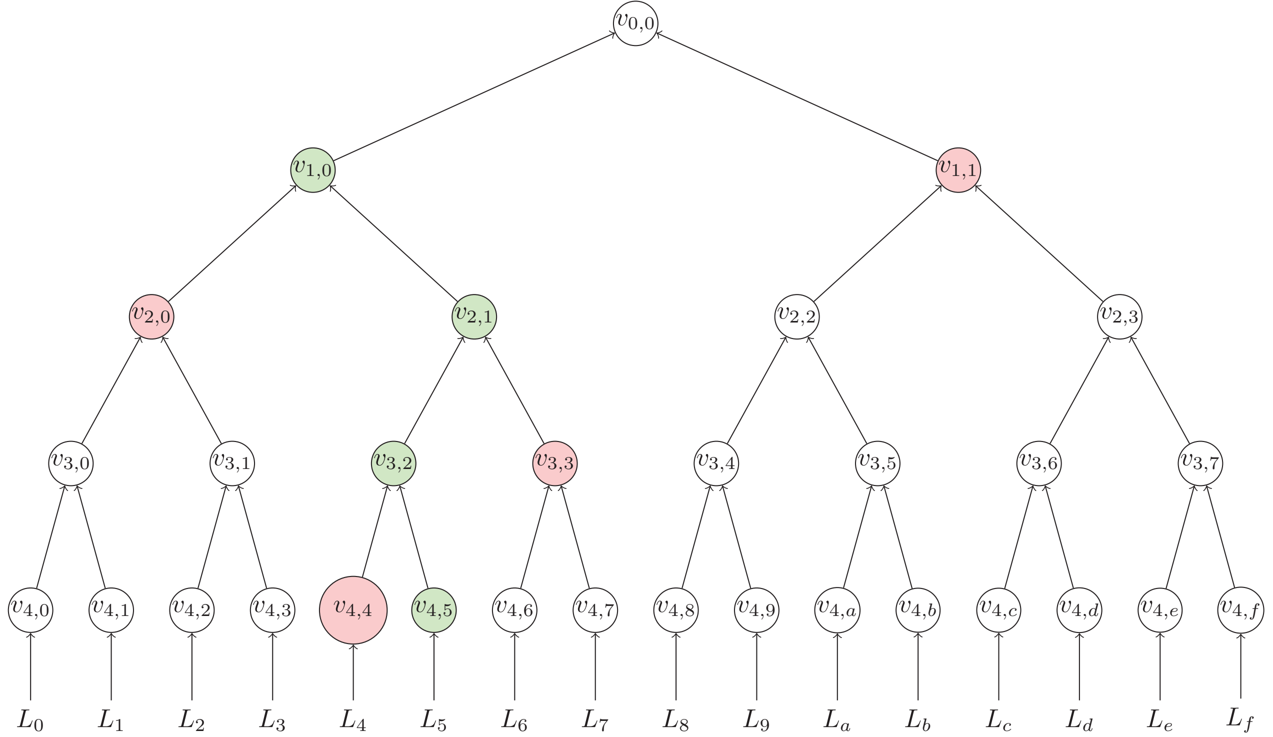

Figure 1. A Merkle Hash Tree with depth 4.

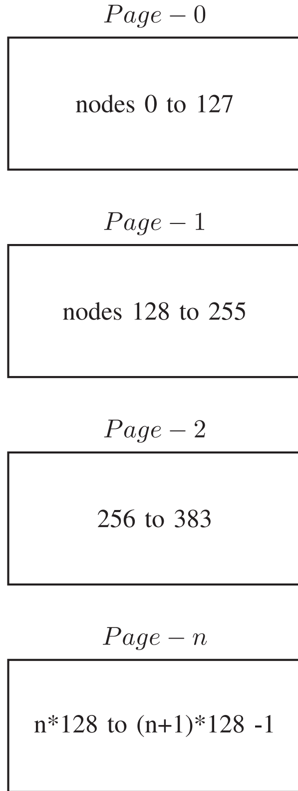

Figure 2. Each 4K Page with nodes arranged linearly

III. Sub Tree Implementation

In this method of Merkle Tree implementation, pages are used to allocate a subtree of nodes. Assuming a Page size of 4K, each SubTree can accommodate up to 128 nodes. To optimize page access, the following approach is taken:

1) **Page Zero:**Since the Page size is 4k and each node occupies 32 bytes, a maximum of 128 nodes can fit on each page. A SubTree of depth 6, from position (0,0) to (6,63), consists of a total of 127 nodes (not 128 nodes). There are 32 bytes of empty memory remaining.
   - The root starts at level 0, so Page Zero contains nodes from level 0 to level 6.
   - The page has 127 nodes, representing a complete binary tree with a height of 7, i.e., $2^7 - 1 = 127$.
   - In this tree, level 6 would be the leaf nodes.
   - The number of leaf nodes in a complete binary tree at its bottom level is $2^6 = 64$.
2) **Subsequent Pages:** Page 1 and beyond will each contain only 126 nodes, and 64 bytes of memory will be left empty.
   - The pull-down method means the root of the previous subtree (from Page Zero) becomes the first node (or the root node) in the new

subtree on the subsequent page. This root node occupies one position.

- Hence, subsequent pages will have a total of 126 nodes, considering this root node.
- The structure with 126 nodes will not be a complete binary tree as it's missing one node (that would make it 127). The missing node would be one of the leaf nodes from the subTree above (the root from the previous subtree) is present.
- Removing one leaf node from the structure of the first page would leave us with:

$$64 - 1 = 63$$

leaf nodes.

- Therefore, the number of leaf nodes in subsequent pages will be 63.

3) Each Page will have six rows. The root node of the subtree starts from a page in a level directly above it. For example, Page-1 will have its root node at (6,0) and will descend six rows from row six, i.e., to row 11. Page-2 will have its root node at (6,1) and will also descend to row 11.

A. Memory Mapping in SubTree Implementation

In the subTree method, the first page, StubTree-0 or Page-0, is filled with 127 nodes. Afterward, a new page is allocated and a new subTree, SubTree-1 or Page-1, is created. SubTree-1 will contain node numbers from (7,0), (7,1), (8,0), (8,1), (8,2), (8,3) up to (11,0), (11,1)... (11,63). The root of this page is in Page-0 and it is node (6,0). As shown in the figure below, the root of Sub-tree-1, starting from Sub-tree-0, is node (6,0), which is colored red. All the nodes in Page-1 are filled before moving on to the next subTree in Page-2.

Similarly, Page-2 will have its root in Page-0, and the root will be (6,1). The nodes in Page-2 will start from (7,2), (7,3), (8,4),(8,5),(8,6),(8,7)... and go up to (12,64),(12,65)...(12,127). Page-65 will contain all the remaining nodes starting from row 7, i.e., from (7,126), (7,127) to row 12, i.e., from (12,4031) (12,4095). The same method continues with Page-66, where each Sub-Tree will have its root start in row 12 and it grows into row 18.

Given that Page Zero initiates at Level 0, it can hold the roots of subtrees in its subsequent levels. By Level 1, the tree grows, culminating in 64 leaf nodes at Level 6.

With the presence of 64 leaf nodes at Level 6, we can add 64 distinct subtrees, each emanating from an individual leaf node. Each of these subtrees, will consist of 64 leaf nodes.So at level 12, we will have 64×64 leaf nodes. This pattern would continue as we progress.

- At **Depth 1**, we begin with just one subtree. This subtree occupies levels from 0 to 6. Given that each subtree is capable of housing 64 leaf nodes, our inaugural subtree provides 64 leaf nodes to the overarching Merkle Tree.
- By **Depth 2**, the 64 leaf nodes from our first subtree each sprout their own individual subtree. As a result, 64 new subtrees come into being. Every one of these subtrees also comprises 64 leaf nodes. Therefore, at this depth, we add a total of $64 \times 64 = 4096$ leaf nodes.
- Proceeding to **Depth 3**, the growth trend maintains its exponential trajectory. Every single leaf node from the second depth, amounting to $64 \times 64 = 4096$, germinates its own subtree. This gives rise to 4096 new subtrees. Each one of these subtrees contains 64 leaf nodes. Doing the math, this depth contributes $4096 \times 64 = 262,144$ leaf nodes to the Merkle Tree.

Depth	Levels Occupied	Total Leaf Nodes
1	0-6	$1 \times 64 = 64$
2	6-12	$64 \times 64 = 4096$
3	12-18	$4096 \times 64 = 262,144$

Table I

DEPTH OF SUBTREES IN A MERKLE TREE

This pattern continues in the same vein. With each increasing depth, we witness an exponential surge in the number of both subtrees and leaf nodes as the Merkle Tree continues to flourish.

B. Inserting New Nodes in SubTrees

Inserting a node typically involves splitting a leaf node into two new nodes, then pulled down node the parent node to the left node, and then the new node to the right. As shown in Figure 3, insertion involves pulling a leaf node down one level and adding the new node as sibling. Say if the node is $v_{(r,p)}$, then we pull it down by one level and create two nodes, $v_{(r+1,2p)}$ and $v_{(r+1,2p+1)}$. The hash value of the leaf node $v_{(r,p)}$ will be copied into $v_{(r+1,2p)}$, and the new node is inserted into $v_{(r+1,2p+1)}$. Assuming each SubTree goes seven levels deep and following the above method for insertion, after we pull down the node $v_{5,31}$, which results in filling up $v_{6,62}$ and $v_{6,63}$, the current SubTree ends. The insertion of the next node requires a new SubTree to be started i.e, the 65th node requires new SubTree to add $v_{7,0}$ and $v_{7,1}$.

C. Accessing Complementary Nodes in SubTree

In the SubTree implementation, assuming a 4K page size, we can get hold of six complementary nodes per page. This increases the efficiency of reads and writes in the SubTree.

From the given Figure 3, consider any node labeled as $v_{(r,p)}$ with child nodes $v_{(r+1,2p)}$ and $v_{(r+1,2p+1)}$. These

nodes are typically at the next depth level and serve as the child nodes for $v_{(r,p)}$.

For example, in the figure, node $v_{2,0}$ which is formed by combining its child nodes $v_{3,0}$ and $v_{3,1}$. Similarly, node $v_{1,0}$ is formed by combining its child nodes $v_{2,0}$ and $v_{2,1}$.

Figure 3. Inserting a node into Subtree Hash Tree using Pull down method

The key operations that occur on a Merkle Tree are:

- **Insert:** A new key or a leaf node is added in an insert operation. This results in changing the hash of the parent and grandparents all the way to the root node.
- **Update:** This operation involves updating an existing leaf node's hash, which results in updating the parent's and grandparent's hashes all the way to the root node.

- **Get Complementary Nodes:** This operation involves retrieving the complementary nodes of a given leaf node. For any given leaf node, we can acquire the complementary nodes required to construct the parent hashes along the path to the root. For example, in figure 2, the complementary nodes for node $v_{4,5}$ are colored in red. The resulting operation to retrieve the complementary nodes provides $v_{4,4}$ on level 4, $v_{3,3}$ on level 3, $v_{2,0}$ on level 2 and $v_{1,1}$ on level 1.
- Delete: An application can delete a leaf node. Deleting a node in a Merkle tree involves removing the node and then recalculating the hashes of all parent nodes up to the root. However, please note that it's not common to delete nodes in a Merkle tree since its primary purpose is to maintain an auditable history of data changes.

Figure 4. A Merkle Hash Tree with SubTree

D. SubTree-based API Operations for Merkle Trees

- **createMerkleSubTree():** This function initializes a new Merkle SubTree.
- **insertSubTreeNode(data):** This function handles the insertion of a new node into the SubTree. The data argument is the data that needs to be added as a leaf node in the tree. It also handles updating the hashes of the parent nodes up to the root.
- **deleteSubTreeNode(data):** This function handles the deletion (marking) of a node, it takes in the data contained in the node that needs to be deleted

as an argument. It also updates the hashes of the parent nodes up to the root.

- **updateSubTreeNode(oldData, newData):** This function handles the updating of a node. The arguments are the data currently contained in the node and the data that will replace it. This function also updates the hashes of the parent nodes up to the root.
- **getComplementaryNodesSubTree(data):** This function finds the complementary nodes of a given node in the SubTree. The data argument is the data contained in the node for which the complementary nodes need to be found.
- **verifyRecordSubTree(leaf, proof):** This function verifies that a given leaf is part of the Merkle SubTree. It takes the leaf and the proof (which is the complementary nodes) as arguments.
- **getRootSubTree():** This function returns the root of the Merkle SubTree, which can be used to verify the integrity of the data.

IV. Comparison of Linear and SubTree

The following experiment was carried out to compare the pros and cons of Linear vs SubTree implementations.

The experimental steps for the Linear Tree implementation are as follows:

1) A SQL Server is set up and a table is created to hold blobs of pages for each implementation. The ODBC driver is used to connect to the SQL driver to exchange data from the application to the SQL server.
2) An application that uses the Linear Tree implementation and connects to the SQL server is created.
3) A test application that inserts leaf nodes in sequence of 25000 is created.
4) 20 percent of the inserted leaf nodes are updated, and the IO accesses and the total time required to complete the updates are measured.
5) Steps 3 and 4 are repeated, but the node count is incremented in steps of 25000.

The experimental steps for the SubTree implementation are as follows:

1) A SQL Server is set up and a table is created to hold blobs of pages for each implementation. The ODBC driver is used to connect to the SQL driver to exchange data from the application to the SQL server.
2) An application that uses the SubTree implementation and connects to the SQL server is created.
3) A test application that inserts leaf nodes in sequence of 25000 is created.
4) 20 percent of the inserted leaf nodes are updated, and the IO accesses and the total time required to complete the updates are measured.
5) Steps 3 and 4 are repeated, but the node count is incremented in steps of 25000.

Updating a leaf node involves:

1) Reading blobs from the database for each leaf node and all complementary nodes. The number of complementary nodes read in the Linear Tree implementation is much higher than in the SubTree implementation. In a SubTree implementation, up to six complementary nodes can be found in a single 4K page, whereas in a Linear Tree implementation, at most one or two complementary nodes can be found. This leads to more IO accesses in the Linear Tree implementation.
2) Writing all the pages read in the previous step back to the database after updating the nodes.

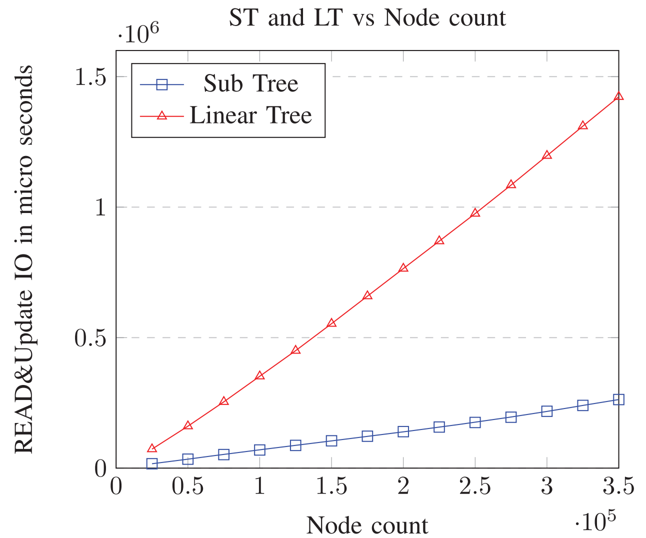

Figure 5. IO Access time for Linear Tree and Sub Tree

The provided figures offer a comparative analysis between the SubTree (ST) and Linear Tree (LT) methodologies, with specific focus on the number of pages accessed and the input/output (IO) access time required for update operations on a SQL Server. The Node Count forms the x-axis, while the y-axis corresponds to the pertinent metric values. With each method represented by a distinct color line, these charts facilitate a straightforward visual comparison of the performance of the ST and LT approaches. In Figure 5, the comparison between the SubTree (ST) and Linear Tree (LT) implementations of the Merkle Tree is depicted in terms of I/O access time. The x-axis represents the count of nodes, ranging from 25000 to 350000 nodes, while the y-axis represents the Read and Update I/O access time in microseconds.

Two series are plotted on the graph. The blue series with square markers represents the SubTree implementation, and the red series with triangle markers represents the Linear Tree implementation. As the node count

increases, it is evident that the SubTree implementation consistently outperforms the Linear Tree implementation.

For every given node count, the SubTree method results in significantly fewer microseconds for Read and Update I/O operations compared to the Linear Tree method. This shows that the SubTree method of implementing a Merkle Tree is more efficient when dealing with SQL operations, particularly when the number of nodes becomes large.

V. A Practical Application: Domain Name System

The Domain Name System (DNS) is a hierarchical and decentralized naming system for computers, services, or other resources connected to the Internet or a private network. The primary purpose of the DNS is to translate human-readable domain names (like www.example.com) in to numerical IP addresses needed for routing IP packets.

Running DNS application on a blockchain involves leveraging the decentralized and immutable nature of blockchain technology to manage domain name registrations and resolutions, and guarantee integrity of DNS records. Integrating a prover-verifier protocol ensures that the data being added to the blockchain is genuine and that participants can verify the authenticity of DNS records without relying on centralized authorities.

1) Blockchain Structure:
    - Blocks: Each block in the blockchain will contain a set of transactions related to DNS records. These records can include domain registrations, renewals, transfers, public keys, and other DNS-related operations.
    - Consensus Mechanism: To add a new block to the blockchain, participants (or nodes) must reach a consensus. This can be achieved using mechanisms like Proof-of-Work, Proof-of-Stake, or other consensus algorithms suitable for the network.
2) Domain Registration:
    - Request: A user who wants to register a domain name sends a request to the network.
    - Verification: Before adding the domain registration to a block, the network verifies that the domain name is not already registered. Furthermore, if the name to be added is say $a.b.c$ then the request for adding a new name should be authorized (using a digital signature) by the owner of the parent zone $b.c$.
    - Add to Block: Once verified, the domain registration is added to the next block to be appended to the blockchain.
3) Domain Resolution:
    - Query: When someone wants to resolve a domain name (translate it to an IP address), they query the blockchain.
    - Search: The blockchain is searched to find the latest valid record for the domain name.
    - Response: The DNS record (of type A) containing the IP address associated with the domain name is returned to the querier.
4) Updates and Transfers:
    - Domain Updates: If a domain owner wants to update their domain's IP address or other details, they can submit a new record to the blockchain. The prover-verifier protocol ensures the authenticity of the update.
    - Domain Transfers: Domain ownership transfers can also be managed on the blockchain. The original owner can sign a transfer transaction, and the new owner can verify the transfer using the prover-verifier protocol.

A. SubTree Method for DNS

DNS records are added to a Merkle Tree nodes using the SubTree method with Page Size of 4K. The table provides a comprehensive overview of the SubTree method's structure. It details the total leaf nodes, total pages occupied at each depth level, and the number of page accesses required during a read operation for each depth level for DNS application of 16,777,216 leaf nodes.

Depth	Levels	Leaves	Pages	Page Accesses
1	0-6	64	1	1
2	6-12	4096	66	2
3	12-18	262,144	4162	3
4	18-24	16,777,216	266,139	4
Grand Total			270,368	-

Table II
SubTree Method for DNS depicting the average number of page accesses required as a function of number of leaves. The number of leaves (or leaf nodes) equal the number of transactions used to create / alter the dynamic database of DNS records.

B. Linear Method for DNS

DNS records are added to a Merkle Tree nodes using the Linear method with Page Size of 4K. The table provides a comprehensive overview of the Linear method's structure, detailing the total leaf nodes, the total pages occupied at each depth level, and the page accesses required for complementary nodes for DNS application of 16,777,216 leaf nodes.

In this table, the "Page Accesses" column provides an approximation of the logarithm of the total leaf nodes for each depth level. This gives an idea of the number of page accesses required for complementary nodes. The depth level increases with each additional page,

Depth	Leaves	Pages	Page Accesses
1	64	1	$\approx \log(1) + 1$
2	128	2	$\approx \log(2) + 1$
3	192	3	$\approx \log(3) + 1$
...	...	...	...
262,144	16,777,216	262,144	$\approx \log(262, 144) + 1$
Grand Total			262,144

Table III
LINEAR METHOD FOR DNS. NOTICE THE SUBSTANTIALLY LARGER NUMBER OF PAGE ACCESSES REQUIRED COMPARED TO SUBTREE METHOD.

and the total pages occupied are directly proportional to the depth level. The grand total of pages required to represent 16,777,216 leaf nodes in the Linear method is 262,144.

Table IV provides a quick summary of the comparison between the SubTree and Linear methods, for a total of 16,777,216 leaf nodes.

Parameter	SubTree Method	Linear Method
Depth	4	262,144
Nodes per Page	126 (127 for first page)	128
Page Accesses	4	$\approx \log(262, 144)$
Grand Total of Pages	270,368	262,144

Table IV
COMPARISON OF SUBTREE AND LINEAR METHODS

From Table IV we can observe the following:

The Pros of SubTree Method are two-fold:

- Lower depth compared to the Linear method.
- Efficient for operations that require traversing the tree depth-wise.

The con of SubTree Method is that it requires slightly more pages than the Linear method (by about 1.5%).

The Pros of Linear Method are 2-fold:

- Simpler structure with a direct correlation between depth and pages.
- Slightly fewer total pages required for the same number of leaf nodes.

The main con of Linear Method is the substantially larger number of page accesses required for day to day operations.

VI. CONCLUSION

The Merkle Tree, originally conceptualized for cryptographic applications, has expanded its significance to a wide array of technological domains. It's uniqueness in providing efficient and secure verification of data contents makes it indispensable, especially in large scale database applications and blockchain technologies. An integral facet to its efficient implementation lies in optimizing the access times and IO operations, as these can significantly impact the performance, scalability, and operational costs of systems utilizing Merkle Trees.

A key approach to addressing these concerns is through the strategic segmentation of the Merkle Tree into subtrees. Such a division brings forth a more manageable and structured layout, facilitating quicker data access and reduced IO operations. This is especially consequential when we consider real-time applications where response times can significantly influence the user experience and overall system performance.

In essence, while the traditional linear implementation of Merkle Trees remains valid for certain use cases, the SubTree method offers a compelling alternative, especially for large scale applications. By marrying the strengths of the Merkle Tree's inherent security and the operational efficiencies brought about by the SubTree design, we stand on the cusp of realizing more robust, scalable, and cost-effective database solutions for the future.

REFERENCES

[1] M. Ramkumar, "Executing Large Scale Processes in a Blockchain," Journal of Capital Market Studies, Nov 2018.

[2] M.Ramkumar, "Scalable Computing in a Blockchain," The 2018 IEEE Sarnoff Symposium, Newark, NJ, Sep 23-24 2018.

[3] M. Dotan, et al. "Survey on blockchain networking: Context, state-of-the-art, challenges," ACM Computing Surveys (CSUR) 54.5 (2021): 1-34.

[4] M.Ramkumar, "A Blockchain System Integrity Model," The 17th International Conference on Security and Management (SAM'18), Las Vegas, USA, July 30-Aug 2, 2018.

[5] P. Bright, "Meltdown and Spectre: Here's what Intel, Apple, Microsoft, others are doing about it". Ars Technica. January 5, 2018.

[6] M. Lipp et.al., "ARMageddon: Cache Attacks on Mobile Devices," USENIX Security Symposium (2016).

[7] M. Ramkumar, *Symmetric Cryptographic Protocols*, Springer, August 2014.

[8] N. Adhikari, N.Bushra, M. Ramkumar, "Complete Merkle Hash Trees for Large Dynamic Spatial Data," The 2019 International Conference on Computational Science and Computational Intelligence (CSCI'19), December 2019, Las Vegas, USA.

[9] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," Advances in Cryptology, CRYPTO '87. Lecture Notes in Computer Science 293. 1987.

[10] "Ethereum. A Next-Generation Smart Contract and Decentralized Application Platform," https://github.com/ethereum/wiki/wiki/White-Paper.

[11] W. Wang, et al., "A survey on consensus mechanisms and mining strategy management in blockchain networks," IEEE Access 7 (2019): 22328-22370.

[12] S. Nakamoto, S. "Bitcoin: A Peer-to-Peer Electronic Cash System," 31 October 2008. http://nakamotoinstitute.org/bitcoin/.

[13] N. Bushra, N. Adhikari, M.Ramkumar, "TMMC: A TCB Minimizing Model of Computation," The Sixth International Symposium on Security in Computing and Communications (SSCC-18), Workshop on Multidisciplinary Perspectives in Cryptology and Information Security (CIS-2018), Bangalore, India, Sep 19-22, 2018.

[14] N. Adhikari, N. Bushra, M. Ramkumar, "Secure Queryable Dynamic Maps," Enterprise Information Systems, and e-Government (EEE'17), July 17-20, 2017, Las Vegas, USA.

[15] Wikipedia contributors, *The Comprehensive Tex Archive Network (CTAN) Merkle tree, Wikipedia, The Free Encyclopedia*,

[16] John McGuinness and Herman Tong, *Cloud File System Using Merkle Tree*,

[17] IBM, *Resilient, transparent, and trusted supply chains*,

[18] WCI, *What is a Merkle Tree*,

[19] Amit Das, *Introduction to Merkle Tree*,

[20] UTKARSH MISHRA, *The Merkle Tree And its Applications*,

[21] CodeMentor Blog, *Merkle Trees: What They Are and the Problems They Solve*,