

Protocol Foundations 004: Data proofs

Mario Havel and Tim Beiko

The digitalization of the modern world has brought an exponential surge in the amount of information computer systems must handle. Hashes, which we covered in Protocol Foundations 003: Hashing,¹ can be leveraged to structure large quantities of data in a way that allows for efficient integrity verification.

A common example is a database using hashes as indices for faster data lookup. The so-called *hash table* stores data in key-value pairs and accesses them by hashes. In *Figure 1*, we see this applied to storing phone numbers, one of hash tables' original uses dating from the 1950s.

With the digitalization of telephone directories, bibliographies, and dictionaries, hash tables enable quick lookups, even across large amounts of data, where each query's execution time is roughly equal to the speed of calculating a hash. Traditional search algorithms require scanning through entries individually which grows the search time with each added entry. The lookup function in a hash table only needs to calculate the hash of the search key that acts as an index. The index hash of a key points

to the location of its associated value and therefore the lookup requires only a single operation no matter how many entries there are. This approach is one of the basic functions used for addressing, covered in Protocol Foundations 002.²

While hash tables enable databases with more efficient data retrieval, they are not the only kind of data structure enabled by hashing. More than 25 years after the invention of hash tables, computer scientist and mathematician Ralph Merkle proposed and named another data structure composed of a tree of hashes: the *Merkle tree*.

The key innovation of Merkle trees is that they allow for efficient verification of the entire data integrity as well as any single value stored. This verification requires computing hash functions on only a subset of the data, rather than the entire dataset. In addition, one can track changes to any value in the tree by looking at a single value—its root.

Verifying a large dataset by hashing all values together would require a lot of computation, might become very expensive if the data changes often, and is inefficient

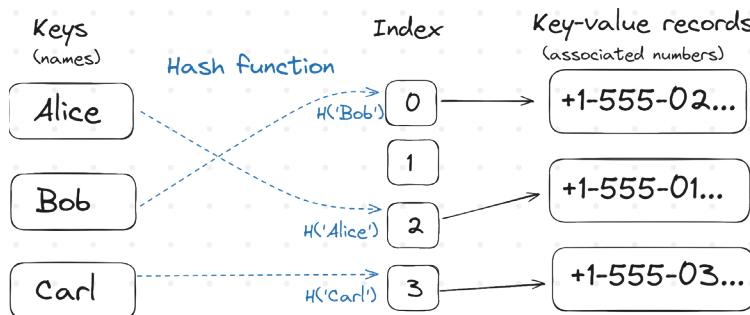


Figure 1. Hash table example for key-value pairs of a phone number database

1. Mario Havel and Tim Beiko, "Protocol Foundations 003: Hashing," Summer of Protocols (2023).

<https://summerofprotocols.com/wp-content/uploads/tktktk>

2. Mario Havel and Tim Beiko, "Protocol Foundations 002: Addressing," Summer of Protocols (2023).

<https://summerofprotocols.com/wp-content/uploads/2024/03/Addressing-Tim-Beiko-Mario-Havel.pdf>

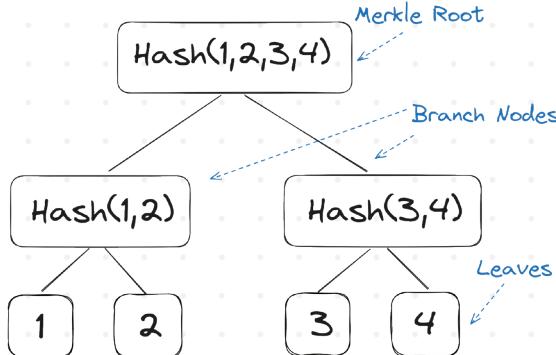


Figure 2. Merkle tree example—leaves, nodes, and the root

for verifying only a small subset of the data. Merkle trees group data into smaller logically isolated parts, each of which is hashed and can be verified individually.

Figure 2 shows how units of data, e.g., individual records, are divided into pairs and form the very bottom of the tree, called **leaves**. Each piece of data is hashed and then pairs of these hashes are themselves hashed together. Hashing together means joining them and producing a hash of joined strings and is also called **concatenation**. These intermediate hashes are called **nodes**. The process of hashing in pairs continues until there is a single **root** hash.

In this tree-like structure, leaf nodes hold the actual data values and each non-leaf node is a hash product of its child nodes. The root, at the very top, is a single hash derived from all hashes below and therefore allows for easy integrity checks on all the data. If there is a single change in any of the leaf nodes' data, nodes' hashes will propagate changes all the way up to the new Merkle root. Applying a change and updating the root only requires hashing the parts of the branch linked to the change. Computing a few intermediate hashes is much more efficient than recomputing the hash of each piece of data. For example, changing value 3 to 5 in the example in **Figure 3** only requires us to recompute hashes on the right side of the tree and the root, leaving the left side intact.

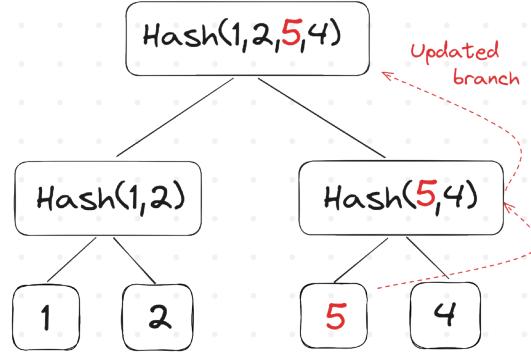


Figure 3. Example of the propagation of a changing value in a Merkle tree

Verifying the existence of a specific piece of data within this structure is also efficient because we don't need the entire dataset. To demonstrate that the data in one of the leaf nodes is included in the root hash, only the branch of its parent nodes up to the root are needed. The demonstration below shows what is necessary to verify inclusion of a single leaf node.

The data in **Figure 4** represents account balances of customers. There are 6 leaf nodes at the bottom of the tree, each includes a hash of the corresponding balance value represented by the first 4 characters of the hash. The Merkle root is calculated by hashing all data in pairs as described above. To prove that the balance 2600.00 is indeed included and correct, we only need 3 other hash values and no other data. Values marked red are the only 3 hashes needed for verification, the value to be verified is green.

The values highlighted in **red** are an externally provided proof. To verify that the leaf with value 2600.00 actually belongs where it is pictured in the image above, we calculate the Merkle root from the branch of provided data. The hash of stored data, account balance 2600.00, is represented by **4e07** and needs to be first combined with its paired neighbor **4b22**. These hashes are joined and the hash of resulting concatenation is **1365** which is then combined with **33b6** to produce **85df**. Hash

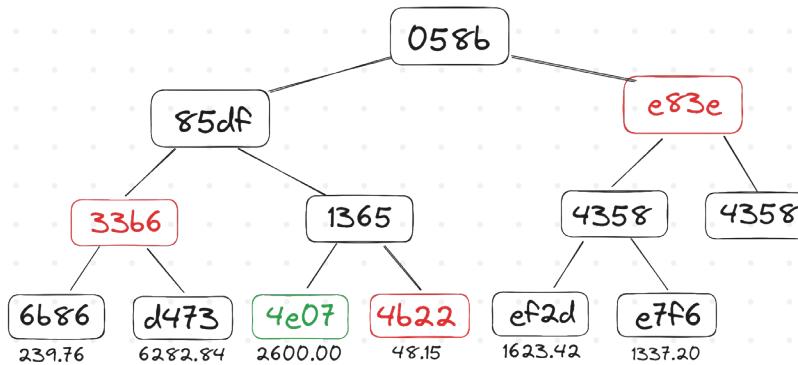


Figure 4. Example verification: green indicates the hash to be verified, red indicates the hashes needed for verification

of this value with the final hash provided in the proof, **e83e**, creates the Merkle root, **058b**. The calculation of root was done in three operations of hashing a string with no need for the rest of the tree or data values.

One more thing to notice in this example is how the last branch **4358** was duplicated and added at the end. This demonstrates the *binary* property of the tree. Because we hash in pairs, the number of hashes in a row of a binary tree cannot be odd. Because we have only 6 customer balances, which would result in 3 elements on the second row, the copied node is added to be hashed with itself and makes the list even.

Merkle trees are a fundamental building block of cryptocurrencies, which are designed around the premise that anyone should be able to verify the validity of transactions. Blockchains not only chain blocks by hashing them continuously but each of these blocks also includes a Merkle tree of its transactions. User transactions are leaves in this tree and the tree's root represents the output of all included transactions. If a single transaction in a block changes, the Merkle root will be different and all subsequent blocks would be affected. Users can therefore be confident that if other peers on the network agree with their latest Merkle root, they also agree with them on the entire blockchain's

history. In cases where two users may have different Merkle roots in their most recent blocks, they can then look at their historical Merkle roots and find the exact point at which their local copy of the blockchain varies from the other's copy.

This explainer covers the basic form of a Merkle tree. Variations like Merkle sum trees, sparse Merkle trees, Merkle Patricia trees, and others are optimized for uses like generating inclusion proofs, key-pair storage similar to databases, efficient verification, low storage space, and so on. IPFS, also mentioned in Protocol Foundation 002 on addressing, uses Merkle Directed Acyclic Graphs (DAGs) to retrieve distributed chunks of data. In the Merkle DAG, file data make up the nodes. The root, which is then the hash of the entire content, can be used by nodes in the network to efficiently retrieve pieces of data and verify that they indeed belong to the requested file.

MARIO HAVEL & TIM BEIKO are part of the Ethereum Foundation's Protocol Support team which helps facilitate Ethereum network upgrades as well as other protocol-related initiatives, including Summer of Protocols.

MARIO | github.com/taxmeifyoucan
TIM | warpcast.com/tim

ProtocolKit

summerofprotocols.com
hello@summerofprotocols.com

© 2023 Ethereum Foundation. All contributions are the property of their respective authors and are used by Ethereum Foundation under license. All contributions are licensed by their respective authors under CC BY-NC 4.0. After 2026-12-13, all contributions will be licensed by their respective authors under CC BY 4.0.
Learn more at: summerofprotocols.com/ccplus-license-2023

ISBN-13: 978-1-962872-53-9

Printed in the United States of America
Printing history: May 2024

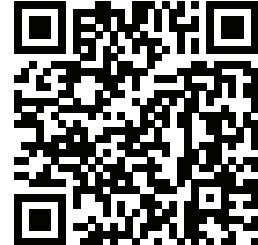
1 3 5 7 9 10 8 6 4 2

ISBN 978-1-962872-53-9



9 781962 872539

ProtocolKit



RETROSPECTUS



NEWSLETTER

