Understanding Merkle Roots

Tim Johns

January 2026

1 What is the Merkle Root?

A Merkle root is a way to compute a single hash that represents an entire collection of data. It acts like a fingerprint: if any element in the collection changes, the Merkle root changes as well. This enables *Merkle proofs* (proofs of inclusion), which let you prove an element is in a collection efficiently and without requiring trust between the requester of the proof, and the provider.

2 How to Generate a Merkle Root

Suppose we have a collection of names:

```
["Bob", "Alice", "Charly", "Sarah", "Evan"]
```

We begin by computing the hash of each element in the collection, using a hash function such as SHA-256. Since we have an odd number of elements, we simply duplicate the last item. This is required in order to cleanly arrive at a merkle root.

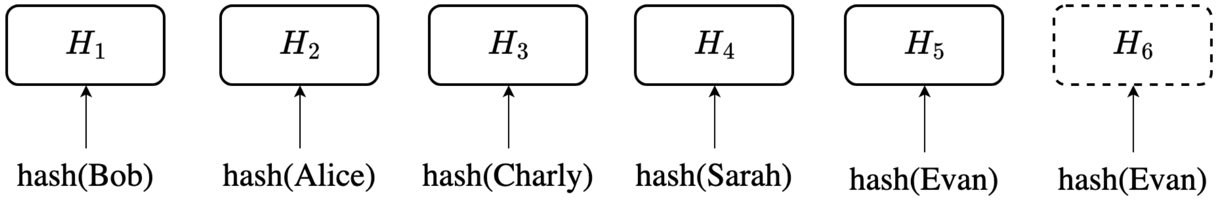

Figure 1: Hash each element to form the leaf hashes. Note that we duplicated the last element to get an H_6 hash.

Now, with out initial hashes, we calculate the next row of hashes. To do this, we simply pair the elements off (that is why we had to duplicate the last element). For each pair, we concatenate the hashes (that is, add one to the end of the other to create a long hash).

$$H_a = \text{81b637d8fcd2c6da6359e6963113a1170de795e4b725b84d1e0b4cfd9ec58ce9}$$

$$H_b = \text{f87165e305b0f7c4824d3806434f9d0909610a25641ab8773cf92a48c9d77670}$$

$$H_{H_a \parallel H_b} = \text{hash}(\text{81b637d8fcd2c6da6359e6963113a1170de795e4b725b84d1e0b4cfd9ec58ce9f87165e305b0f7c4824d3806434f9d0909610a25641ab8773cf92a48c9d77670})$$

$$= \text{48b312d7177eadac5e555698f003cdfc4e7cb7054ddfdd6c6e804f2dea588f74}$$

Figure 2: How to use a hash function to generate a new hash from two hashes via concatenation.

We then input the concatenated string of text back into the SHA-256 hash function. That hash becomes the parent node of the two hash leaves that comprise it:

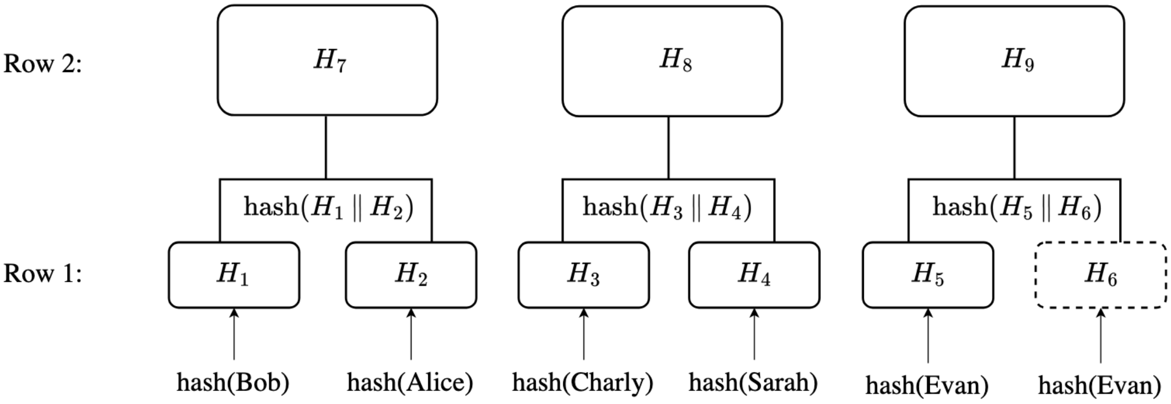

Figure 3: Create the next row by pairing the leaf-node hashes, concatenating them, then running those long text strings back through the SHA-256 hashing function.

You then repeat this process for the next row (in this case, row 2). Recall that since we now have an odd number of hashes, we'll have to duplicate the last element:

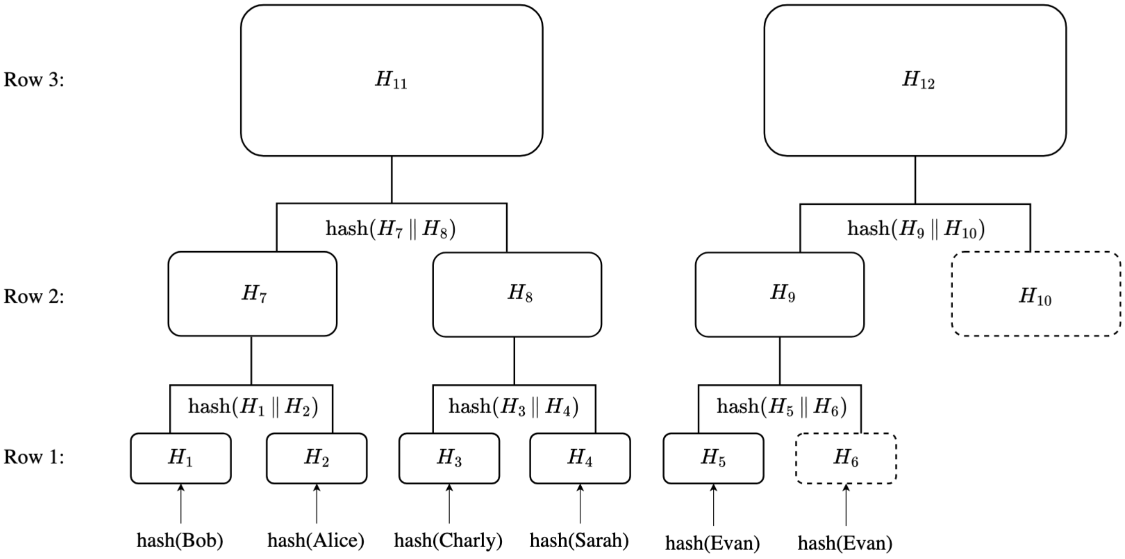

Figure 4: Repeat the hash generation process on row 2 to generate row 3.

Now, finally, do this one more time to arrive at our single hash. This root hash is the **Merkle root!**

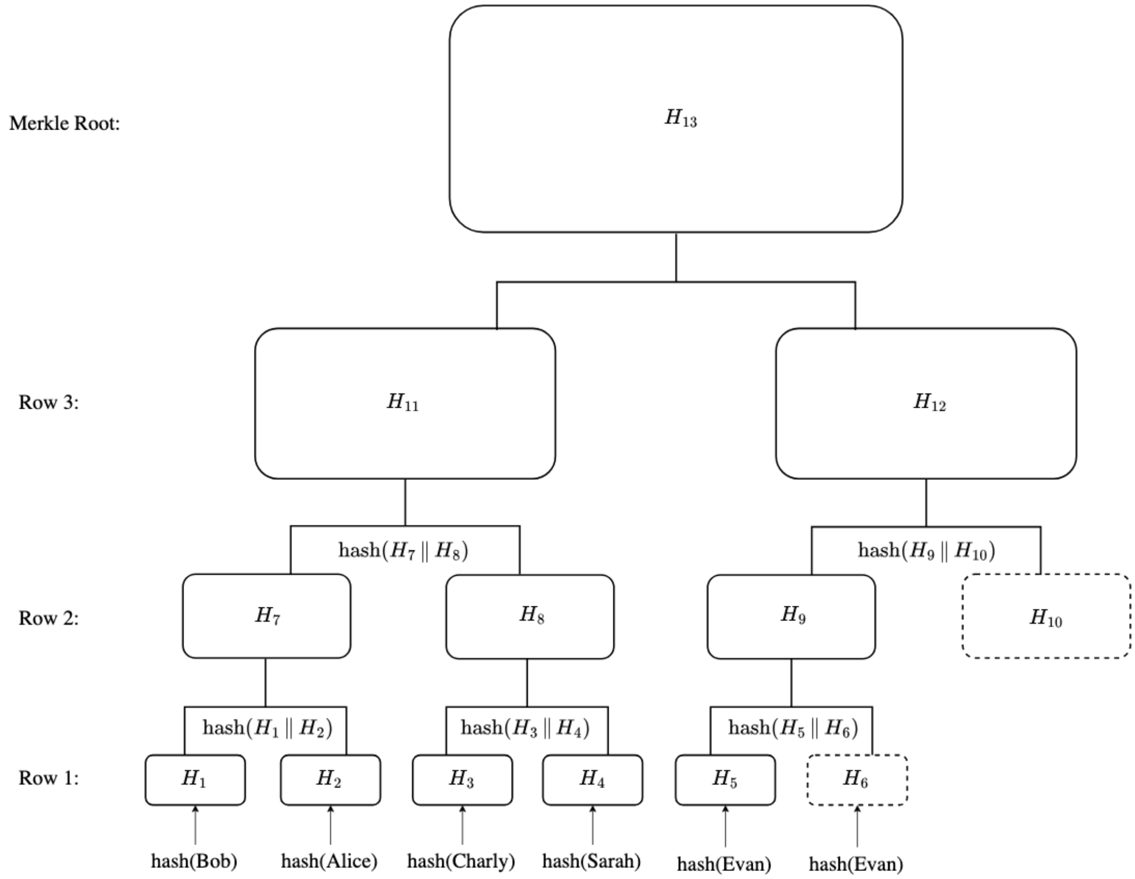

Figure 5: We know have our Merkle root (*Node* H_{13}).

3 Merkle Proofs

Now that we understand how a Merkle root is calculated, we can discuss how it is used. One of the most important applications is a *Merkle proof* (also called a *proof of inclusion*), which allows someone to verify that a specific element is part of a larger collection.

This is useful in an **untrusted setting**: Bob does not want to simply take Alice's word for it when she claims that Bob's item is (or is not) included. At the same time, Alice may not want to reveal the entire collection to Bob (for privacy reasons). A Merkle proof addresses both problems: it lets Bob verify inclusion without learning anything about the other items beyond what is necessary.

Instead of sending Bob the full list, Alice sends:

- the Merkle root of the collection, and
- a small set of intermediate hashes from the Merkle tree (the *authentication path*).

Bob then hashes *his own item* to obtain the corresponding leaf hash and uses the provided hashes to recompute the hashes along the path from that leaf to the root. If the final value he computes matches the Merkle root Alice provided, then his item is included; otherwise, it is not.

This works because Alice provides the *sibling hash* at each level of the tree along Bob's path to the root, starting from the leaf level. With these sibling hashes, Bob can recompute each parent hash efficiently and eventually recompute the Merkle root.

In our example, suppose Bob's item corresponds to the leaf hash H_1. Alice must provide the sibling hash at each level needed to reach the root: the sibling of H_1 on the first level (e.g., H_2), then the sibling of the resulting parent hash on the next level (e.g., H_8), and so on, until Bob can compute the root. Finally, Bob compares his computed root to the root Alice provided. If they match, the proof verifies inclusion.

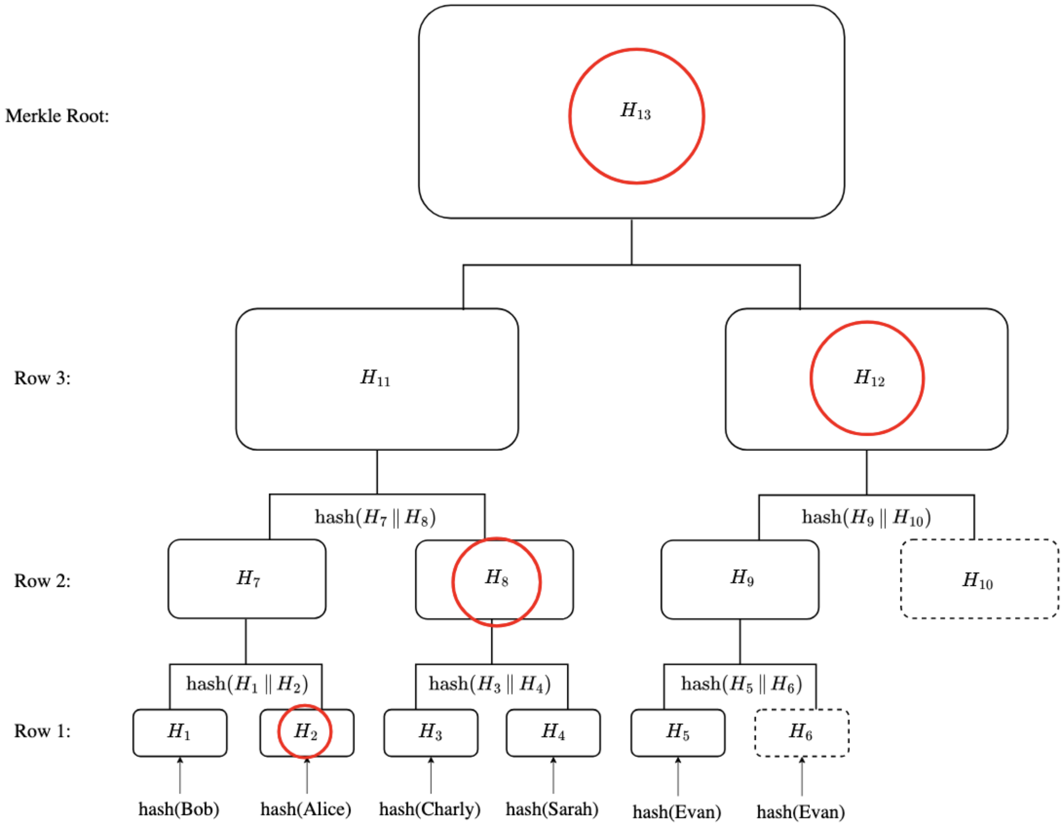

Figure 6: The hashes in red circles are the hashes that Alice shares with Bob.

Here is a fully color coded diagram that demonstrates how Bob can arrive at the Merkle root:

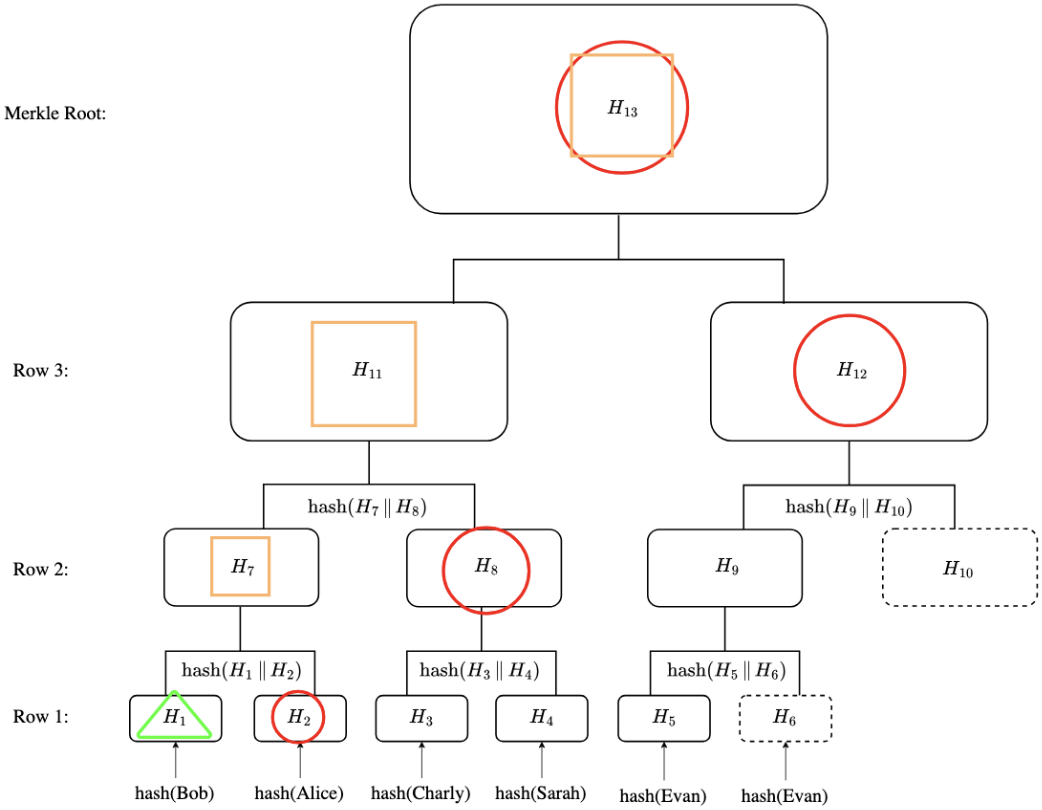

Figure 7: The green triangle hash is the hash that Bob starts with. The hashes in red circles are the hashes that Alice shares with Bob. The hashes in orange squares are the hashes Bob can calculate himself by concatenating the hash he has (green) with the hashes she shares (red).

4 Where Merkle Roots Are Used

Merkle roots show up anywhere you want a single, compact commitment to a large set of data, plus an efficient way to prove that *one specific item* is included in that set.

4.1 Bitcoin (Blocks and Transaction Inclusion)

In Bitcoin, the Merkle root is stored in each block header and summarizes all transactions in that block. Because the block header is small and widely distributed, anyone can:

- verify that a transaction is included in a particular block using a Merkle proof, and
- do so without downloading every transaction in the block.

This is the core idea behind *Simplified Payment Verification (SPV)*: lightweight clients can verify inclusion by checking only block headers plus a short proof, rather than storing the full blockchain.

4.2 Ethereum and Other Blockchains (Commitments to Large Data Structures)

Many blockchains use Merkle-style constructions to commit to more than just transactions. For example, Ethereum commits to large data structures (such as the system state and transaction-related data) by storing root hashes in each block header. This lets nodes verify specific pieces of data (balances, contract storage entries, receipts, etc.) with compact proofs, without needing the entire dataset locally.

4.3 Git (Content Addressing and Integrity)

Git is effectively a Merkle DAG: commits, trees (directories), and blobs (file contents) are linked together by hashes. A single commit hash indirectly commits to the entire repository state at that point in time. If any file content changes, the hashes change upward through the structure, making tampering easy to detect.

4.4 Transparency Logs (e.g., Certificate Transparency)

Append-only public logs often use Merkle trees so the log operator can publish a single root hash that commits to the entire log contents. Users can then request short proofs showing that an entry *is included* (or sometimes *not included*) without downloading the full log, enabling public auditability.

4.5 Distributed Storage and Data Distribution

Systems that distribute large files in chunks (or across many peers) can use Merkle roots to:

- verify each chunk as it is received,
- detect corruption or malicious peers immediately, and
- uniquely identify content by its root hash (content addressing).

This pattern appears in various peer-to-peer and content-addressed storage systems.

5 Conclusion

Merkle trees provide a compact way to commit to a large collection of data using a single Merkle root, while still enabling efficient proofs that a specific element is included. This makes them a foundational tool for integrity and verification in systems like Bitcoin, Git, and transparency logs.