Q2A – Why is the Merkle Tree better than the single hash?

Victor Agboli

May 03, 2021

Introduction to Hashing

Hashing is designed to solve the problem of needing to efficiently find or store an item in a collection. For example, if you had to look up 100, 000 names in a name register, you will agree with me that looking up each name through the 100, 000 names in the register till you find a match is a stressful and less efficient method[1]. This is where hashing comes into play as it makes this process more efficient.

Hashing means using some function or algorithm to map object data to some representative integer data. The hashing transforms data into a far shorter fixed-length value or key which represents the original string. A good hashing algorithm is computationally fast, and makes it hard to find two strings that would produce the same hash value.

Single Hash

Let's say we want to store this file (from above) in a distributed manner, we can compute the hash of the entire file, store this hash at a trusted location (server), and use it for verification going forward.

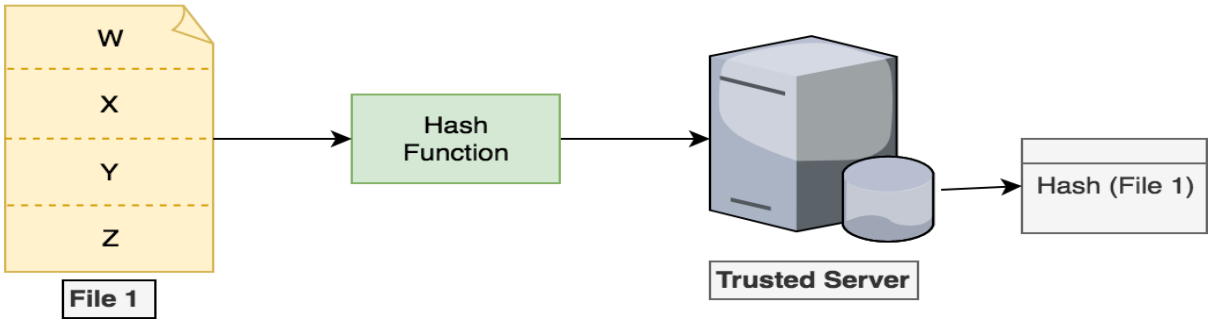

Figure 1: Merkle Trees (Satwik 2020)

Imagine after computing the hash, we handover the file to our friends; Musa, Nnamdi, and Tayo. Anytime any of them needs the file, he/she needs to download the different chunks, combine them, compute the hash of the combined file, and verify if the hash exists on our server. This is inefficient because it verifies after completion, no way of fixing the problem if verification fails, too much overhead-cost in-case we want to edit the content of our existing file because the entire file has to be hashed again, and the generated hash needs to be communicated to our server.

Merkle Tree

What if we hash the individual hashes and hash them again to obtain a **root hash**? This means we just need to store that root hash at our trusted server. This means a faster and more efficient verification process because once any chuck is downloaded, the peer can compute its hash and then its root hash, and verify if it exists in the server, easy!.

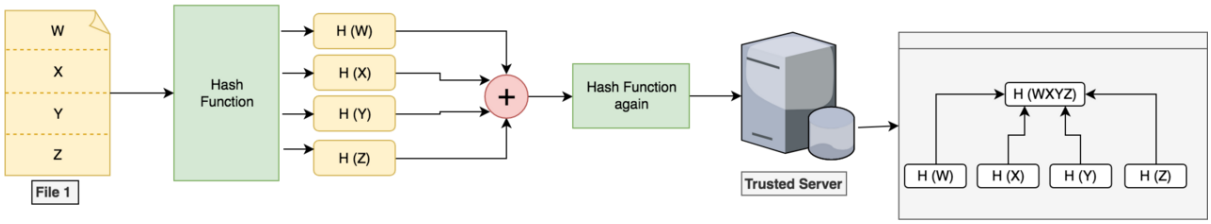

Figure 2: Merkle Trees (Satwik 2020)

What if we store a tree at the trusted server, something that looks like:

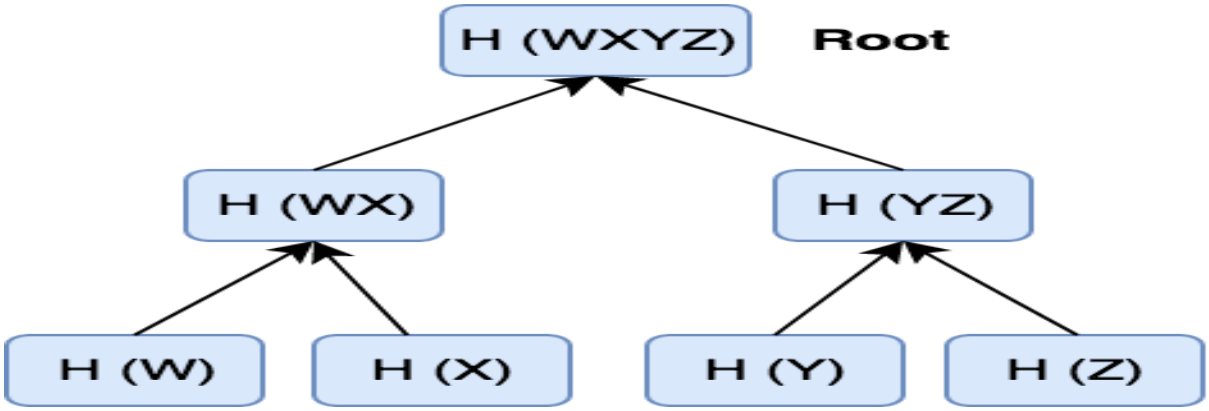

Figure 3: Merkle Trees (Satwik 2020)

The leaves are the hashes of the different data chunks of the file, and the parents of these chunks being the hashes of the concatenation. This approach is efficient because it handles the data verification process with little information required by the server and a small data verification packet size. It's also very efficient in maintaining that the log of data/file is un-tampered. It's also used to synchronize data across multiple nodes (friends in this case) in a distributed system, rather than comparing the entire data to figure out what changed[2].

In summary, the Merkle Tree is a better central data structure, especially for cryptocurrency implementations than the single hash because it allows us to carryout data verification, consistency verification, and data synchronization in a very storage – and – computation efficient way.

References

[1] https://www.freecodecamp.org/news/what-is-hashing/

[2] https://brilliant.org/wiki/merkle-tree/ - Merkle Tree. Brilliant.org

[3] Satwik Kansal, Merkle Trees: What they are and the problems they solve, accessed 3rd May 2021, < https://www.codementor.io/blog/merkle-trees-5h9arzd3n8>