An Overview of Trees in Blockchain Technology: Merkle Trees and Merkle Patricia Tries

Preprint · February 2022

CITATIONS
0

READS
6,792

1 author:

Haitz Sáez de Ocáriz Borde
University of Oxford
26 PUBLICATIONS **22** CITATIONS

SEE PROFILE

An Overview of Trees in Blockchain Technology: Merkle Trees and Merkle Patricia Tries

Haitz Sáez de Ocáriz Borde

Department of Engineering, University of Cambridge

February 21, 2022

Abstract

In this work we present an overview of trees in distributed systems and blockchain, and summarize some of the key concepts. We focus on Merkle Trees and Merkle Patricia Tries, which are used in Bitcoin and Ethereum, respectively.

1 Introduction

Blockchain technology facilitates the digital flow of information in the form of a distributed software network that aims at guaranteeing the secure transfer and exchange of units of value without an intermediary centralized institution.

The Bitcoin blockchain emerged in 2009 and was first proposed by Satoshi Nakamoto in the famous Bitcoin White Paper [1], which suggested a peer-to-peer version of electronic cash that would enable online payments to be sent directly without the need for centralized financial institutions. Although in present times Bitcoin has gathered a lot of attention in the media, blockchain technology has much wider applications beyond cryptocurrency networks: a diversity of things such as digital assets, contracts, titles or votes can be tokenized, stored, and exchanged on a blockchain network. Ethereum was proposed as a secure decentralized generalized transaction ledger aiming at implementing what Wood called a "transactional singleton machine with shared-state" in a generalised manner [2]. Ethereum has now become one of the largest, most known public blockchains.

Blockchain technology will potentially revolutionize business and transform society in the near future. In this work we present an overview of trees in distributed systems, especially focusing on Merkle Trees and their implementation in Bitcoin, and Merkle Patricia Tries which are used in Ethereum. We aim at providing a concise, clear, and understandable explanation of trees in blockchain technology, since information over the Internet seems to be scarce, hard to interpret, and sometimes contradictory.

Section 2 covers some background information on tree data structures and hash functions. Section 3 and Section 4 dive into Merkle Trees and Merkle Patricia Tries and discuss their applications in Bitcoin and Ethereum, respectively. Lastly, Section 5 summarizes the final conclusions.

2 Background

2.1 Tree Data Structures

In computer science, a tree is a data type used to represent hierarchical data. It may be defined recursively as a collection of nodes, where each node is a data structure consisting of a value and a list of references to nodes. Nodes are linked together to represent hierarchy with elements being arranged in multiple levels and hence resulting in a non-linear data structure.

Nodes are connected by edges, each node contains a value or data, and it may or may not have a child node. If a node is a descendant of any node, then that node is known to be a child node of the parent node. The first or topmost node is the root and it is the only node without a parent node. Nodes that have the same parent are siblings and nodes without children are called leaf nodes.

We can distinguish between different types of tress. In the case of general trees there is no restrictions over the number of nodes a node may contain, that is, nodes may have an arbitrarily large number of children. The children are known as subtrees and they are unordered. Another popular type of tree are binary trees, where each node in a tree can have utmost two child nodes. Furthermore, a binary search tree is a special type of binary tree in which the key of the left child node is lesser than the key of the parent node and the key of the right child node is greater than the parent node. It is called a search tree because the order of the nodes is sorted by the value of the data. Other types of trees include AVL trees, red-black trees, splay trees, treaps, B-trees, etc. Note that we could compare a singly linked list to a tree in which each node has a single child: items are linked together through a single next pointer.

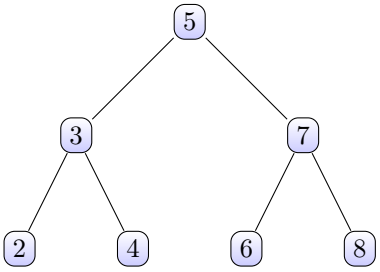

Figure 1: Simple binary search tree. The root node contains a value of 5, the leaf nodes have values 2, 4, 6, and 8. 3 and 7, 2 and 4, and 6 and 8 are siblings. 3 is the parent node of 2 and 4. 6 and 8 are the child nodes of 7.

We shall also review tries, which may alternatively be called digital trees or prefix trees. These are a type of search tree used for locating specific keys from within a given set. This tree data structure is suitable for use as an associative array, where branches or edges correspond to parts of a key. Supposing the keys come from an alphabet containing N characters, each node in the trie can have up to N children. Also, note that the maximum depth of the trie is given by the length of the longest key. Such data structure allows keys starting with the same sequence of characters to have values that are closer together in the tree and avoids key collisions, see Figure 2.

Ethereum uses a special trie, called a radix trie and combines this with a Merkle Tree, which is generally referred to as a Merkle Patricia Trie, later discussed in Section 4. Radix tries, also known as radix trees, Patricia trees, Patricia tries or compact prefix trees are a type of data structure that represents a space-optimized trie in which each node that is the only child is merged with its parent, condensing common prefix parts. Effectively a radix trie is a compressed version of a trie, see Figure 3.

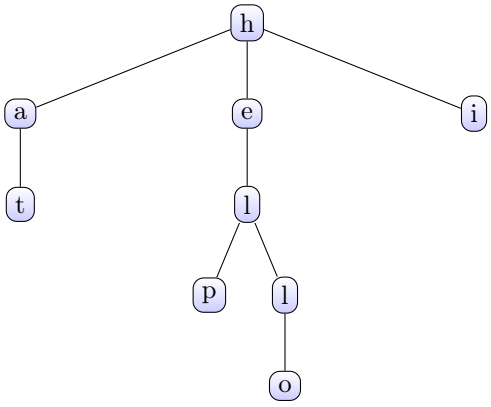

Figure 2: Sample trie, which stores the words `hat`, `help`, `hello`, and `hi`.

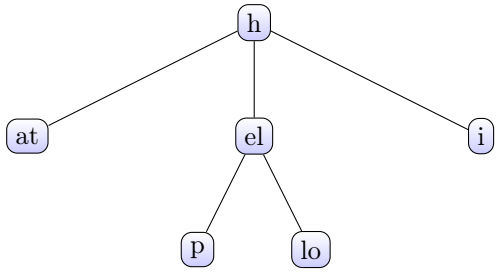

Figure 3: Sample radix trie, which stores the words `hat`, `help`, `hello`, and `hi`.

2.2 Hash functions

In the case of Bitcoin, a block of transactions is run through the SHA-256 hash algorithm to generate a hash, a string of numbers and letters which can be used to corroborate that a given set of data is the same as the original set of transactions. Hash functions such as the SHA-256 algorithm generate a fixed-length output often called hash, digest, or tag, that serves as a shortened reference to the original message which may be arbitrarily long, see Table 1. On the other hand, Ethereum uses the Keccak-256 hash fucntion, which was designed as a candidate for the SHA-3 Cryptographic Hash Function Competition held in 2007 by the National Institute of Science and Technology [3]. Note that hash functions are deterministic because they will always return the same output given a fixed input.

Cryptographic hash functions must fulfill a set of requirements. First of all, they must be computationally efficient and able to produce a hash from a message within reasonable time. Especially in the context of digital signing, cryptographic hash functions must be collision resistant, meaning it is hard to find a colliding pair of inputs: two distinct input messages whose corresponding digest is identical. Furthermore, their output should be well distributed and look random. It should hide information about the input and make it hard to infer any relevant information about the original message. These properties are desirable but they can be difficult to guarantee mathematically. Note that every cryptographic hash function is a hash function, but not every hash function is a cryptographic hash.

Table 1: Sample output hashes obtained using the SHA-256 algorithm using similar inputs. Note that the function is case sensitive with respect to the input and using similar input messages of different lengths generates fixed-length dissimilar outputs.

Input	Output hash
`Hello`	`185f8db32271fe25f561a6fc938b2e26` `4306ec304eda518007d1764826381969`
`hello`	`2cf24dba5fb0a30e26e83b2ac5b9e29e` `1b161e5c1fa7425e73043362938b9824`
`hello:)`	`29f5288fadcdc2ac50b3a1aeead0de75` `9facbb68d5b74d7e490135315760111c`

3 Merkle Trees

Merkle trees are named after Ralph Merkle [4]. Bitcoin and other cryptocurrencies use Merkle trees, also called hash trees, to encode and encrypt blockchain data efficiently and securely. Merkle trees are created by repeatedly calculating hashing pairs of nodes until there is only one hash left: the root hash or Merkle root, which is a summary value. They are constructed using a bottom-up approach in which each transaction is hashed, then each pair of transactions is concatenated and hashed together, and so on until there is one hash for the entire block. Figure 4 displays a simple binary Merkle tree.

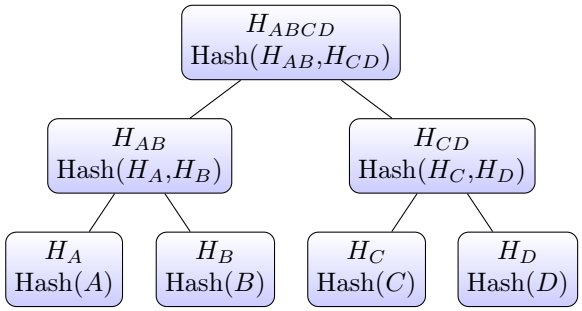

Figure 4: Simple binary hash tree, where H_{ABCD} is the Merkle root.

The Merkle tree allows us to efficiently proof that some piece of data was used to generate a root hash without needing to have access or store all the original pieces of information. The proof size (Merkle proof) scales logarithmically with the number of data blocks as opposed to simply concatenating and hashing all the data at once which would require storing large amounts of data. For example, in Figure 4 if we wanted to check if the piece of data A was used to generate the root hash, we would only need B (or H_B) and H_{CD}, because the latter summarises data C and D. If any of the data changes, the root hash will also change. This may be extended to also authenticate large databases of potentially unbounded size by all computers large and small, without compromising on scalability and avoiding centralization of the services.

In Bitcoin, a Merkle tree serves as a summary of all transactions in a block by producing a digital fingerprint of the entire set of transactions. The Merkle root of a given block is stored in the block header and it is combined with other information. All these information is then hashed again to produce the block's hash which is not actually included in the relevant block, but in the next block (as the previous block's hash). To get the leaves of the Merkle tree, Bitcoin uses the transaction hash, that is, the transaction ID (TXID) of every transaction included in the block. As a reminder, the TXID is a unique string of characters given to every transaction that is verified and added to the blockchain.

Satoshi described this as a simplified payment verification (SPV) in which instead of downloading every transaction and every block, a light client with limited computing and storage capabilities can only download the chain of block headers with the hash of the previous header, a timestamp, a mining difficulty value, a proof of work nonce, and a root hash for the Merkle tree containing the transactions

for that block, see Figure 5. If a light client wanted to verify the status of a transaction, the client could simply ask for a Merkle proof. The Merkle proof would be used to show that a given transaction is in one of the Merkle trees whose root is in a block header for the main chain. Note that the verification process in only reliable as long as honest nodes control the network, and the SPV method can be fooled by an attacker's fabricated transactions if the attacker is able to continue overpowering the network.

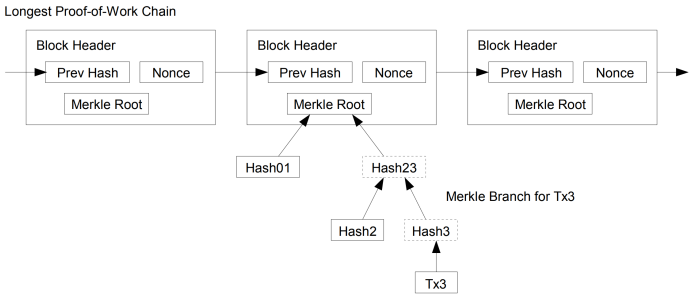

Figure 5: Original Simplified Payment Verification diagram from the Bitcoin White Paper [1] which proposed using Merkle Trees.

4 Merkle Patricia Tries

In the 2015 Ethereum Foundation blog post "Merkling in Ethereum" [5], Buterin mentioned that Bitcoin-style light clients have limitations, for example they cannot prove anything about the current state (i.e. how many Bitcoin the user holds or the account balance on Ethereum), and discussed a more complex implementation of the Merkle tree concept for the Ethereum blockchain.

In Ethereum, every block header contains three Merkle trees for three kinds of objects: transactions, receipts, and state, which aims at allowing light clients to make and get verifiable answers to many kinds of queries. The transactions tree can be used to check whether a transaction has been included in a block, the receipts tree can be used to retrieve all instances of a given event (e.g. amount of gas used or any event logs from the transactions), and the state tree to check the current balance of an account, whether an account actually exists, or to simulate running transactions on a given contract.

Regular Merkle trees are appropriate for transactions on Bitcoin because the list of transactions in a Bitcoin block will never change and will effectively be frozen. Hence, edit time is not important. However, the state in Ethereum is essentially a key-value mapping, where keys are account addresses and values are account declarations. This changes frequently since new accounts are being created, balances change, and account nonces update. The transactions trie is also a key-value mapping where the key is the transaction ID and the value is the transaction itself.

Ethereum requires a tree data structure that can quickly recalculate a tree root after an edit, insert or delete operation. The tree root must dependent on the data and not on the order in which updates are made. Note that this requirement is not satisfied by regular Merkle trees. Furthermore, the data structure must prevent Denial-of-Service attacks where malicious attackers may craft transactions to make the tree as deep as possible and hence, making updates really slow. To stop this the data structure must have bounded depth.

As previously mentioned in Section 2.1, Ethereum uses Merkle Patricia Tries, which solve inefficiency issues in radix tries. As a sidenote, Patricia stands for Practical Algorithm To Retrieve Information Coded In Alphanumeric. In Merkle Patricia Tries nodes are referenced by their hash and therefore, the root note can act as a cryptographic fingerprint for the data structure, as it was the case for Merkle trees in Section 3. Moreover, four node types are introduced to improve the efficiency of the data structure,

- empty nodes: blank nodes,
- leaf nodes: standard nodes with a key and a value,
- branch nodes: lists with a length of 17; the first 16 elements in the list correspond to the 16 hexadecimal characters that can be in a key, and the last element represents the value, given there is a key-value pair where the key ends at the branch node,
- extension nodes: key-value nodes whose value is the hash of another node.

Both leaf nodes and extension nodes have a key-value mapping format. Keys in Merkle Patricia Tries are encoded using a special Hex-Prefix (HP) encoding. A nibble, a single hexadecimal character, is appended to the beginning of the key to signify parity and terminator status. Parity refers to the node having even or odd length, and the terminator status denotes whether the node is an extension or leaf node. The lowest significant bit encodes parity, and the next lowest encodes the terminator status. Note that if the original key value was of even length, an extra zero nibble would be appended to achieve overall evenness. This ensures that the key can be properly represented in bytes.

In general when inserting into a Merkle Patricia Trie when stopping at a

- empty node: add a leaf node with the remaining path and replace the empty node with the hash of the new leaf node,
- leaf node: convert the node to an extension node and add both a new branch and leaf node,
- extension node: convert it to a new extension node with shorter path and create a new branch and leaf.

Note that we would not stop at branch nodes and therefore, they are not included in the list above.

5 Conclusion

In this work we have outlined some of the key concepts regarding trees in blockchain technology, mainly focusing on Merkle Trees and Merkle Patricia Tries. Note that this document may be modified and extended in the future to include further explanations.

References

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[2] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger eip-150 revision.

[3] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum*. O'Reilly Media, 2018.

[4] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[5] Vitalik Buterin. Merkling in ethereum. https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/, 2015.