Data Structure of Ethereum Blockchain

Biyang Fu
School of Computing
Clemson University
Clemson, US
biyangf@clemson.edu

***Abstract*—This article first briefly introduces some basic knowledge of blockchain, and then explains in detail the composition, key concepts and technologies of Ethereum. This document is the author's understanding of Ethereum based on some online surveys and research, so I can't guarantee that the article is 100% correct.**

Keywords—blockchain, Ethereum, composition, data structure

I. INTRODUCTION

As bitcoin begins to receive more attention from developers and other technical personnel, some new projects that use the bitcoin network to implement token transactions that are different from bitcoin logic, or other digital asset transactions besides tokens, have begun to appear. Since Bitcoin is not very flexible, most of these projects make some changes based on the Bitcoin system, add some new features and functions, and then independently run on unused nodes. In other words, each new project must be repeated and independently established a Bitcoin-like system. Can you design a more general system? Through the preparation of the application layer, let different digital assets run on a unified platform? Ethereum's inventor, Vitalik Buterin, is thinking about this problem.

Similar to Bitcoin, Ethereum is a decentralized blockchain platform. There are many nodes participating in this blockchain platform, they form a P2P network, these nodes are equal to each other, no one node has special permissions, and there is no coordination or scheduling by one or more nodes[1]. Each node in the Ethereum network can send out "transactions" or "accounting", that is, record and execute "transactions" issued on the network. These transactions will be packaged by nodes into "blocks", where each block contains the index of the previous block, so these blocks are connected in sequence to form a blockchain[2]. A consensus mechanism is adopted between these nodes to achieve data consistency, thereby forming a whole. Earlier versions of Ethereum used a proof-of-work (PoW) consensus mechanism like Bitcoin to ensure consistency.

There are many differences between Ethereum and Bitcoin. From the perspective of performance and characteristics, there are mainly the following differences.

- Ethereum has a faster "block generation" speed and a more advanced reward mechanism. At present, the average block generation time of Bitcoin is 10min, and the block generation interval of Ethereum is 12s, which means that Ethereum has a larger system throughput and a smaller transaction confirmation interval.
- Ethereum supports smart contracts, users can define their own digital assets and circulation logic, and can perform almost any calculation through the Ethereum virtual machine, while Bitcoin can only support the transfer of Bitcoin. This means that Ethereum can be used as a more general blockchain platform to support various decentralized applications (Dapp)[2].

II. DATA STRUCTURE

Ethereum is described as a transaction-driven state machine. After accepting some input in a certain state, it will be transferred to a new state. Specifically, in an Ethereum state, each account has a certain balance and storage information. When a set of transactions is received, the balance and storage information on the affected account will change. From the first genesis block, transactions are continuously received, and thus a series of new states can be entered. Ethereum packs transaction data and verification information into a block at regular intervals, and connects them in sequence to form a chain. The newer the block, the larger the block number (or block height)[3]. The block of Ethereum is composed of three parts: block header, transaction list and uncle block[4]. The block header contains block area code, block hash, parent block hash and other information. State Root, Transaction Root, and Recipet Root represent the hash of the state tree, transaction tree, and transaction tree, respectively. Except for the genesis block, each block has a parent block, which is connected into a blockchain with Parent Hash. As shown below:

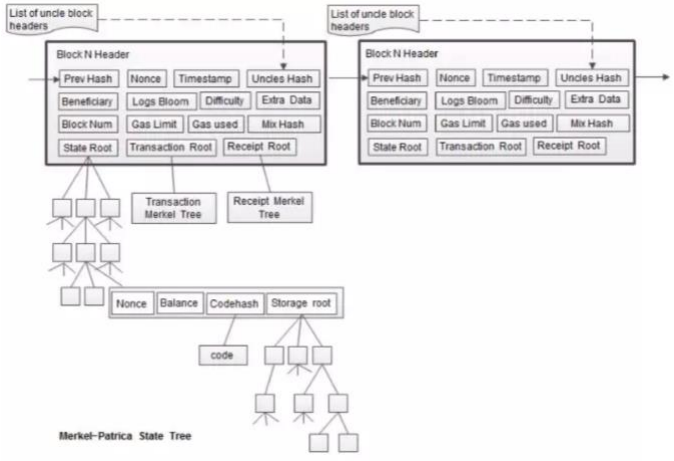

Fig. 1. Block data structure.

Ethereum uses Merkle Patircia Trie (MPT) as a data organization form to organize and manage important data such as user account status and transaction information. MPT is an encrypted and authenticated data structure that combines the advantages of Merkle tree and Trie tree (prefix tree). We first introduce these two data structures.

A. *Merkle Tree*

Merkle Tree, also called hash tree, is a concept of cryptography[5]. In the hash tree, the label of the leaf node is the hash value of the data block it is associated with, and the label of the non-leaf node is the hash value of the string of all its child nodes[5]. The advantage of the hash tree is that it can quickly and efficiently verify large amounts of data content. Assuming that there are n leaf nodes in a hash tree, if you want to verify whether one of the leaf nodes is correct, that is, whether the node data belongs to the source data set and the data itself is complete, the time complexity of the hash calculation required is O(log (n)). In contrast, hash lists require about O(n) hash calculation with time complexity, and the performance of the hash tree is undoubtedly excellent.

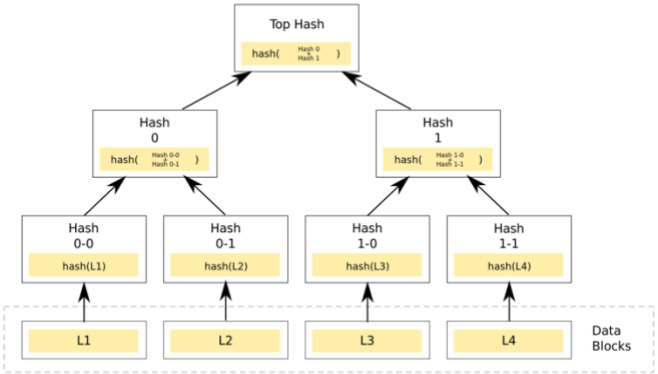

Fig. 2. Example of a binary hash tree.

The image above is from wiki-MerkleTree and shows a simple binary hash tree. The four valid data blocks L1-L4 are respectively associated with a leaf node. Hash0-0 and Hash0-1 are equal to the hash values of data blocks L1 and L2, respectively, and Hash0 is equal to the hash value of the new string spliced by Hash0-0 and Hash0-1, and so on, and the label of the root node top Hash is equal to the hash value of the new string spliced by both Hash0 and Hash1.

In the p2p network, before downloading, first obtain the Merkle Tree root of the file from a trusted source[6]. Once the tree root is obtained, the Merkle tree can be obtained from other untrusted sources[6]. Check the received Merkle Tree through the trusted roots[7]. If the Merkle Tree is damaged or fake, another Merkle Tree is obtained from other sources until a Merkle Tree that matches the root of the trusted tree is obtained[8].

B. *Trie Tree*

Trie tree, also known as prefix tree or dictionary tree[8]. Use the common prefix of the string to reduce the query time, minimize unnecessary string comparison, and the query efficiency is higher than the hash tree[9]. Typical applications are used for statistics, sorting and storing a large number of strings (not limited to strings), and are often used by search engine systems for text word frequency statistics[9]. The following picture is from wiki-Trie and shows a simple Trie structure.

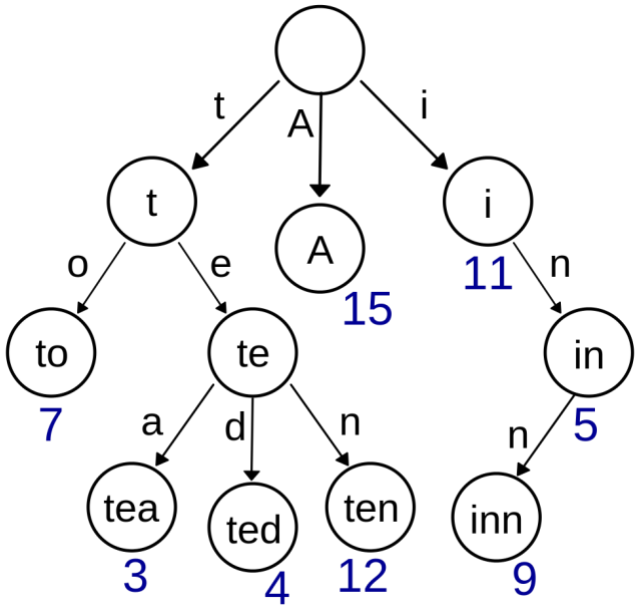

Fig. 3. Example of a Trie tree.

The Trie tree in the above figure represents the set of strings {“a”, “to”, “tea”, “ted”, “ten”, “i”, “in”, “inn”}. From the above figure, we can see the characteristics of the Trie tree[10]:

- The root node does not contain characters, and every child node except the root node contains a character.
- From the root node to a certain node, the characters passing on the path are connected to be the character string corresponding to the node.
- All the child nodes of each node contain different characters.

However, the above structure also shows a problem: it is highly uncontrollable. So there is the Patricia Trie (compressed prefix tree).

C. Patricia Trie

Patricia Trie, also known as radix tree or compact prefix tree, is a Trie with optimized space usage[11]. Unlike Trie, if there is a parent node and only one child node in Patricia Trie, then this parent node will be merged with its child nodes. This can shorten the unnecessary depth in Trie and greatly speed up the search node.

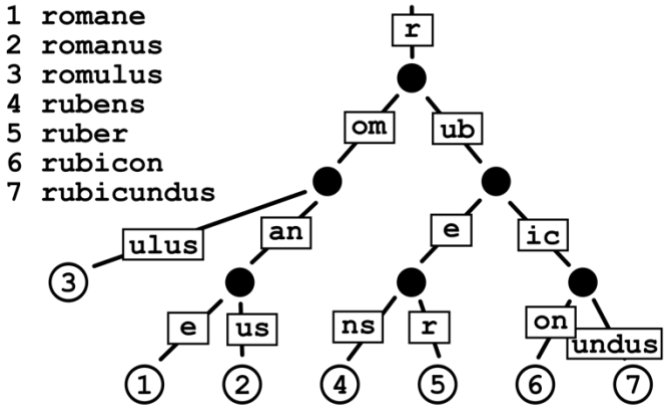

Fig. 4. Example of a Patricia Trie.

III. ETHEREUM TREE

Most blockchain projects, including Ethereum and Bitcoin, will use Merkle trees, and Ethereum has designed three Merkle trees for three kinds of objects, namely a state tree, a transaction tree, and a receipt tree[12]. These three trees can help the Ethereum client to do some simple queries, such as querying the balance of an account and whether a transaction is included in the block. Data such as blocks and transactions are ultimately stored in the LevelDB database. The LevelDB database is a key-value database. The key is generally related to the hash, and the value is the RLP encoding of the stored content. To summarize with a picture:

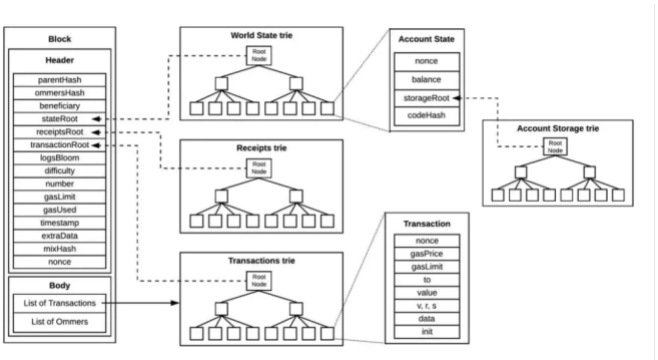

Fig. 5. Blocks, transactions, account status objects, and Ethereum's Merkel tree.

A. State Tree

All account information in Ethereum is reflected in the state and is saved by the state tree, which is continuously updated. For each account in the Ethereum network, a key-value pair is stored in the status tree, where the key is the address of the 160-bit account, and the value is the relevant information of the account, as shown in the following figure, including: nonce, balance, storageRoot, codeHash.

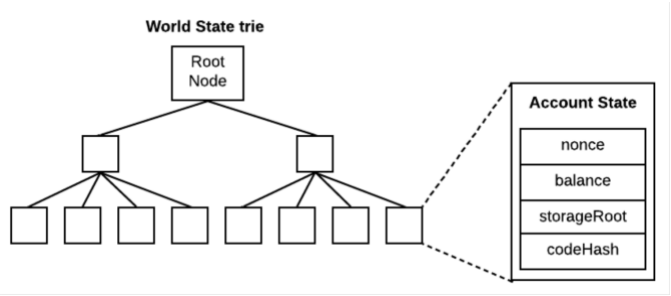

Fig. 6. World state trie and account state

If you want to know the balance of an account, or the current status of a smart contract, you need to obtain the specific status information of the account by querying the world status tree. The account storage tree is a structure for storing data associated with accounts. This item is only available for contract accounts. In Externally Owned Accounts (EOA), storageRoot is left blank and codeHash is the hash value of a string of empty strings.

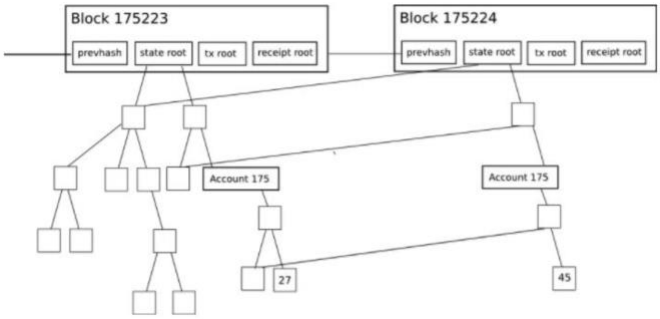

Fig. 7. Example of a Storage Trie

B. Transaction Tree

Whenever a block is published, all transactions contained in the block will be organized into a transaction tree. The tree is a Merkle Patricia Tree. The key value to be searched is the sequence number of the transaction at the time of publication (the order of transactions is determined by the node that posted the block). The transaction tree is used to prove that a transaction is in a block.

Each block has an independent transaction tree. The order of transactions in the block is mainly determined by the “miners”, and these data are unknown until this block is mined. However, "miners" generally sort transactions according to the Gas Price and nonce of the transaction (the number of transactions in the account address, which is used to ensure that each transaction can only be processed by a counter to avoid replay attacks). First, the transactions in the transaction list are divided into sending accounts, and the transactions in each account are sorted according to the nonce of these transactions. After the transaction sorting of each account is completed, the transaction with the highest price is selected by comparing the first transaction of each account. These are realized through a heap.

Each time a new block is dug, the transaction tree is updated. In the key-value pairs contained in the transaction tree, each key is the transaction number, and the value is the transaction content.

C. Receipt Tree

After each transaction is executed, a receipt will be formed to record the relevant information of the transaction, and these receipts will be organized into a receipt tree, the tree is a Merkle Patricia Tree, the key value is the serial number of the transaction at the time of publication (the order of transactions is determined by the node that issued the block). Information about the corresponding transaction in the receipt tree is post-transaction state, the cumulative gas used, the set of logs created through execution of the transaction, and the Bloom filter composed from information in those logs[8].

So, since there is already a transaction tree, why do you need the data structure of the receipt tree? Because Ethereum has smart contracts, and the execution process of smart contracts is more complicated, by adding a receipt tree, it is helpful for the system to quickly query the execution results.

In Ethereum, the receipt corresponding to each transaction will contain a Bloom filter, recording the transaction type, address and other information. There is also a total Bloom filter in the block header of the released block[13]. This Bloom filter is the union of the Bloom filters in all receipts[13].

If we need to find all the transactions related to a smart contract that occurred in the past period of time, we first need to see whether there is the type of transaction we want to find in the Bloom filter in the block header, and if so, then go to the Bloom filter of the receipt corresponding to each transaction in the block to find; if there is not, then there must be no transaction type in the block we want to find. Through this method, you can quickly exclude irrelevant receipts, thereby improving the search speed.

IV. K VALUE ENCODING

In Ethereum, there are three different encoding methods for the key value of the MPT tree to meet the different needs of different scenarios, which are introduced separately as a section here. The three encoding methods are: Raw encoding (native characters); Hex encoding (extended hexadecimal encoding); Hex-Prefix encoding (hexadecimal prefix encoding)[14].

A. Raw encoding

Raw encoding is the original key value without any changes. The key of this encoding method is the default encoding method for the interface provided by MPT. For example, for a data item whose key is "cat" and value is "dog", the Raw code is ['c', 'a', 't'], and the ASCII representation is [63, 61, 74].

B. Hex encoding

The conversion rule from Raw encoding to Hex encoding is:

- Divide each character of Raw encoding into two bytes according to the upper 4 bits and lower 4 bits;
- If the node corresponding to the key stores the actual data item content (that is, the node is a leaf node), add a character with an ASCII value of 16 as the termination identifier at the end;
- If the node corresponding to the key stores the hash index of another node (that is, the node is an extended node), no characters are added;

Hex coding is used to encode the MPT tree node key in memory. For example, for data items whose key is "cat" and value is "dog", the Hex coding is [6, 3, 6, 1, 7, 4, 10].

C. HP encoding

Another important concept in the MPT tree is a special hex-prefix (HP) encoding used to encode the key[15]. Because there are two [key, value] nodes (leaf nodes and extended nodes), they need to be distinguished. At this time, a special identifier (a bit is enough) is introduced to identify whether the key corresponds to a leaf or a hash of other nodes. If the identifier is 1, then the key corresponds to a leaf node, otherwise it is an expansion node.

Another point to note is that at a certain node, the length of the current path may be odd. One problem that will be faced at this time is that the path itself is based on 4 bits, which is a nibble, but it is always stored in bytes when storing. Assuming that there are currently two paths, '136' and '0136', there is no way to distinguish when storing, because when storing in bytes, it will always be converted into two bytes of 01 + 36. Therefore, in HP coding, there must also be an identifier for the path length parity.

Therefore, in the MPT tree, for each path (leaf node first remove the last 16), always add a nibble first, the lowest bit of this nibble indicates the length and parity of the node path, and the second lowest bit indicates the nature of the node. If the key is an even length, then because another four bits are added, another nibble with a value of 0 needs to be added to make the overall length even. The rules for HP coding are as follows:

- If the value of the last byte of the original key is 16 (that is, the node is a leaf node), remove the byte;
- Add a nibble before the key, where the lowest bit is used to encode the parity information of the original key length. If the key length is odd, the bit is 1; the lower 2 bits encode a special termination tag. If the node is a leaf node, the bit is 1;
- If the length of the original key is even, add a nibble with a value of 0x0 before the key;
- Compress the content of the original key, that is, divide the two characters into high 4 bits and low 4 bits, and store them in a byte (the inverse process of Hex expansion);

If the Hex code is [6, 3, 6, 1, 7, 4, 10], the HP code value is [20, 63, 61, 74]. The value of the added nibble and the path of the corresponding node's property table is as follows:

hex char	bits	node type partial	path length
0	0000	extension	even
1	0001	extension	odd
2	0010	terminating (leaf)	even
3	0011	terminating (leaf)	odd

Fig. 8. Nibble and the property table

According to the bits, the parity of the node type and the search path length can be determined. 0000, check the last two digits. The penultimate digit is 0, indicating an extension node; the last digit is 0, indicating that the path length is even. 0011, the value of the last two digits is 11, the second lowest digit is 1 indicating an expansion node, and the last digit is 1 indicating that the path length is an odd number. The remaining two situations are similar and can be analyzed.

Let's describe this process again with a specific example.

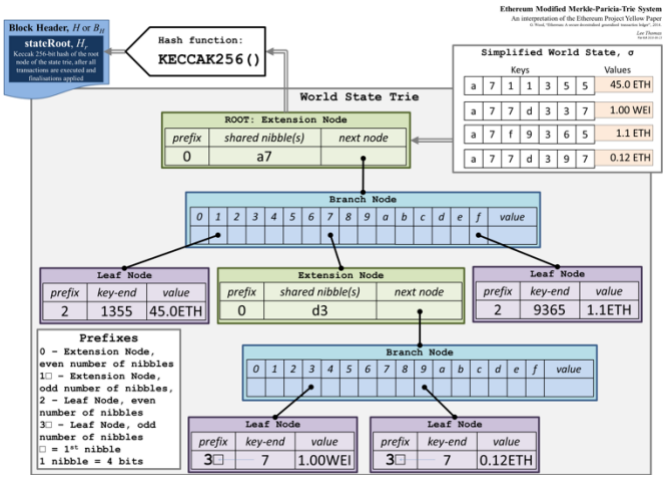

Fig. 9. The official example

In the figure above, there are four key-value pairs. The four keys have a common prefix a7, so the first node (also the root node) is an expansion node. Then there are three branches, 1, 7, and f. Therefore, the expansion node is followed by a branch node, and there are three branches in the branch node. There is only one node after branches 1 and 7, so it directly reaches the leaf node. There are two nodes after branch 7, and these two nodes have the common prefix'd3', so branch 7 is connected to an expansion node. After the expansion node there are two branches, 3 and 9, so keep up with another branch node. After the branch node are two leaf nodes.

Then analyze the prefix. In the extended node whose path is'a7', the path length is an even number and itself is an extended node. Therefore, the first nibble is 0000 and the length is even. Add a nibble 0000 and the prefix should be 00. The two leaf nodes in the second layer have even lengths, so the first nibble is 0010, and a nibble 0000 is added, so the final prefix should be 20. The extension nodes in the second layer are also of even length, so the prefix should be 00. In the leaf nodes of the last layer, the length is odd, so the added nibble is 0011, which is 3.

REFERENCES

[1] Wood, Gavin. "Ethereum yellow paper." Internet: https://github.com/ethereum/yellowpaper,[Oct. 30, 2018] (2014).

[2] Ethereum.org. "Ethereum." Internet: https://ethereum.org/zh/learn/,[Jun. 25, 2020] .

[3] Lucas Saldanha. " Merkle Tree and Ethereum Objects - Ethereum Yellow Paper Walkthrough (2/7)." Internet: https://www.lucassaldanha.com/ethereum-yellow-paper-walkthrough-2/,[Dec. 11, 2018] .

[4] Hui Ge, “[Deep knowledge] Ethereum block data structure and 4 Ethereum numbers,” Internet: https://learnblockchain.cn/2020/01/27/7c1fcd777d7b,[Jan. 27, 2020] .

[5] Blockchain base camp, “Understand the core data structure of Ethereum storage data: MPT,” Internet: https://www.tuoniaox.com/news/p-351583.html, [May 19, 2019] .

[6] Pony. Chen, “Ethereum data structure, storage, block header relationship and analysis,” Internet: https://blog.csdn.net/weixin_41545330/article/details/79394153, [Feb. 27, 2018] .

[7] Wood, Gavin, “Matering Ethereum,” Internet: https://github.com/inoutcode/ethereum_book, [Dec. 1, 2018].

[8] Lee, Timothy B. Ethereum, explained: why Bitcoin's stranger cousin is now worth $1 billion. Vox. 2016-05-24 [2016-05-25].

[9] A Next-Generation Cryptocurrency and Decentralized Application Platform. Bitcoin Magazine. [2016-05-06] .

[10] Lester Coleman. Coinbase Co-Founder: “Ethereum Is Ahead of Bitcoin in Many Ways”. Cryptocoins News. 2016-05-25 [2016-01-10].

[11] https://github.com/ethereum/wiki/wiki/White-Paper.

[12] Allison, Ian. How are banks actually going to use blockchains and smart contracts?. International Business Times. 2016-01-25 [2016-05-04].

[13] Laurie, B.; Langley, A.; Kasper, E. (June 2013). "Certificate Transparency". IETF: RFC6962. doi:10.17487/rfc6962.

[14] Franklin Mark Liang (1983). Word Hy-phen-a-tion By Com-put-er (PDF) (Doctor of Philosophy thesis). Stanford University. Archived (PDF) from the original on 2005-11-11. Retrieved 2010-03-28.

[15] Edward Fredkin (1960). "Trie Memory". Communications of the ACM. 3 (9): 490–499. doi:10.1145/367390.367400.