How the Ethereum trie actually works

A Blockchain and Cryptocurrency presentation

Why do we need a tree?

Need for a tree in an Ethereum block

- A block in the ethereum blockchain consists of:
  - header
  - list of transactions
  - list of uncle blocks.
- Included in the header is a transaction root hash, which is used to validate the list of transactions.
- While transactions are sent over the wire from peer to peer as a simple list, they must be assembled into a special data structure called a trie to compute the root hash.

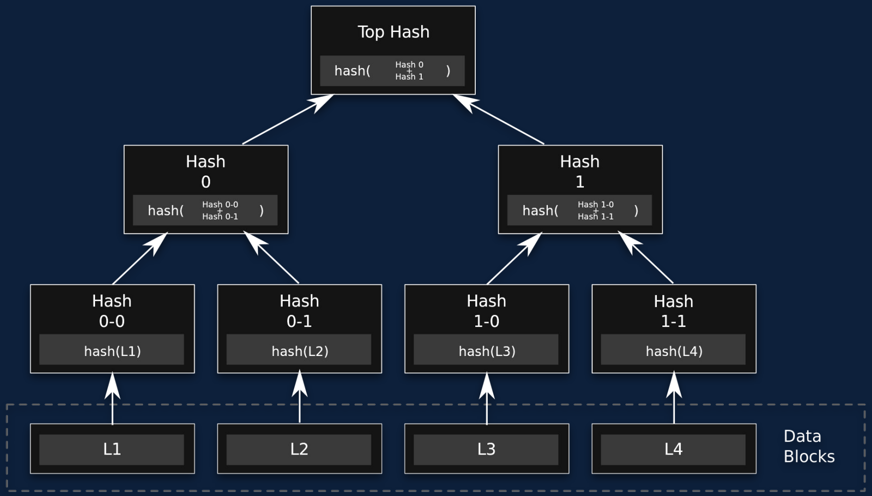
Need for a tree in an Ethereum block

This data structure is not needed except to verify blocks (and hence of course to mine them), and can technically be discarded once a block has been verified.

However, it is implied that the transaction lists are stored locally in a trie, and serialized to lists to send to clients requesting the blockchain.

What is a Merkle-Patricia trie?

Merkle Tree

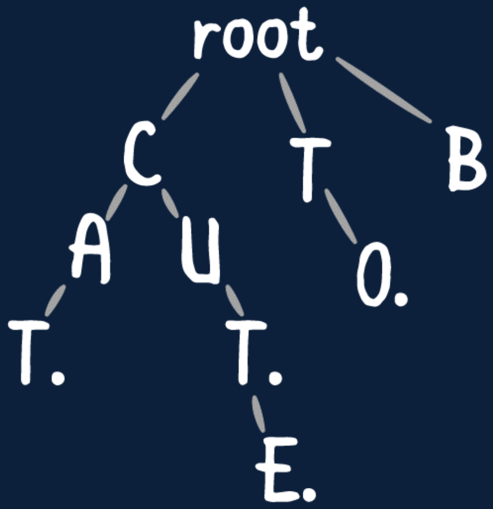


A **hash tree** or **Merkle tree** is a tree in which:

- every leaf node is labelled with the hash of a data block
- every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes

Hash trees can be used to verify any kind of data stored, handled and transferred in and between computers. They can help ensure that data blocks received from other peers in a peer-to-peer network are received undamaged and unaltered, and even to check that the other peers do not lie and send fake blocks.

Trie



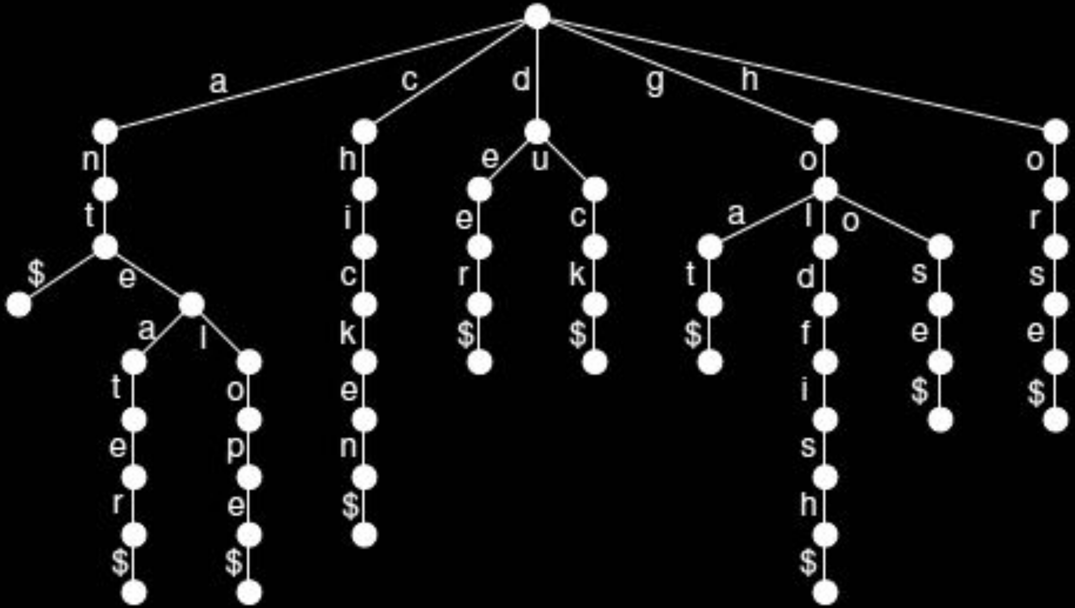
A **trie** is a tree-like data structure whose nodes store the letters of an alphabet, and that words and strings can be retrieved from the structure by traversing down a branch path of the tree.

Problem with a standard Trie

Tries make it easy to retrieve words, but they also take up a lot of memory and space with empty (null) pointers.

The redundancy of a standard trie comes from the fact that we are repeating ourselves by allocating space for nodes or edges that contain only one possible string or word. Another way to to think about is that we repeat ourselves by allocating a lot of space for something that only has one possible branch path.

Example of redundancy



Solution: A compressed trie

We can compress our trie so that we neither repeat ourselves, nor use up more space than is necessary. A compressed trie is one that has been compacted down to save space.

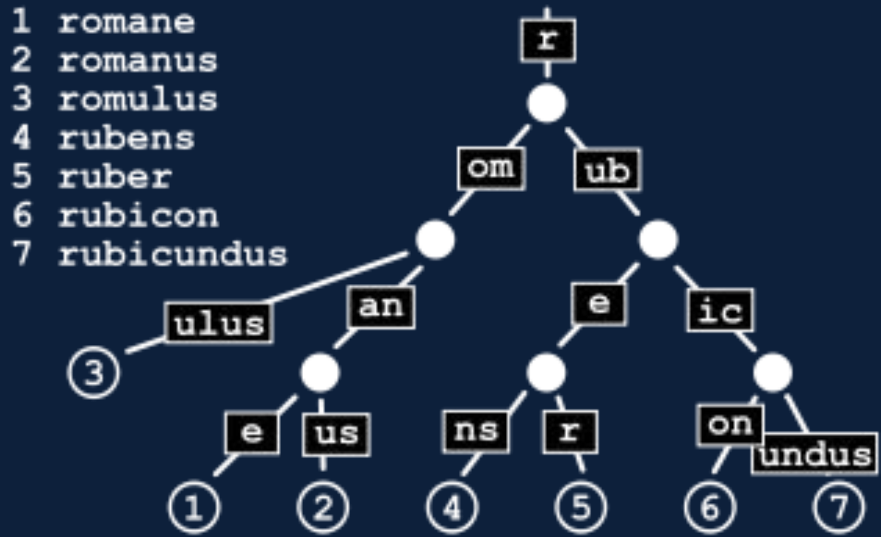
A compressed trie is a trie that has been compacted down to save space.
This is done by one crucial rule:

Each internal node (every parent node) has to have two or more children.

In other words, a node must have more than one branch path to leaf nodes in order for it to have children.
In order to compact the trie, each node that contained a single child is merged together.

Compressed tries are also known as **radix trees**, **radix tries**, or **compact prefix trees**.

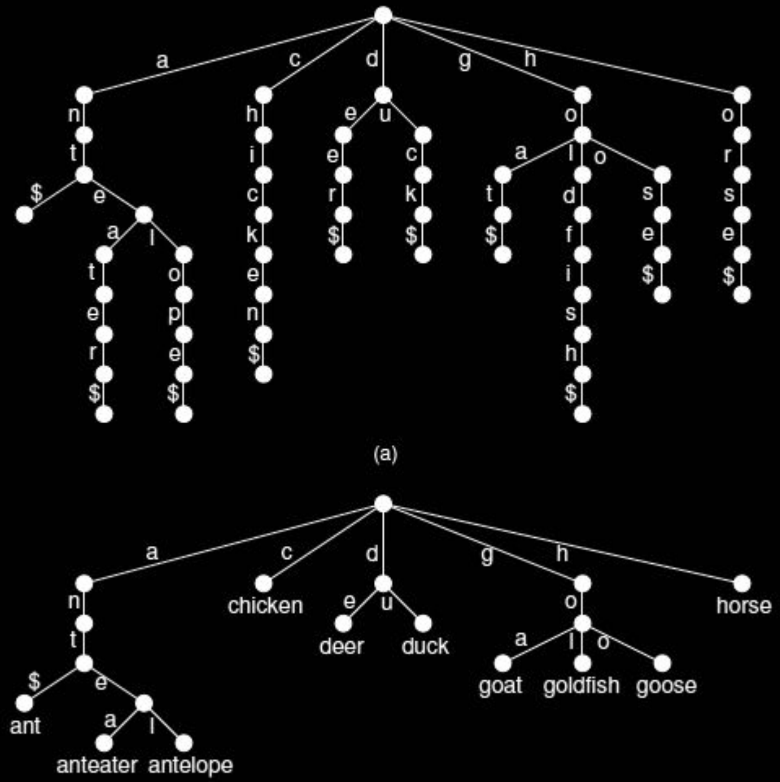
Radix Tree



A **radix tree** (also radix trie or compact prefix tree) is a data structure that represents a space-optimized trie in which each node that is the only child is merged with its parent.

Unlike regular trees (where whole keys are compared en masse from their beginning up to the point of inequality), the key at each node is compared chunk-of-bits by chunk-of-bits.

Space efficiency in a radix tree


(a)

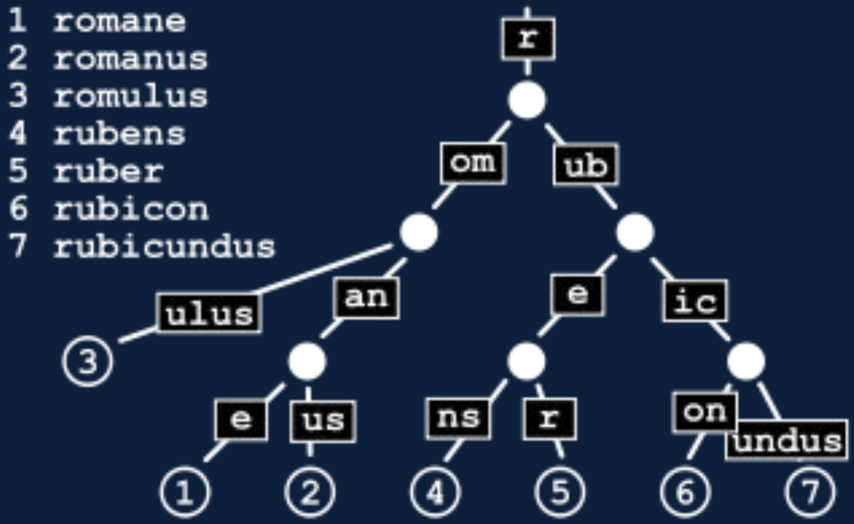
But what is a radix?

In order to understand the radix of a tree, we must understand how tries are read by our machines. In radix trees, keys are read in bits, or binary digits.
They are compared r bits at a time, where 2^r is the radix of the tree.

A trie's keys could be read and processed a byte at a time, half a byte at a time, or two bits at a time.

However, there is one particular type of radix tree that processes keys in a really interesting way, called a **PATRICIA tree**.

PATRICIA Tree



The **PATRICIA tree** was created in 1968 by Donald R. Morrison, who coined the acronym based on an algorithm he created for retrieving information efficiently from tries; PATRICIA stands for “Practical Algorithm To Retrieve Information Coded In Alphanumeric”.

PATRICIA trees are radix trees with radix equals 2.

But why PATRICIA?

The most important thing to remember about a PATRICIA tree is that its radix is 2. Since we know that the way that keys are compared happens r bits at a time, where 2^r is the radix of the tree, we can use this math to figure out how a PATRICIA tree reads a key.

Since the radix of a PATRICIA tree is 2, we know that r must be equal to 1, since $2^1 = 2$. Thus, a PATRICIA tree processes its keys one bit at a time.

Because a PATRICIA tree reads its keys in streams of bits, comparing 1 bit at a time, the entire tree is built up to represent binary digits.

PATRICIA Tree: A binary Radix Tree

A binary tree can only have two children, with the value of the left node being less than the value of the right node. So, in a binary radix tree, the right node is always used to represent binary digits (or bits) of 1, and the left node is used to represent bits of 0.

Because of this, each node in a PATRICIA tree has a 2-way branch, making this particular type of radix tree a binary radix tree.

PATRICIA Tree: An under-the-hood example

Let’s say that we want to turn our original set of keys, ["dog", "doge", "dogs"] into a PATRICIA tree representation. Since a PATRICIA tree reads keys one bit at a time, we’ll need to convert these strings down into binary so that we can look at them bit by bit.

```
dog:  01100100 01101111 01100111
doge: 01100100 01101111 01100111 01100101
dogs: 01100100 01101111 01100111 01110011
```

The keys "doge" and "dogs" are both substrings of "dog". The binary representation of these words is the exact same up until the 25th digit. Interestingly, even "doge" is a substring of "dogs"; the binary representation of both of these two words is the same up until the 28th digit.

How does Ethereum use the tree?

The Ethereum implementation: Modification 1

To make the tree cryptographically secure, each node is referenced by its hash, which in current implementations are used for look-up in a leveldb database.

With this scheme, the root node becomes a cryptographic fingerprint of the entire data structure (hence, Merkle).

The Ethereum implementation: Modification 2

A number of node ‘types’ are introduced to improve efficiency.

BLANK NODE	An empty node
LEAF NODE	Simple list of `[key, value]`
EXTENSION NODE	Simple `[key, value]` lists, but where value is a hash of some other node. The hash can be used to look-up that node in the database.
BRANCH NODE	Lists of length 17. The first 16 elements correspond to the 16 possible hex characters in a key, and the final element holds a value if there is a [key, value] pair where the key ends at the branch node.

How is the data encoded?

Key encoding techniques to understand

- RLP (recursive length prefix encoding), ethereum's home-rolled encoding system, is used to encode all entries in the trie.
- Hex-prefix (HP) encoding is used for keys in every `[key, value]` pair in the tree.

RLP (recursive length prefix encoding)

- For a **single byte** whose value is in the `[0x00, 0x7f]` range, that byte is its own RLP encoding.
- Otherwise, if a string is **0-55 bytes** long, the RLP encoding consists of a single byte with value **`0x80`** plus the length of the string followed by the string. The range of the first byte is thus `[0x80, 0xb7]`.
- If a string is **more than 55 bytes** long, the RLP encoding consists of a single byte with value **`0xb7`** plus the length of the length of the string in binary form, followed by the length of the string, followed by the string.
  - For example, a **length-1024** string would be encoded as **`\xb9\x04\x00`** followed by the string. The range of the first byte is thus `[0xb8, 0xbf]`.
- If the total payload of a list (i.e. the combined length of all its items) is **0-55** bytes long, the RLP encoding consists of a single byte with value **`0xc0`** plus the length of the list followed by the concatenation of the RLP encodings of the items. The range of the first byte is thus `[0xc0, 0xf7]`.
- If the total payload of a list is **more than 55 bytes long**, the RLP encoding consists of a single byte with value **`0xf7`** plus the length of the length of the payload in binary form, followed by the length of the payload, followed by the concatenation of the RLP encodings of the items. The range of the first byte is thus `[0xf8, 0xff]`.

Hex-prefix (HP) encoding

Since there are two kinds of `[key, value]` nodes (leaf and extension), a special ‘terminator’ flag is used to denote which type the key refers to.

TERMINATOR FLAG ON (LEAF)	: corresponding value is the value for that key.
TERMINATOR FLAG OFF (EXTENSION)	: value is a hash to be used to look-up the corresponding node in the `db`.

HP also encodes whether or not the key is of odd or even length.

The specifications, therefore, are:

- A nibble is appended to the key that encodes both the terminator status and parity. The lowest significant bit in the nibble encodes parity, while the next lowest encodes terminator status.
- If the key was in fact even, then we add another nibble, of value 0, to maintain overall evenness (so we can properly represent in bytes).

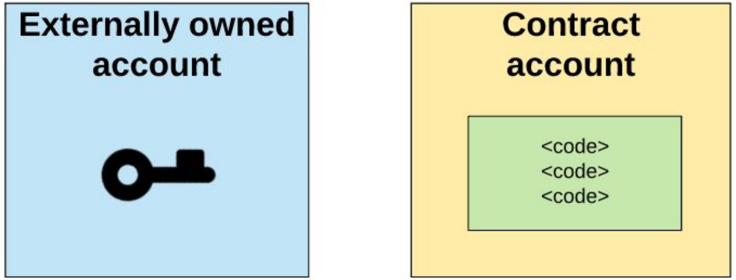
Where do all of these fit in the Ethereum architecture?

Ethereum Accounts

The **global “shared-state”** of Ethereum is comprised of many small objects (“accounts”) that are able to interact with one another through a message-passing framework. Each account has a state associated with it and a 20-byte address. An address in Ethereum is a 160-bit identifier that is used to identify any account.

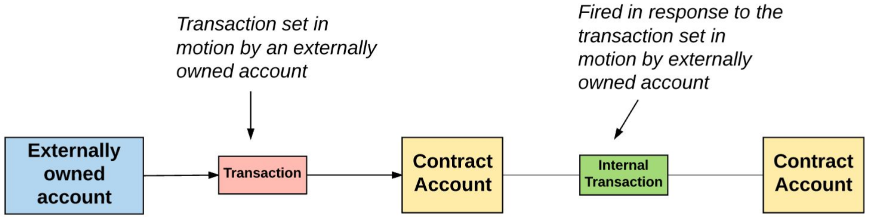
There are two types of accounts:

- **Externally owned accounts**, which are controlled by private keys and have no code associated with them.
- **Contract accounts**, which are controlled by their contract code and have code associated with them.



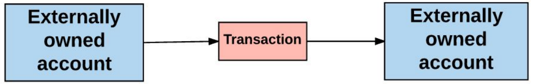
Externally owned accounts vs. Contract accounts

- An externally owned account can send messages to other externally owned accounts OR to other contract accounts by creating and signing a transaction using its private key.
  - A message between two externally owned accounts is simply a value transfer.
    A message from an externally owned account to a contract account activates the contract account’s code, allowing it to perform various actions (e.g. transfer tokens, write to internal storage, mint new tokens, perform some calculation, create new contracts, etc.).
- Unlike externally owned accounts, contract accounts can’t initiate new transactions on their own. Instead, contract accounts can only fire transactions in response to other transactions they have received.



Transaction set in motion by an externally owned account

Fired in response to the transaction set in motion by externally owned account



Account state

The account state consists of four components, which are present regardless of the type of account:

- `nonce`: If the account is an externally owned account, this number represents the number of transactions sent from the account's address. If the account is a contract account, the nonce is the number of contracts created by the account.

- `balance`: The number of Wei owned by this address. There are 1e+18 Wei per Ether.

- `storageRoot`: A hash of the root node of a Merkle Patricia tree. This tree encodes the hash of the storage contents of this account, and is empty by default.

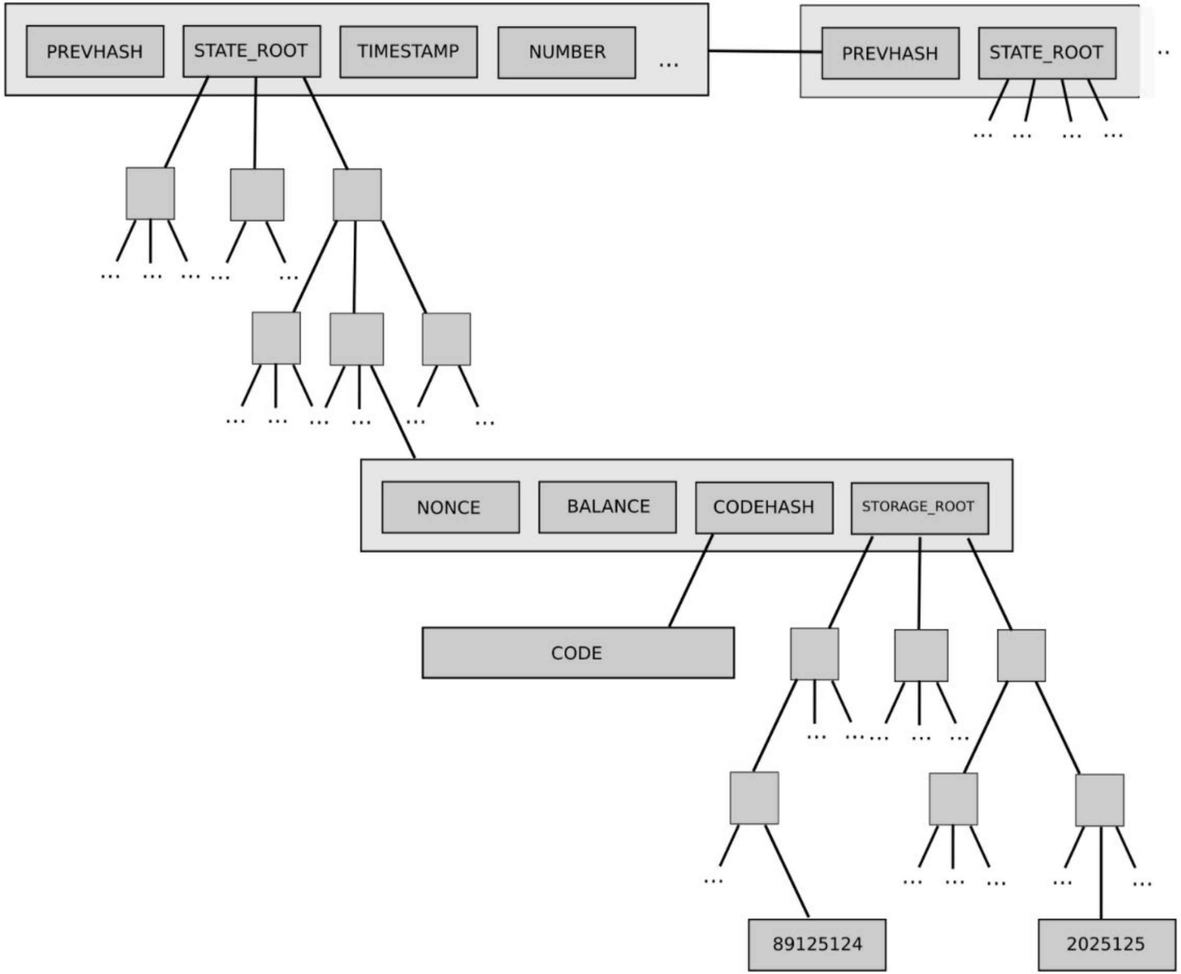
- `codeHash`: The hash of the EVM (Ethereum Virtual Machine—more on this later) code of this account. For contract accounts, this is the code that gets hashed and stored as the codeHash. For externally owned accounts, the codeHash field is the hash of the empty string.

World state

We know that Ethereum's global state consists of a mapping between account addresses and the account states. This mapping is stored in a data structure known as a **Merkle Patricia trie**.

This tree is required to have a key for every value stored inside it. Beginning from the root node of the tree, the key should tell you which child node to follow to get to the corresponding value, which is stored in the leaf nodes.

In Ethereum's case, the key/value mapping for the state tree is between addresses and their associated accounts, including the `balance`, `nonce`, `codeHash`, and `storageRoot` for each account (where the `storageRoot` is itself a tree).

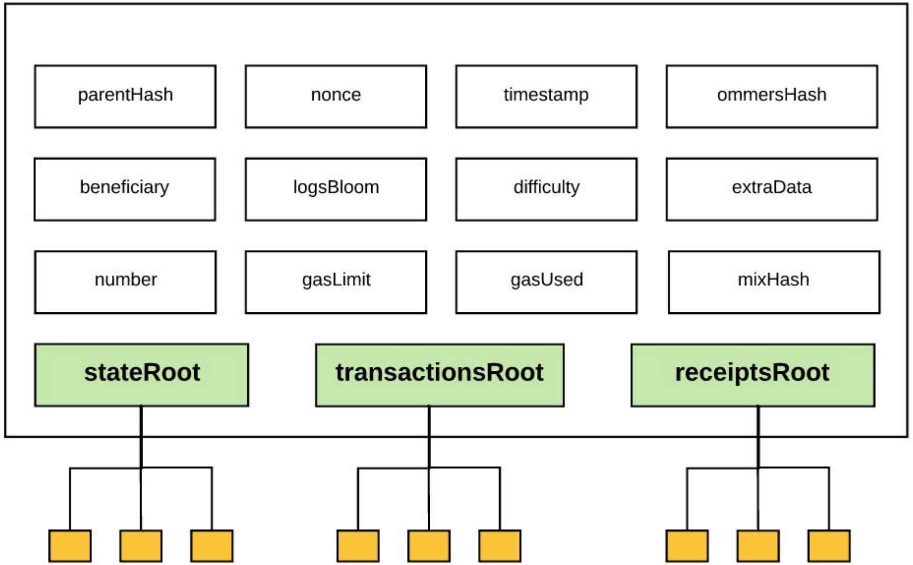

PREVHASH
STATE_ROOT
TIMESTAMP
NUMBER
...
PREVHASH
STATE_ROOT
..
NONCE
BALANCE
CODEHASH
STORAGE_ROOT
CODE
89125124
2025125

Ethereum Block Header

This same trie structure is used also to store transactions and receipts. More specifically, every block has a "**header**" which stores the hash of the root node of three different Merkle trie structures, including:

- State trie
- Transactions trie
- Receipts trie

Block header



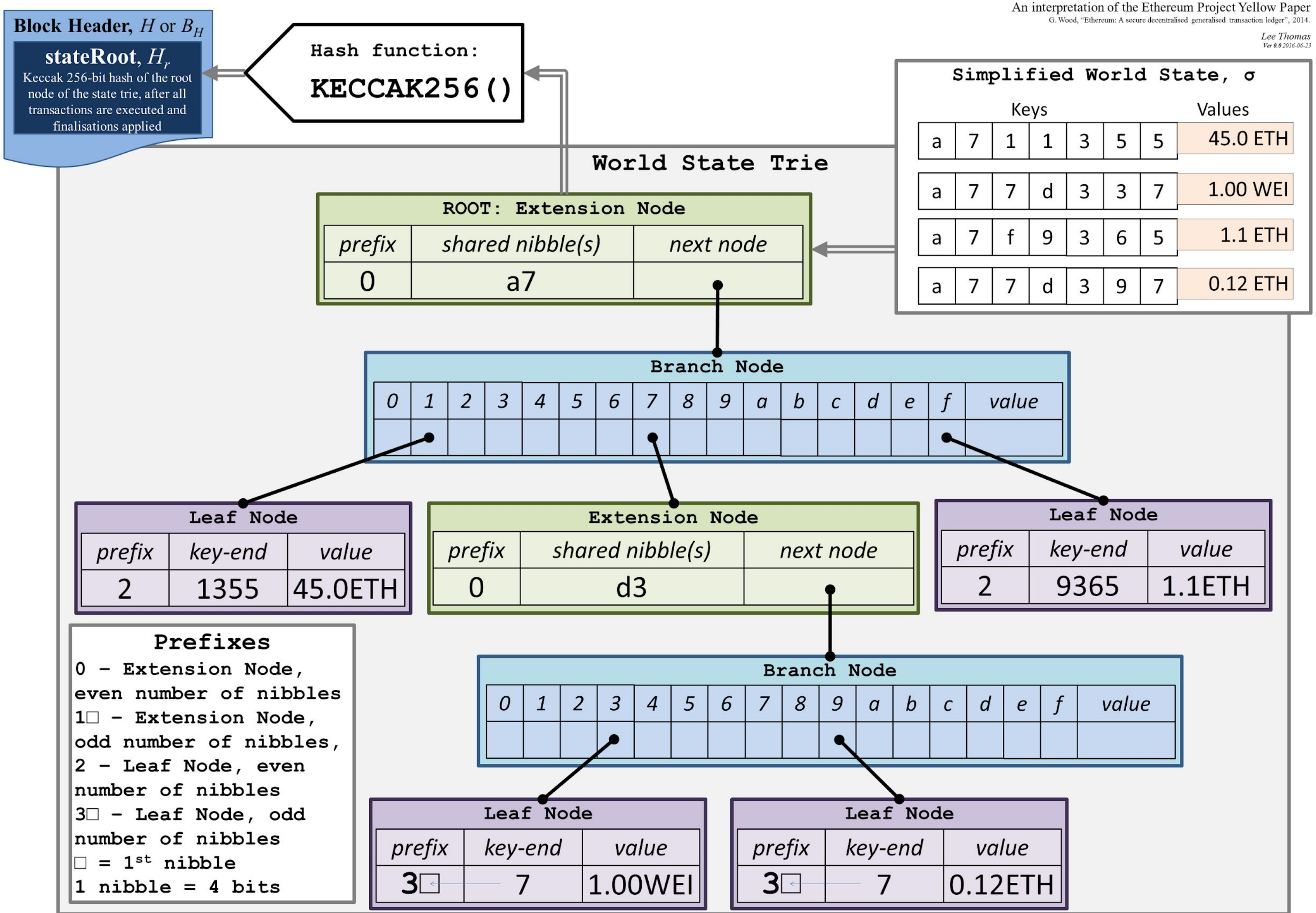
Ethereum Modified Merkle-Paricia-Trie System

An interpretation of the Ethereum Project Yellow Paper

G. Wood, "Ethereum: A secure decentralised generalised transaction ledger", 2014.

Lee Thomas

Ver 0.0 2016-06-23



Block Header, H or B_H

stateRoot, H_r
Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied

Hash function:

`KECCAK256()`

Simplified World State, σ

Keys							Values
a	7	1	1	3	5	5	45.0 ETH
a	7	7	d	3	3	7	1.00 WEI
a	7	f	9	3	6	5	1.1 ETH
a	7	7	d	3	9	7	0.12 ETH

World State Trie

ROOT: Extension Node

prefix	*shared nibble(s)*	*next node*
0	a7	●

Branch Node

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*a*	*b*	*c*	*d*	*e*	*f*	*value*
	●						●								●	

Leaf Node

prefix	*key-end*	*value*
2	1355	45.0ETH

Extension Node

prefix	*shared nibble(s)*	*next node*
0	d3	●

Leaf Node

prefix	*key-end*	*value*
2	9365	1.1ETH

Branch Node

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*a*	*b*	*c*	*d*	*e*	*f*	*value*
			●						●							

Leaf Node

prefix	*key-end*	*value*
3□	7	1.00WEI

Leaf Node

prefix	*key-end*	*value*
3□	7	0.12ETH

Prefixes

0 – Extension Node, even number of nibbles
1□ – Extension Node, odd number of nibbles,
2 – Leaf Node, even number of nibbles
3□ – Leaf Node, odd number of nibbles
□ = 1st nibble
1 nibble = 4 bits

Code implementation

```
git clone git@github.com:coderIlluminatus/Blockchain
```

`ex1.py:` Initialize the tree with a blank root and add a single entry

```
state = trie.Trie('triedb', trie.BLANK_ROOT)
state.update('\x01\x01\x02', rlp.encode(['hello']))
print state.root_hash.encode('hex')


k, v = state.root_node
print 'root node:', [k, v]
print 'hp encoded key, in hex', k.encode('hex')
```

Code implementation

The output of `ex1.py` is

```
root hash 15da97c42b7ed2e1c0c8dab6a6d7e3d9dc0a75580bbc4f1f29c33996d1415dcc
root node: [' \x01\x01\x02', '\xc6\x85hello']
hp encoded key, in hex: 20010102
```

Note the final 6 nibbles are the key we used, `010102`, while the first two give us the HP encoding.

The first nibble tells us that this is a terminator node (since it would be `10` in binary, so the second least significant bit is on), and since the key was even length (least significant bit is `0`), we add a second `0` nibble.

Code implementation

`ex2.py`: Add some more entries to the trie by starting off with the previous hash

```
state = trie.Trie('triedb',
'15da97c42b7ed2e1c0c8dab6a6d7e3d9dc0a75580bbc4f1f29c33996d1415dcc'.decode('hex'))
print state.root_node

state.update('\x01\x01\x02', rlp.encode(['hellothere']))
print state.root_hash.encode('hex')
print state.root_node
```

The output of `ex2.py` is

```
05e13d8be09601998499c89846ec5f3101a1ca09373a5f0b74021261af85d396
[‘ \x01\x01\x02’, ‘\xcb\x8ahellothere’]
```

Code implementation

`ex2b.py:` Add some more entries to the trie by starting off with the previous hash

```
state.update('\x01\x01\x03', rlp.encode(['hellothere']))
print 'root hash:', state.root_hash.encode('hex')

k, v = state.root_node
print 'root node:', [k, v]
print 'hp encoded key, in hex:', k.encode('hex')

print state._get_node_type(state.root_node) == trie.NODE_TYPE_EXTENSION
common_prefix_key, node_hash = state.root_node
print state._decode_to_node(node_hash)
print state._get_node_type(state._decode_to_node(node_hash)) == trie.NODE_TYPE_BRANCH
```

Code implementation

The output of `ex2b.py` is

```
root hash: b5e187f15f1a250e51a78561e29ccfc0a7f48e06d19ce02f98dd61159e81f71d

root node: ['\x10\x10\x10',
'"\x01\xab\x83u\x15o\'\xf7T-h\xde\x94K/\xba\xa3[\x83l\x94\xe7\xb3\x8a\xcf\n\nt\xbb\xef\xd9']

hp encoded key, in hex: 101010

True

['', '', [' ', '\xc6\x85hello'], [' ', '\xcb\x8ahellothere'], '', '', '', '', '', '', '',
'', '', '', '', '', '']

True
```

Code implementation

What we have here is a branch node, a list with 17 entries.

Note the difference in our original keys: they both start with [0,1,0,1,0], and one ends in 2 while the other ends in 3. So, when we add the new entry (key ending in 3), the node that previously held the key ending in 2 is replaced with a branch node whose key is the HP encoded common prefix of the two keys.

The branch node is stored as a [key, value] extension node, where key is the HP encoded common prefix and value is the hash of the node, which can be used to look-up the branch node that it points to.

The entry at index 2 of this branch node is the original node with key ending in 2 (‘hello’), while the entry at index 3 is the new node (‘hellothere’).

Since both keys are only one nibble longer than the key for the branch node itself, the final nibble is encoded implicitly by the position of the nodes in the branch node. And since that exhausts all the characters in the keys, these nodes are stored with empty keys in the branch node.

Light Nodes

The ability to store all this information efficiently in Merkle tries is incredibly useful in Ethereum for what we call “light clients” or “light nodes.” Broadly speaking, there are two types of nodes: **full nodes** and **light nodes**.

A **full archive node** synchronizes the blockchain by downloading the full chain, from the genesis block to the current head block, executing all of the transactions contained within. Typically, miners store the full archive node, because they are required to do so for the mining process.

But unless a node needs to execute every transaction or easily query historical data, there’s really no need to store the entire chain. This is where the concept of a light node comes in.

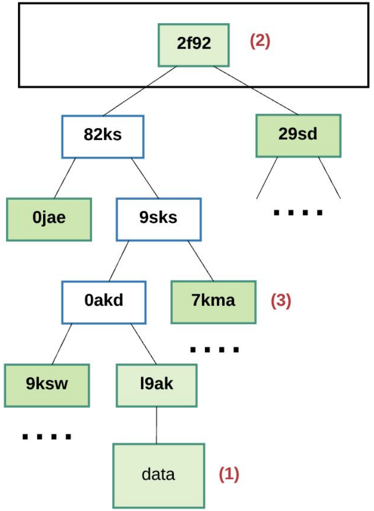
Instead of downloading and storing the full chain and executing all of the transactions, **light nodes** download only the chain of headers, from the genesis block to the current head, without executing any transactions or retrieving any associated state. Because light nodes have access to block headers, which contain hashes of three tries, they can still easily generate and receive verifiable answers about transactions, events, balances, etc.

Merkle Proof

Any node that wants to verify a piece of data can use something called a "**Merkle proof**" to do so. A Merkle proof consists of:

- A chunk of data to be verified and its hash
- The root hash of the tree
- The "branch" (all of the partner hashes going up along the path from the chunk to the root)

Anyone reading the proof can verify that the hashing for that branch is consistent all the way up the tree, and therefore that the given chunk is actually at that position in the tree.



Conclusion

The benefit of using a **Merkle Patricia trie** is that the root node of this structure is cryptographically dependent on the data stored in the tree, and so the hash of the root node can be used as a secure identity for this data.

Since the block header includes the root hash of the state, transactions, and receipts trees, any node can validate a small part of state of Ethereum without needing to store the entire state, which can be potentially unbounded in size.

Useful Resources

Diving into Ethereum’s world state

`https://medium.com/cybermiles/diving-into-ethereums-world-state-c893102030ed`

Data Structure in Ethereum

`https://medium.com/@phansnt`

Compressing Radix Trees Without (Too Many) Tears

`https://medium.com/basecs/compressing-radix-trees-without-too-many-tears-a2e658adb9a0`

How does Ethereum work, anyway?

`https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369`

Presented by

Sayantan Chatterjee IIT 2015 511