Thanks to @sinamahmoodi for his input, @poemm and @carver for pointing out typos.

A method for the conversion of hexary tries into binary tries has been proposed in [1]. This proposal focuses on how the trie is represented, independently of the conversion method.

It is made up of three parts: the first part is a simpler write up, the second part illustrates the structure with an example, and the last part presents the formalization of the structure, using the notation from the yellow paper[2].

Overview

The proposed structure has only one type of node, which merges the functionalities of branch, extension and leaf nodes.

Field	Type	Description	Leaf node content	Internal node content
Prefix	RLP byte list	Prefix common to all subnodes of this node	bin prefix	bin prefix
Left	Hash / child RLP byte list	Pointer to the left child	empty RLP list	RLP of the left child, or its hash if it takes more than 32 bytes
Right	Hash / child RLP byte list	Pointer to the right child	empty RLP list	RLP of the right child, or its hash if it takes more than 32 bytes
Value	Bytes	Empty list if internal node, value bytes if leaf node.	value bytes	empty RLP list

In hexary tries, key encoding required a hex prefix

to specify if the key length was odd or even, as well as specifying if the value was a leaf.

Likewise, key segments are encoded using a bin(ary) prefix

. It packs all bits into the minimal number of bytes, and maintains a header to specifies how many bits of the last byte are in use. Bits are stored using the big endian notation. Unlike its hexary counterpart, the bin prefix doesn't encode if the node is a leaf: this information is encoded by either child being empty.

Implementation

A first, naive implementation of this has been implemented in Geth[3]. It is based on the new snapshot feature introduced in Geth, in which all (key, values) are directly stored in the database.

Converting the trie from hexary to binary takes roughly 45 minutes on a mainnet machine, and the resulting dataset size is 20GB.

Example

Let’s assume a trie with the following three (key, value) pairs:

Binary key

Hexadecimal key

Hex value

1100 1010 1111 1110

cafe

00

1100 1010 1111 1111

caff

01

1011 1110 1110 1111

beef

02

First key

Upon insertion of the first key, the trie would look like:

There are 2 bytes in key 0xcafe

, and all the nodes in the trie (i.e. only one) are prefixed by the same bit string (1100 1010 1111 1110

, that is, the bits of 0xcafe

). So the prefix of this single node is cafe

.

Since all 8 bits are being used in the last byte, the value of the first byte in \mathtt{BP}

(0xcafe

) is 8 \mod 8 = 0

.

This node has no children so both left and right child are empty values.

Second key

Inserting the second value will cause a fork at the last bit:

The last bit in the root is not used, so the first byte of the bin prefix is now 7, although the prefix seems not to have changed from its initial value (the last bit is to be set to 0, which is the value it already had).

Two sub-nodes had to be inserted, the left one to contain the initial value 0x00

, and the second one to contain the new value 0x01

. Neither of these nodes have a prefix since the fork occurs at the last bit, but their bin prefix is therefore simply 0x00

, since all the bits in the non-existent last byte are being used.

Third key

Inserting the third value will cause a fork at the second bit:

The root is now only 1 bit long, which is why the first byte of its bin prefix reflects is 1. That bit is 1

, and it's packed to the left so the byte key is 0b10000000 == 0x80

.

The new leaf has 6 bits since the top bit is stored in the root's bin prefix and the second bit 0

is covered by going left in the trie. The byte value are the remaining 0b11111011101111

part of the key, which have been packed to the left, so 0b1111101110111100 == 0xfbbc

.

An intermediate node is also created on the c

branch, and contains the former key prefix 0xcafe = 0b1100101011111110

whose first two bits have been removed, and the remaining packed to the left. So that's 0b0010101111111000 == 0x2bf8

.

Formalization

Notation

The notation here is based on that of the yellow paper.

In the formulas of this text, the product symbol $\prod$

is redefined to mean the element concatenation, formally:

$$\forall a=(a_0,a1,...,a_{n-1}), \forall (i,j) : 0 \le i \le j \lt n, \prod_{i}^{j}a_i \equiv (a_i, a_{i+1}, ..., a_{j-1}, a_j)$$

It is also assumed that bit strings are indefinitely extensible with 0s to the right. Formally:

$$\forall x : x \in {0,1}^n, \forall i : i \in \mathbb{N}, i \ge \lVert x \rVert \implies x[i] = 0$$

Binary prefix

A trie node encode a segment of its key with the help of a Binary Prefix

, which is a function that is used to pack the bits of a key into a sequence of bytes.

Formally, the binary prefix function $\mathtt{BP}$

is defined as:

$$\mathtt{BP}(x) : x \in {0,1}^n \equiv \Big(\lVert x \rVert \mod 8,\prod_{i=0}^{\lceil \lVert x \rVert \div 8 \rceil}\sum_{k=8i}^{8i+7}{\big(x[k]\times2^{k-8i}}\big)\Big)$$

which means that for a sequence of bits $b_0, b_1, ..., b_n$

, $\mathtt{BP}$

will return a sequence of bytes in which the first byte will contain the number of bits used in the last byte, and then a sequence of bytes in which all b_i

bits are packed.

Binary Trie

Equations (190) and (191) in the yellow paper are updated to reflect that keys are viewed as a series of bits

(as opposed to nibbles

), and the notation is still big-endian:

b(\mathfrak{I}) = { (\mathbf{k}*0' \in {0, 1}^n, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}_1' \in {0, 1}^n, \mathbf{v}_1 \in \mathbb{B}), ... } \ \forall n : \quad \forall i < 8\lVert\mathbf{k}*{n}\rVert: \quad \mathbf{k}*n'[i] \equiv \mathbf{k}[i \div 8]*(i \mod 8)}

The \mathtt{TRIE}

function is still defined as:

\texttt{TRIE}(\mathfrak{I}) \equiv \texttt{KEC}(c(\mathfrak{I}, 0))

and accordingly the trie's node cap function is still defined as:

n(\mathfrak{I},i)\equiv\begin{cases} () & \text{if} \quad \mathfrak{I}=\varnothing \ c(\mathfrak{I},i) & \text{if} \quad \lVert c(\mathfrak{I}, i) \rVert < 32 \ \texttt{KEC}(c(\mathfrak{I}, i)) & \text{otherwise} \end{cases}

Only one type of node exists, which contains 4 items:

- A key prefix

, which is the part of the key common between all sub-nodes of this node;

- A left

(respectively, right

) pointer

that is pointing to the root of the left (respectively, right) subtree; and

- a value

field that contains the arbitrarily long value stored at this node.

A leaf

is a node with an empty left and right child. Note that a node either has two non-empty children, or both of them are empty.

Consequently the structural composition function (194) is updated to:

c(\mathfrak{I}, i) \equiv \texttt{RLP}(\texttt{BP}(I_0[i..(j-1)]), u(0), u(1), v) \quad \text{where} \begin{array}[t]{rcl} j & = & \max { x : \exists \mathbf{l}: \lVert \mathbf{l} \rVert = x \wedge \forall I \in \mathfrak{I}: I_0[0 .. (x - 1)] = \mathbf{l} } \ u(k) & \equiv & n({I : \mathfrak{I} \wedge I_0[i]=k}, j+1) \ v & = & \begin{cases} I_1 & \text{if} \quad \exists I : I \in \mathfrak{I} \wedge \lVert I_0 \rVert = i \ () & \text{otherwise} \end{cases} \end{array}

References

1. hex -> bin conversion with the overlay tree method
2. The ethereum yellow paper
3. Binary trie test implementation