# Fast Indexing Engine for Data Identified by a Hashed ID and Stored in an Immutable File

Cyril Dever
Edgewhere

November 26, 2018

**Abstract**

*We define here an algorithm for indexing the system based on identifiers that are hashed values which is at the same time very powerful to the writing and the reading. We call the Treee™.*

## I. Introduction

The challenge was to set up a powerful yet safe search engine to use when the data is some linked list of items that could be themselves connected to each other in subchains, indexed through their identifiers that are only made of hashed values (like SHA-256 string representations), and all stored in an immutable file.

Its best application is for a blockchain file where an *item* is a transaction embedding a smart contract, and each subchain of items the subsequent uses and/or modifications of this smart contract.

This present document describes such indexing engine.

**Definition 1** (Item)**.** An *item* is the actual object recorded in the immutable file subject to indexing.

As previously stated, it is initially meant to be a transaction or a block in a blockchain file.

**Definition 2** (Leaf)**.** A leaf $\lambda$ embeds the information helping to retrieve one or more possibly linked *items*.

## II. Formal Description

### 1. Acyclic graph

Treee™ is an algorithm for indexing *items* recorded in an immutable file based on their identifiers that are hashed values.

It is constructed as an acyclic graph (a tree $T$), each node containing either a node address (its sons) or a set of *leaf*.

**Definition 3** (Hashed Identifier)**.** We call $\iota$ a hashed identifier (or hashed value) the hexadecimal string representation of the result of a data $d$ passed through a hashing function $\mathfrak{h}()$ such as:

$$\iota := \mathfrak{h}(d) \tag{1}$$

The passed data $d$ could be anything but it must be unique if it were to be used as identifier per se.

In blockchains we operate, this data $d$ is usually the *item* itself, ie. a transaction or a block.

The hashing function could use any cryptographic hashing algorithm as long as it is set beforehand and once and for all in the targeted system [1]. The number $\mathfrak{N}$ represents the number of bits of the returned hash, eg. 256 for `SHA-256`.

The number of sons of a node is deterministic and depends on the depth of the tree.

Let $p_k$ be the number of sons of a node $N_k$ at depth $k$.

The goal is to create a balanced tree whose width is adaptive to decrease depth and optimize performance.

We are looking to index numbers, in this case the numerical value of the *item*'s unique identifiers $\iota_i (\forall i \in \mathbb{N})$.

[1] We currently use the `SHA-256` algorithm because of its wide adoption in both end-user and back-end environments

*Recall.* An identifier is at its core a hashed value, that is its digest is fundamentally a byte array that could represent any positive integer.

## 2. Index

We now explain the course of the index.

Let $\iota_i^b$ be the value of the hashed identifier for the *item* $i$ in binary form, eg.

$$\iota_i^b := \texttt{"a1"} \mapsto \texttt{10100001}$$

indicating its position in the tree $T$.

At each step $j := [0..n) \mid n \leq \mathfrak{N}$, we would pass to child 0 if the $j$-th bit of $\iota_i^b$ equals 0; otherwise, we would pass to child 1.

Let $R_j^{\iota_i}$ be the value of this representation of $\iota_i^b$ at step $j < k$.

For a full tree $T$, we build a representation of this number at each step and traverse the tree the same way.

At the step $j$ of depth $k$, we pass to child 0 if $R_j^{\iota_i}$ equals 0, we pass to child 1 if $R_j^{\iota_i}$ equals $1, \ldots,$ we pass to child $p_{k-1}$ if $R_j^{\iota_i}$ equals $p_{k-1}$. We stop when the node is an empty leaf $\lambda_j$.

**Definition 4** (Representative)**.** To construct $R_j^{\iota_i}$, this representative at step $j$, we will successively take the modulo of prime numbers, each step $j$ using the $j$-th prime number in the ordered sequence of all prime numbers $\mathcal{P} := [1, 2, 3, 5, 7, 11, 13, \ldots]$.

This construction ensures that $R_j^{\iota_i}$ be unique.

*Proof.* According to the Chinese remainder theorem[1], each number has a unique representative that could be written as the continuation of these modulos.

Indeed, a number $n$ can be written in the following form:

$$n \equiv n \bmod P_i$$

where $P_i$ is the $i$-th modulo in $\mathcal{P}$.

Modulos are calculated in $O(1)$ for fixed-sized integers. Since the multiplication is faster than the division (necessary for the calculation of the modulo), one may use multiplications by means of floating:

$$P_i \times \left( n - \left\lfloor n \times \frac{1}{P_i} \right\rfloor \right)$$

This writing in the form of a sequence allows to uniquely define each integer $n$. $\square$

Given the random nature of the numbers (or pseudo-random, since the identifiers of the *items* are generated by cryptographic hashing technologies), the tree $T$ is balanced.

To unbalance it in a malicious way, it would be necessary to be able to generate hashes whose modulo follows a particular trajectory.

However, the difficulty of such an operation increases exponentially (in the order of $e^{(k \times log(k))}$ where $k$ is the depth of $T$).

As a reminder, the product of the first 16 prime numbers already equals:

$$32,589,158,477,190,044,730 \simeq 3 \times 10^{19}$$

Therefore, as soon as the index contains a reasonably large amount of data, unbalancing the tree in a malicious way would become more and more impossible, if at all possible.

## 3. Leaf is information on item

Let $s$ be a suchain of linked *items*. For example, it could be a sequence of transactions between two stakeholders defining the progressive evolution of the terms of their smart contract.

And let $s_0$ be the first *item* in a subchain of linked *items*.

A leaf $\lambda_{i_s} \in T$ contains the following list of information about an *item* $i$ referred to by its identifier $\iota_{i_s}$ in subchain $s$:

- Identifier (ID) of the current *item* (as a hash string):
$$\lambda_{i_s}^{ID} := \iota_{i_s}$$
- Position, ie. start address of the current *item* in the file, eg.
$$\lambda_{i_s}^{Pos} := 12080$$
- Size (in bytes) of the saved item in the file, eg.
$$\lambda_{i_s}^{Size} := 2074$$
- Origin, ie. the unique identifier of the *item* that is at the start of the *item*'s subchain (if any):
$$\lambda_{i_s}^{Origin} := \iota_{i_{s_0}}$$

- Previous, ie. the optional unique identifier of the previous *item* chained to it:

$$\lambda_{i_s}^{Prev} := \iota_{i_{s-1}}$$

- Next, ie. the optional unique identifier of the next *item* chained:

$$\lambda_{i_s}^{Next} := \iota_{i_{s+1}}$$

A leaf whose next *item* field is empty is the last item in the subchain:

$$\lambda_{i_s}^{Next} = \emptyset \iff \iota_{i_{s+1}} \notin T \qquad (2)$$

A leaf whose origin *item* field is equal to the identifier of the current *item* is necessarily the origin of the subchain:

$$\lambda_{i_s}^{Origin} = \lambda_{i_s}^{ID} \iff s = s_0 \qquad (3)$$

As such, it has a particular operating since, if there were to be one or more *items* thereafter, the last *item* of the subchain will be identified here as its previous item. Therefore, let $s_z$ be the last index in a subchain of linked *items*, we have:

$$\begin{cases} \lambda_{i_s}^{Origin} = \lambda_{i_s}^{ID} \\ \lambda_{i_s}^{Prev} \neq \emptyset \end{cases} \iff \lambda_{i_s}^{Prev} := \iota_{i_{s_z}} \qquad (4)$$

The last three fields of the leaf therefore transforms $s$ as a circular linked list.

## III. Implementation

### 1. Performance

Lorem ipsum ...

### 2. Memory

Lorem ipsum ...

## References

[1] Gauss. *Disquisitiones Arithmeticae*, translated by Arthur A. Clarke, Springer, 1986.