ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Blockchain collections

Matteo Monti

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

January 2018

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Abstract

Blockchains are often used as a decentralized mechanism to store and evolve in time data that can be represented as a set of *key/value* associations. Among the most notable examples of this are cryptocurrencies, which store ledgers of transactions whose application results in a set of associations between public keys and account balances. In order to determine the value associated to a key, or to validate new blocks, nodes are often required to store either a full copy of the *ledger* (i.e., the complete set of transactions in the blockchain) or at least a full copy of the resulting *database* (i.e., the set of *key/value* associations resulting from the application of all the transactions in the ledger). This results in a high storage complexity that usually prevents lightweight nodes (like mobile phones or web applications) from using the blockchain without the aid of trusted third parties.

In order to address this problem, we introduce *collections*, a Merkle-tree based *key/value* store. Collections can be used to securely store a set of *key/value* associations (or *records*) on one or more untrusted servers. From the set of records in a collection, an $O(1)$-size *state* can be efficiently computed. Nodes holding a copy of the collection's state can query the untrusted server(s) for records, and securely verify their responses. Moreover, nodes holding a copy of an old state of the collection can, provided with an *update*, verify its applicability and autonomously and efficiently compute the new state of the collection, resulting from the application of the update.

In this report, we provide an overview on how collections are designed, provide arguments for their security and describe a blockchain architecture with total space complexity quasilinear in the number of records. We then briefly discuss and benchmark a full Go collections implementation, that we release as an easy-to-use, open-source library. Finally, we propose to use collections to develop a decentralized, highly efficient database optimized for low-energy, high-uptime devices.

Introduction

Motivation

Blockchains Blockchains (see, e.g., [6]) are becoming an increasingly popular solution to store and evolve data in time in a trustless environment. Nodes partaking in a blockchain store a sequence of *transactions* in a chain of *blocks*; each block includes the hash of the previous block, and a Sybil-resistant, decentralized protocol allows nodes to generate new blocks. Honest nodes trust and extend the longest valid chain, and use its content to validate new transactions. Under the assumption that the majority of nodes is not cooperating to attack the network, a blockchain can be shown to be tamper and censorship resistant.

Scope of this work Throughout this work, we will call *ledger* the sequence of transactions in a blockchain, and *database* the data that results from the successive application of all the transactions to an *initial* (or *empty*) database. For example, the ledger of a cryptocurrency stores the sequence of payments among the cryptocurrency's wallets, while its database is represented by the balance of each wallet (which can be computed by successively applying all the transactions in the ledger to an initial database where all accounts have null balance).

This work studies those blockchains whose database can be expressed as a set of *key/value* associations. This includes, for example: cryptocurrencies (whose database is a set of associations between public keys and account balances); blockchain-based certificate authorities (whose database is a set of associations between, e.g., domain names and public keys); art authenticity services (the like of, e.g., *Verisart*, whose database includes a set of associations between artwork identifiers and owner identifiers).

Motivation In a traditional blockchain architecture, in order to determine the value associated to a given key in the database, or to verify a new transaction, a blockchain node usually needs to either store a full copy of the ledger, or at least a full copy of the database.

While, due to its append-only nature, a ledger can always be shown to be larger than its corresponding database, both can become extremely large in real-case scenarios. For example, Figure 1 shows the time evolution of (a) the size of the Bitcoin ledger (currently over 150GB) and (b) the number of Bitcoin wallets (i.e., the number of keys in its database, currently over $22 \cdot 10^6$). Since 2014, both values show an exponential trend.

Due to its potentially very large space complexity, a traditional blockchain easily becomes prone to exluding nodes with limited storage capacity, that could otherwise contribute to the overall security of the system. Moreover, lightweight clients, like cellphones and web applications, are sometimes forced to resort to trusted third-parties in order to perform queries on the database, which ultimately undermines the decentralized nature of the blockchain.

Goal of this project This work aims at reducing the per-node space complexity of blockchain architectures, without compromising security and/or introducing trusted third parties. In order to do so, we develop *collections*, a Merkle-tree based *key/value* dictionary that we will use to store the blockchain's database.

The content of a collection can be stored on an untrusted server, or sharded among multiple nodes. From the set of records in a collection, an $O(1)$-size *state* can be efficiently computed. Nodes and clients holding a copy of the collection's state can query the untrusted servers for records and securely verify their responses. Moreover, nodes holding a copy of an old state of the collection can, provided with an *update*, verify its applicability and autonomously and efficiently compute the new state of the collection, resulting from the application of the update.

We express transactions on a blockchain as updates on a collection. At the end of each block, we add an *epoch state* resulting from the application of all the updates in the block to the *epoch state* of the previous block (see Figure 2). As in a traditional blockchain paradigm, nodes and clients download the blockchain and verify its validity. However, once the blockchain is successfully verified, a node only needs to permanently store the state of the collection (32B in a real-case scenario) in order to securely perform queries on the database and verify new updates.

Structure of this work

In Section 1 we discuss collections: we describe a strawman approach to storing *key/value* associations on a Merkle tree, then develop the tools needed to overcome its limitations, extending an authenticated dictionary used by CONIKS [2], a key transparency architecture for online authorities. In Section 2 we provide an overview on the collections library and benchmark its performance. In Section 3, we discuss two potential future developments of this project. First, we describe how a collections-based database can be implemented on top of a Catena log [8], leveraging the double spending resistance of Bitcoin

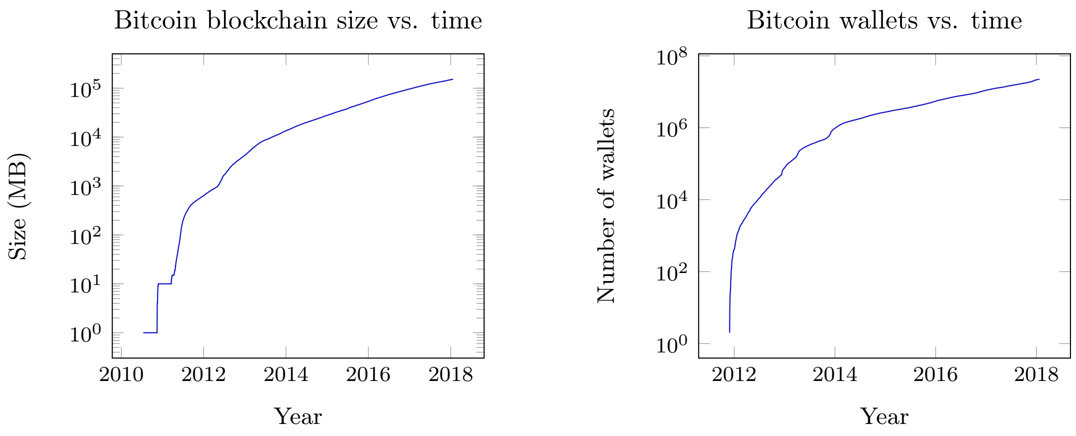

Figure 1: (a) Size of the Bitcoin blockchain vs. time. (b) Number of wallets in the Bitcoin blockchain vs. time. Both values show an exponential behavior starting from year 2014. Source: blockchain.info.

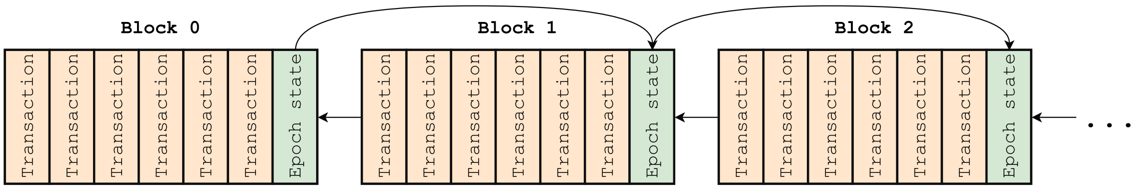

Figure 2: To each block we append an *epoch state*, resulting from the application of all the updates in the block to the epoch state of the previous block. From the updates and the old epoch state, nodes can autonomously verify that the new epoch state was computed correctly.

to provide non-equivocation and transparency to a public registry. Finally, we introduce Leaf, a highly efficient collections-only database for low-energy, high-uptime devices.

1 Collections

1.1 Overview

In this section we discuss how collections are designed, and provide arguments for their security. As we mentioned in Introduction, a *collection* is an authenticated data structure storing a set of *key/value* associations (or *records*).

From the set of records stored in a collection, a fixed-size *state* can be efficiently computed. Provided with a copy of its *state*, a *verifier* can perform queries on a collection stored by an untrusted *server*, and securely verify its responses.

Moreover, provided with the old state of a collection and an *update*, a *verifier* can autonomously and efficiently compute the new state of the collection, resulting from the application of the update.

1.2 Formal definition

Let

$$R = \{(k_1, v_1), \dots, (k_N, v_N)\}$$

(k_i and v_i being finite strings of bits) be a set of *key/value* associations. A collection provides:

- A function

  $$\texttt{state}(R) \in \{0,1\}^n$$

  n being a security parameter of the collection. Let $S = \texttt{state}(R)$.

- Two functions `prove` and `verify` so that:

  $$\begin{aligned} \texttt{prove}(R, k_i) &= (S, P_{S,k_i}, k_i, v_i) \\ \texttt{prove}(R, k^* \notin \{k_N, \dots, k_N\}) &= (S, P_{S,k^*}, k^*, \emptyset) \end{aligned}$$

  and

  $$\begin{aligned} \texttt{verify}(S, P, k, v) &\in \{\text{accept}, \text{reject}\} \\ \texttt{verify}(S, P, k, v \neq \emptyset) &= \text{accept} \Leftrightarrow (k, v) \in R \\ \texttt{verify}(S, P, k, \emptyset) &= \text{accept} \Leftrightarrow \nexists\, v \text{ s.t. } (k, v) \in R \end{aligned}$$

- Three *manipulators*

  $$\begin{aligned} \texttt{add}(R, (k, v) \notin R) &= (S, P^{+}_{S,(k,v)}, k, v, S^{+}_{k,v}) \\ \texttt{remove}(R, k_i) &= (S, P^{-}_{S,k_i}, k, S^{-}_{k_i}) \\ \texttt{update}(R, k_i, v'_i) &= (S, P^{\sim}_{S,(k_i,v'_i)}, k_i, v'_i, S^{\sim}_{k_i,v'_i}) \end{aligned}$$

with

$$\begin{aligned} S^{+}_{k,v} &= \texttt{state}(R \cup \{(k, v)\}) \\ S^{-}_{k_i} &= \texttt{state}(R \setminus \{(k_i, v_i)\}) \\ S^{\sim}_{k_i,v'_i} &= \texttt{state}(R \setminus \{(k_i, v_i)\} \cup \{(k_i, v'_i)\}) \end{aligned}$$

and their corresponding *verifiers*

$$\begin{aligned}\texttt{add}^{(verif)}(S, P^{+}_{S,(k,v)}, k, v) &= S^{+}_{k,v}\\ \texttt{remove}^{(verif)}(S, P^{-}_{S,k_i}, k_i) &= S^{-}_{k_i}\\ \texttt{update}^{(verif)}(S, P^{\sim}_{S,(k_i,v'_i)}, k_i, v'_i) &= S^{\sim}_{k_i,v'_i}\end{aligned}$$

Explanation From the set of records R, an n-bit `state` can be computed. From R and k_i, `prove` can produce a tuple that includes the state S of the collection and a *proof* based on the state of the collection that a record exists for k_i and has value v_i. Provided with a proof P for an association (k, v), `verify` yields `accept` if and only if $(v, k) \in R$ and P is well formed. Proofs of non-inclusion can also be generated and verified.

Each manipulator computes the new state of the collection after the manipulation has taken place, and a proof for that manipulation. Provided with a proof, a manipulation verifier can autonomously recompute the new state of the collection, after the manipulation has taken place.

1.3 Merkle tree strawman

Merkle trees [3] are a popular cryptographic tool used to authenticate large sets of data blocks, allowing for compact proofs of inclusion and modification.

Let $\{D_1, \ldots, D_M\}$ be a set of bit strings. For the smallest $L \geq \log_2(M)$, $D_1, \ldots, D_M$ can be organized on the leaves of a balanced binary tree with height L. A Merkle tree recursively labels each internal node with a cryptographic hash of the labels of its children. Figure 3 shows an example Merkle tree storing 8 data blocks.

Let $P_0, \ldots, P_L$ be a path on the tree, P_0 being the root and P_L being the leaf whose label is D_i. Then the labels of $P_0, P_1, \sigma(P_1), \ldots, P_L, \sigma(P_L)$, $\sigma(N)$ being the sibling node of N, form an inclusion proof for D_i.

Noting that a *key/value* association can be represented in a data block (e.g., (k, v) can be represented in a string in the form "$k \rightarrow v$") our strategy will be to organize all associations on the leaves of a Merkle tree, and use the label of the root of the Merkle tree as state of the collection.

As a strawman design, we could set $M = N$, $D_i =$ "$k_i \rightarrow v_i$", and store the data blocks on the leaves of the Merkle tree in a similar way to the example in Figure 3. It is easy to see, however, that while inclusion proofs can be efficiently produced, a valid exclusion proof comprises of the whole tree.

Following from Figure 3, in order to prove the exclusion, e.g., of "kangaroo", one would have to produce inclusion proofs for all D_i and show that, for every i, $D_i \neq$ "kangaroo". This is due to the fact that records can be arbitrarily organized on the tree.

1.4 Prefix trees

As we have seen in Section 1.3, we will encode $\{(k_i, v_i)\}$ in data blocks that will be stored on the leaves of a Merkle tree, and use the label of the root of the Merkle tree as state of the collection. We have shown, however, that if data blocks can be arbitrarily arranged on the leaves of the tree, exclusion proofs become impractical to produce.

Efficient exclusion proofs can be produced, however, if the leaf storing each (k_i, v_i) is forced to lie **along a path starting from the root and uniquely determined by** k_i. In order to prove that some k^* is not present on the tree, it would be sufficient to:

- Starting from the root, navigate along the path determined by k^*. For every node N on the path, append to the proof the label of N and $\sigma(N)$.
- When a leaf is reached, show that its label is not in the form "$k^* \rightarrow v^*$".

It is easy to see that a path on a binary tree starting from the root can be expressed as a sequence of edges, each connecting an internal node either to its left or right child, until a leaf is reached. Therefore, a path on a binary tree can be easily expressed as a sequence of bits, where 0 represents an edge connecting a node to its left child and an 1 represents an edge connecting a node to its right child.

Let $p_i = \texttt{path}(h(k_i))$, where h is a cryptographic hash function and `path` inputs a sequence of bits and outputs a path on a binary tree, as described in the previous paragraph. We will organize the records $(k_i, v_i)_{i \in [1,N]}$ on the tree according to the following three rules:

1. Each record is stored in a distinct leaf of the tree.
2. The leaf storing (k_i, v_i) will always lie **along** p_i.
3. The tree has minimal size.

Rule 1 accounts for efficient proofs of inclusion, as showed in Section 1.3.

Rule 2 accounts for efficient proofs of exclusion: since the path along which a record has to lie can be autonomously computed by the verifier by hashing the key, both inclusion and exclusion proofs reduce to navigating along a path until a leaf is encountered.

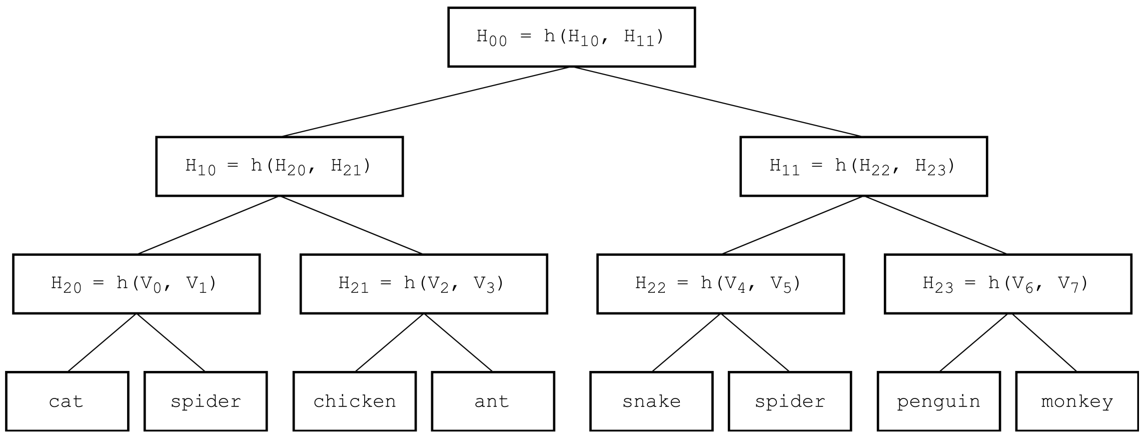

Figure 3: An example Merkle tree storing 8 data blocks. Data blocks are stored on the labels of the leaves, and each internal node is recursively labeled with the hash of the labels of its children.

Rule 3 accounts for space efficiency: indeed, let n be the number of bits produced by h, one could store all associations on a balanced binary tree of height n. While the security of the cryptographic hash would still ensure that each record is stored on a distinct leaf, the tree would have $2^n - 1$ nodes!

It is easy to see that the tree produced by Rules 1, 2, and 3 is not balanced. In order to maintain it binary, *empty leaves* (which store no association) are added to the tree where needed.

Figures 4 and 5 show an example of how a set of records can be organized on the leaves of a Merkle tree, in accordance with Rules 1, 2 and 3.

1.5 Advanced features

Collections can be extended to account for more properties than what we discussed throughout Section 1. Here we briefly enumerate the additional features that were developed throughout this project, and outline their design.

Weighted collections We extended collections to associate to each key not only a value, but also a positive, integral weight. Weights are propagated to internal nodes, the weight of each node being equal to the sum of the weights of its two children. Empty nodes always have weight zero.

Weights allow a verifier to navigate a collection not only by key, but also by *cumulative inversion*: given a random value between zero and the weight of the root (which represent the total weight of all the records in the collection), one can efficiently extract a record with probability proportional to its weight. This is useful in proof-of-stake scenarios, where nodes need to be drawn with probability proportional to some amount of stake.

Batch updates In Section 1.2 we introduced manipulators that perform single-record operations and produce proofs that enable a verifier to compute, from the old state of the collection, the new state resulting from the manipulation performed.

This, however, requires nodes to retrieve and apply all outstanding updates before releasing their own, condition impossible to satisfy in a large decentralized paradigm where multiple nodes release updates simultaneously and synchronize at regular epochs.

First, we generalized updates to atomically affect multiple records. Then, we developed an algorithm to apply a batch of updates, all starting from the same state of the collection. Conflicting updates are dropped, exactly as if they were pushed to an outdated version of the database.

Sharding Throughout Section 1, we described collections as an indivisible tree. This is inefficient in very large scale scenarios, where the number of records could be too large to be stored even by one dedicated (albeit untrusted) server.

We extended collections with two more algorithms, `drop` and `restore`. `drop` effectively prunes a local copy of the collection's Merkle tree, discarding values, weights and descendants of a node, turning it into a *stub*. Provided with an inclusion proof for a record, `restore` re-extends the tree, restoring some of the nodes under a stub.

This allows nodes to store arbitrary subsets of the collection's records. Nodes can produce inclusion proofs for all the records they store, and an inclusion proof is sufficient to extend a local copy of a collection to include the proven record. This enables arbitrary sharding of a collection among multiple nodes. In particular, any Distributed Hash Table algorithm (see, e.g., [7])

$\mathbf{k_i}$	$\mathbf{v_i}$	$\mathbf{(h(k_i))_{16}}$	$\mathbf{(h(k_i))_2}$	$\mathbf{p_i}$
`cat`	`four`	`c7580f..`	`1100 0011 0101 1000..`	↘ ↘ ↙ ↙ ↙ ↙ ↘ ↘ ↙ ↘ ↙ ↘ ↘ ↙ ↙ ↙..
`spider`	`eight`	`09d063..`	`0000 1001 1101 0000..`	↙ ↙ ↙ ↙ ↘ ↙ ↙ ↘ ↘ ↘ ↙ ↘ ↙ ↙ ↙ ↙..
`chicken`	`two`	`33cd78..`	`0011 0011 1100 1101..`	↙ ↙ ↘ ↘ ↙ ↙ ↘ ↘ ↘ ↘ ↙ ↙ ↘ ↘ ↙ ↘..
`ant`	`six`	`3a236f..`	`0011 1010 0010 0011..`	↙ ↙ ↘ ↘ ↘ ↙ ↘ ↙ ↙ ↙ ↘ ↙ ↙ ↙ ↘ ↘..
`snake`	`zero`	`6d717f..`	`0110 1101 0111 0001..`	↙ ↘ ↘ ↙ ↘ ↘ ↙ ↘ ↙ ↘ ↘ ↘ ↙ ↙ ↙ ↘..

Figure 4: Example set of *key/value* associations to store in a collection. Keys appear in the first column, values in the second. The hexadecimal and binary representations of the hash of the key appear in the third and fourth column respectively, and paths are represented in the last column. All hashes are truncated to the 16th bit; their value was determined randomly.

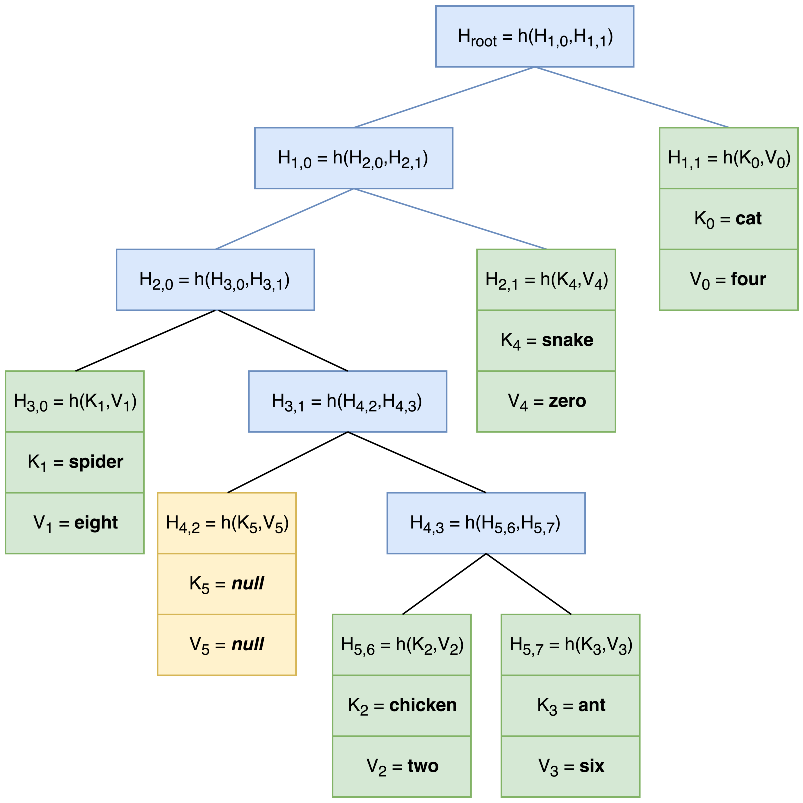
H_{root} = h($H_{1,0}$,$H_{1,1}$)
$H_{1,0}$ = h($H_{2,0}$,$H_{2,1}$)
$H_{1,1}$ = h(K_0,V_0)
K_0 = **cat**
V_0 = **four**
$H_{2,0}$ = h($H_{3,0}$,$H_{3,1}$)
$H_{2,1}$ = h(K_4,V_4)
K_4 = **snake**
V_4 = **zero**
$H_{3,0}$ = h(K_1,V_1)
K_1 = **spider**
V_1 = **eight**
$H_{3,1}$ = h($H_{4,2}$,$H_{4,3}$)
$H_{4,2}$ = h(K_5,V_5)
K_5 = ***null***
V_5 = ***null***
$H_{4,3}$ = h($H_{5,6}$,$H_{5,7}$)
$H_{5,6}$ = h(K_2,V_2)
K_2 = **chicken**
V_2 = **two**
$H_{5,7}$ = h(K_3,V_3)
K_3 = **ant**
V_3 = **six**

Figure 5: Merkle tree of a collection storing the records shown in Figure 4. Internal nodes are represented in blue, leaves storing an association are represented in green, and leaves that are not storing an association (whose only purpose is to keep the tree binary) are represented in yellow. Note how each record (k_i, v_i) is stored along p_i (in accordance with Rule 2) and as close as possible to the root (in accordance with Rule 3). Unlike the tree in Figure 3, this tree is not balanced.

can be used to partition the records among a distributed and redundant set of nodes.

2 Software

All the features described in Section 1 have been implemented in a library, written in `Go` language. Here we briefly discuss its main features, and measure its performance.

2.1 API

Collections and fields The library exposes a `collection` structure whose methods implement all the algorithms described in Section 1.

A `collection` can be configured to associate to each key an arbitrary tuple of *fields*. In order to account for future extensions, a `field` interface is exposed: by defining its encoding, and if and how values propagate to internal nodes, a user can define new types of fields. Raw data fields and integral weights (which we discussed in the previous section) are already implemented and can be used out-of-the-box. Potential upcoming fields include accumulators (which propagate to internal nodes by union) and database indexes (which would enable more elaborate queries on the database).

Manipulators and updates Manipulators are directly exposed, allowing to develop simple applications where serialization and atomicity are less critical. To account for more complex scenarios, an `update` interface is exposed for the user to implement. An `update` is implemented by defining the set of records affected by the update, a condition for the update to be applied and how it is applied. All the collection-specific mechanisms are hidden from the interface, allowing the user to define updates as she would do with a non-authenticated, local *key/value* store.

Transactions By default, updates are applied sequentially, each resulting in a new state of the collection. In order to apply batches of updates, a transaction interface is exposed. All the updates applied between a call to `begin` and a call to `commit` must be based on the same state of the collection. The new state of the collection is computed only when a transaction is committed: applying updates in batches is therefore more efficient than applying them separately. A `rollback` method is offered to revert to the initial state of a transaction, discarding all the changes performed.

Sharding By default, a `collection` instance is configured to store all the records in the collection. The user can personalize what records should be stored by the instance by specifying one or more `scopes` (i.e., bit-prefixes). Only the records the hash of whose key begins with one of the bit-prefixes defined in the collection's `scopes` are permanently stored.

When applying updates, the library automatically extends the records stored in a shard to include all the records affected by the update. After the update is applied, all the records in the shard remain consistent with the new root, even if the update affects records outside the scope of the shard.

An automated garbage collection mechanism is implemented to `drop` out-of-scope records when they are no longer necessary.

2.2 Performance

A benchmark was run to test the performance of the collections library. The test was performed on a `MacBook Pro (Retina, 15-inch, Mid 2014)`, mounting a `2.5 GHz Intel Core i7` processor. The time needed to perform manipulations was measured against the size of the collection (in records).

Figure 6 shows the result of the benchmark. Manipulation times are contained under $60\mu s$ up to $6 \cdot 10^6$ records, with a baseline that shows a logarithmic behavior. However, for reasons yet to be determined, benchmarks are very irregular. We hypothesize this being due to `Go`'s garbage collection process. This hypothesis seems to be supported by the fact that more than one thread displays CPU-intensive activity when manipulators become slower, and that update manipulators, which don't alter the structure of the tree, display a more regular timing.

Further inspection will include a more detailed benchmark using a time profiler, and an implementation sketch in a non-garbage collected language to test against the `Go` version of the library.

3 Future developments

3.1 Catena + collections

We contributed to a design proposal to use collections in the development of an equivocation-resistant, transparent public registry. Using a design similar to the one shown in Figure 2, a centralized server could use collections to manage a *key/value* database.

Clients can perform queries on the database, to which the server will respond with proofs of

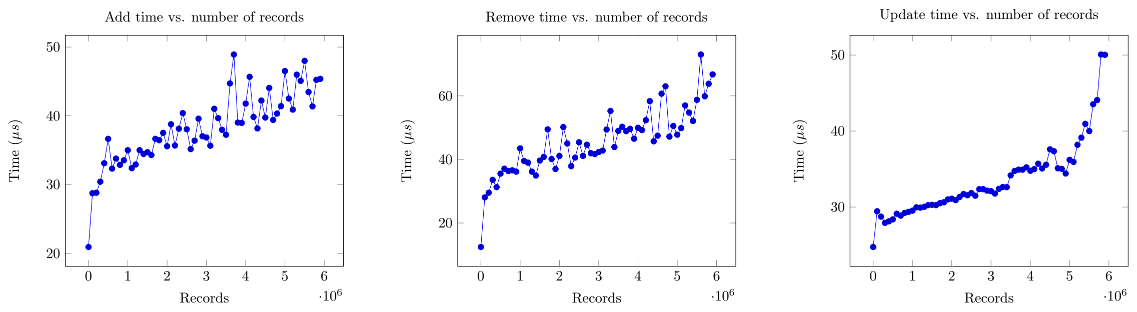

Figure 6: Manipulators benchmark on a `2.5 GHz Intel Core i7`. All figures show a logarithmic, but very irregular behavior. Further inspection is required to determine the cause of such an irregular performance.

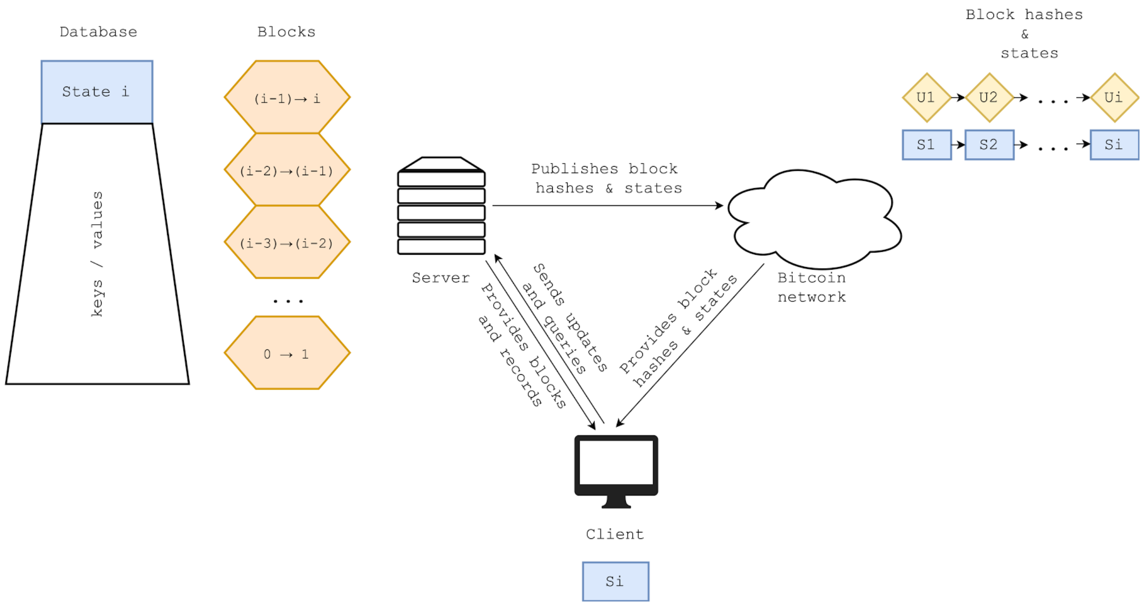

Figure 7: Equivocation resistant database proposal using Catena logs and collections. Updates are organized into blocks, whose hashes are committed to a Catena log on the Bitcoin blockchain. Clients use Simplified Payment Verification to efficiently retrieve the log from Bitcoin, and the server to retrieve the update blocks.

inclusion. A log of the changes is organized in blocks, which the clients will verify. In order to prevent equivocation, the hash of each block is committed to a Catena log [8] on the Bitcoin blockchain.

Leveraging the double-spending resistance of Bitcoin, Catena provides non-repudiation to an append-only log of statements, that we use to commit to the block hashes. Using Simplified Payment Verification, clients efficiently retrieve the log from the Bitcoin network, and use its content to verify the blocks provided by the server.

As in a traditional client-server paradigm, clients push their updates to the server, that acts as a bottleneck to serialize them and organize them in block.

Figure 7 sketches the proposed system's architecture. A more complete draft of this proposal, including an API, can be found here [4].

3.2 Leaf

Throughout the course of this project, we worked on the high-level design of Leaf, a distributed and decentralized high-performance database that uses collections as core component. Leaf is optimized for a swarm of low-energy devices that can be assumed to have a relatively high (e.g., 90%) uptime.

Features

Our preliminary results show that Leaf: (1) is secure as long as the majority of storage space is controlled by nodes that are not cooperating to attack the network, (2) can scale to a practically unlimited number (e.g., 10^{12}) of full nodes, (3) allows sharding of records among all the nodes of the network, and has a total space complexity quasilinear in the number of records[1], (4) has a bootstrap communication complexity logarithmic in the number of records [2], (5) requires only a fixed-size subset of the nodes to verify each update, allowing for a practically unlimited number of updates per unit of time as the network grows, (6) does not require energy-intensive proofs-of-work, (7) can be used as an unbiasable randomness beacon.

Design overview

Nodes are assumed to have a relatively high (e.g., 90%) uptime, and to join and leave the network at a slow rate. This allows the nodes to store in the database, along with the client-issued records, an ordered list of their public keys.

Since a commitment to the label of the root of a Merkle tree computationally implies a commitment to all its content, new nodes and client only need to securely discover the label of the root to be able to verify queries both on the records and on the list of nodes. This allows for an highly efficient bootstrap protocol: a node with no previous information on the network will determine the consensus of nodes behind a state by randomly sampling its corresponding list of nodes and verify what fraction of them are online and correctly abide by a low-energy *proof of space* hardware commitment mechanism.

As in most distributed ledgers, updates are distributed among the nodes using *gossip protocols*, and a random, fixed-size set of nodes (which we call *boule*) is randomly selected at fixed time intervals to organize updates in blocks, which are then applied by each node to its shard of the database.

Since an ordered list of nodes is publicly available within the database itself, a decentralized random beacon can be used to efficiently select the boule. Since the label of the root of a Merkle tree is a a pseudorandom function over all its content, each node is allowed to contribute to the randomness of the root by updating its dedicated *seed* record with random values. In order to prevent biasing, random values are computed from the root of the Merkle tree using a long-to-compute, inherently sequential pseudo-random function [1].

As the boule is selected from a public randomness source, an opponent could perform a denial of service attack on the selected nodes, and prevent them from communicating new blocks to the network. In order to prevent this, nodes partake in an online *key shuffle* protocol: nodes are repeatedly organized in random couples; two nodes in the same couple can release a jointly signed update that removes their (ephemeral) public keys from the database, and adds two new ones, without disclosing which belongs to which node.

In order to prevent malicious boule nodes from waiting indefinitely before publishing their *block shard*, the boule partakes in an *avalanche timestamping protocol*, that can be shown to release valid block shards after a timeout with a vanishingly low probability: to release its block shard, a boule node needs to have it signed by the majority of the boule. An honest boule node adds its signature to a block shard only if it is received within a time frame that grows with the number of signatures the shard already collected.

In order to guarantee scalable concurrency, nodes are organized in 2^C *communities*, each managing a distinct C-deep subtree of the Merkle tree. Communities separately generate blocks and apply them to their subtree. After all blocks are applied, the global root of the tree is computed from the roots of the subtrees in logarithmic time. Nodes are forced to randomly migrate among the communities to prevent malicious nodes from overwhelming any specific community.

In order to apply updates that affect records managed by two or more distinct communities, to each record is associated a list of *pending updates*. Cross-community updates are appplied in two steps. First, they are added to the pending updates of all the records they affect. Then, nodes in each community separately retrieve the values and pending updates of the other records affected by the update in order to determine the final value of the record they manage. An algorithm based on the game of *Mahjong* allows nodes to apply multiple pending updates within the same block.

Current development

A document [5] is under development covering the main features of the database. In the upcoming months, the various sections will be filled with details. An experimental code base is also under development.

[1] For example, 10^6 nodes using 1GB of space each can collectively store 10^{11} records of size 1KB.

[2] For example, a node with no previous information can securely discover the state of a $4 \cdot 10^9$-records database exchanging only 65KB with the network.

References

[1] Arjen K. Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and

trx. Cryptology ePrint Archive, Report 2015/366, 2015. https://eprint.iacr.org/2015/366.

[2] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. Coniks: Bringing key transparency to end users. In *USENIX Security Symposium*, pages 383–398, 2015.

[3] R.C. Merkle. Method of providing digital signatures, January 5 1982. US Patent 4,309,569.

[4] Matteo Monti. Catena + collections (https://goo.gl/dsjk46), 2017.

[5] Matteo Monti. Leaf: a secure, high-performance, decentralized database for the internet of things [work in progress], 2018.

[6] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[7] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[8] A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via bitcoin. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 393–409, May 2017.