A Specification for the Unchained Index

trueblocks-core@v2.0.0-release

Thomas Jay Rush
Dawid Szlachta
Meriam R. Zandi

TrueBlocks, LCC
November 2023

Table of Contents

Introduction

Blockchain data and content-addressable storage have gotten married, and they've had a baby called the "Unchained Index."

The two technologies need each other. Immutable data needs a suitable storage medium. Otherwise, how can it possibly remain immutable? If one modifies immutable data, its location on the content-addressable store will change.

The two concepts are as tightly connected as the front and back sides of a piece of paper. One may try to pull them apart—but one will only end up with two slightly thinner pieces of paper. This is just a fact of nature.

This document describes the Unchained Index, a computer system that purposefully leans into this tight coupling between immutable data and content-addressable storage.

The mechanisms described in this paper apply to any immutable data (for example, any time-ordered log). For the purposes of this paper, the examples focus on the Ethereum Mainnet's index of address appearances as described below.

The Format of the Paper

This document begins by reviewing the Unchained Index. Following that, we present detailed descriptions of the binary file formats used in the system. Five binary file formats are described:

1) Manifest – a JSON object that carries enough information to reconstitute the index;

2) Index Chunks – a single portion of the entire index consisting of approximately 2,000,000 appearance records and covering a certain block range;

3) Bloom Filters – a Bloom filter encoding set membership of each address in the associated Index chunk covering the same block range;

4) Names – a somewhat unrelated collection of named address labels used to articulate query results. This small subset of known accounts contains about 13,000 records, and

5) Timestamps – a flat-file binary database used to optimize timestamp lookups.

The paper concludes by describing (and justifying) the algorithms used to create, read, write, and query the Unchained Index.

The Unchained Index

The Unchained Index is a *naturally-sharded*, *easily-shared*, *reproducible*, and *minimally-sized immutable index* for EVM-based blockchains.

By querying a smart contract, the system obtains an IPFS hash of a manifest file containing a list of other IPFS hashes that point to each *chunk* in the index database. (*Chunk* is defined below.) End users may query this manifest to obtain a list of "everything that ever happened" to an address on a given blockchain. The information contained in the manifest allows the user to reconstruct the transactional history of any address(es) without needing a third-party API. Furthermore, the manifest contains enough information to rebuild the index without permission.

As mentioned, the Unchained Index is a *naturally-sharded*, *easily-shared*, *reproducible*, and *minimally-sized immutable index* for EVM-based blockchains. What does that mean?

Naturally Sharded, Easily Shared

Unlike traditional databases, the Unchained Index is not stored in a single monolithic file on a server. Instead, it is stored as a collection of much smaller files ("*chunks*") and their associated Bloom filters. Breaking the index into chunks allows the system to use content-addressable storage systems such as IPFS. This design allows for a natural distribution of the index while requiring no "extra effort" from the end user. This aspect of the system is "*naturally sharded and easily shared*." A consequence of this design is that the system imposes a near-zero cost of production on the index's publishers, of which there may be many.

Because the index consists of chunks, end users can acquire only those portions of the index they need for their own use. A user's "need" is expressed naturally as a direct result of his/her queries for data. As the end-user explores the history of an address—that is, he exhibits interest in a particular address—the Unchained Index delivers only that portion of the index required to fulfill those specific queries.

This has the happy consequence that "light" users (i.e., users interested in only a few, lightly-used addresses—that is, most of us) carry a light burden, and "heavy" users (those interested in large smart contracts or data science, for example) require a larger number of chunks to satisfy their queries. As a result, heavy users carry the burden of a larger portion of the index than light users. Chunks are naturally shared, as described next. We think this is fair.

The system goes a step further and shares downloaded chunks (and associated Blooms filters) as a default. We call this aspect of the system "pin-by-default". As a side-effect, the system enlists the end-users in sharing the index without them making an "extra effort". Over time, the system becomes more and more distributed as each chunk becomes increasingly more downloaded and made available.

As the system matures, the index becomes shared fully among community members, making it (a) more resilient, (b) higher-performing as more copies are available for download, (c) more

resistant to censorship, (d) more difficult to capture, and (e) imposing a lessening burden on the original publishers as the end users themselves carry the burden of the publication.

Reproducible

The content-addressable nature of IPFS and similar systems also aids in making the Unchained Index reproducible. A primary data structure of the system is the Manifest.

As each chunk is produced, the block range covered by that chunk, the IPFS hash of the chunk, the IPFS hash of the chunk's Bloom filter, and the file sizes of both items are appended to the manifest. Periodically, the manifest itself is written to IPFS and its hash is enshrined in the Unchained Index smart contract.

[image]

The manifest contains enough information to make the Unchained Index *reproducible* in the following sense:

1. The manifest records the version of this specification ("trueblocks-core@v2.0.0-release").

2. The manifest also records the IPFS hash of this exact PDF document. In this way, end-users who acquire the manifest (through the smart contract) have a full specification of the system used to create the index. Once published, this version of the specification will never change.

3. The keccak_256 of the version string is inserted into the header of each binary file in the index prior to publishing it to IPFS. In this way, if the user obtains any portion of the index, he/she knows under which version of the specification the file was written.

4. The IPFS hash of the manifest is periodically posted to the smart contract, thereby enshrining it forever on the blockchain. Once published, the publisher (either TrueBlocks or anyone else) may no longer recant the information. The manifest and, as a result, the entire index is accessible to anyone for as long as the blockchain exists. This illustrates the need for pinning by default.

5. At some later point, if a user wishes to verify the contents of any portion of the index (or all of it), they may read the smart contract, download the manifest, download this document and the tagged commit of the source code, and re-run the code itself against their own blockchain node. Presumably, his will produce the exact same index.

6. We consider it the responsibility of the end-user to satisfy themselves as to the veracity of the data in the index.

Because the manifest contains enough information to reproduce the index, there is no need for end users to trust our data, and we do not expect them to. Nor do we feel the need to "prove" the

data or provide for "watchers" or "fishermen" to verify that the data is correct. It seems obvious that if a "fisherman" is needed to verify a system, then "fishers of fishermen" are also needed. And then "fishers or the fishers of fishermen." If the end user wishes to have proven data, she has all the tools necessary to prove the data herself.

TrueBlocks creates the Unchained Index for our own purposes. We desire that our end-user-focused software work properly. We are motivated to produce accurate data. While the system purposefully allows anyone to use the data by exposing it through the smart contract, we reject any sense of responsibility for the data. The data is correct because we need it to be. Other users may use it if they wish—but beyond that, we make no representations.

A Short Digression on the Use of Bloom Filters

Please see this excellent explainer on Bloom filters. A Bloom filter is "a space-efficient probabilistic data structure…used to test whether an element is a set member." This fits perfectly in our design.

The system produces an associated Bloom filter for each chunk, a much smaller file than the index chunk itself. The entire set of Bloom filters is required by all users.

Upon first use of the system, the end user must choose to download *only* the Bloom filters (about 4 GB), both the Bloom filters and the index chunks (about 100 GB) or, as a final alternative, the end user may create the index themselves.

These three methods are explained briefly here and more fully here.

chifra init: downloading only the Bloom filters from IPFS

Disc footprint:	Small, 4 GB
Download time:	15 minutes
Query speed:	Slower the 1st time one queries an address, subsequently as fast as other methods
Hard drive space:	In direct proportion to the user's query patterns
Sharing:	The user may share the Bloom filters and downloaded index chunks
Security:	Data is created by TrueBlocks, less secure than producing it oneself
RPC endpoints:	Works with remote RPC endpoint, but a local RPC endpoint is much preferred
Ongoing burden:	The end user must run 'chifra scrape' to maintain the front-of-chain index

When initialized with chifra init, TrueBlocks downloads only the Bloom filters for the given chain. Generally, this takes less than 15 minutes. When a user later queries an address (using chifra list or chifra export), the Bloom filters are consulted, and those portions of the full index that hit the Bloom filter are downloaded.

In this way, the end user only acquires index chunks that "matter to them." In other words, the system places a burden in direct proportion to the user's behavior. Users who interact infrequently with the chain get only a small amount of data. Queries against addresses that interact frequently with the chain—such as popular smart contracts—will hit on nearly every Bloom filter. In this case, the user will download a much larger percentage of the entire index.

In this first mode, a query for a never-before-queried address takes longer because the index chunks that hit the Bloom filter must be downloaded. Subsequent queries for the same address are as fast as other methods. Unless one is querying a large collection of different and quickly changing addresses, the cost of a slower initial query may be worth the benefit of a smaller disc footprint.

chifra init –all: downloading Bloom filters and the full index from IPFS

Disc footprint:	Large, ~100 GB at the time of writing
Query speed:	Very fast queries on all addresses as there is no additional downloading
Download time:	"a few" hours depending on connection speeds
Burden size:	The full index is stored on the end user's machine
Sharing:	The user may share the entire index (good citizen award!)
Security:	Data is created by TrueBlocks, less secure than producing it oneself
RPC endpoints:	Works with remote RPC endpoint, but a local RPC endpoint is much preferred
Ongoing burden:	The end user must run 'chifra scrape' to maintain front-of-chain index

If the user chooses to initialize his system with chifra init –all the entire Unchained Index is downloaded. This process may take several hours to complete depending on the end user's connection and the size of the chain. This is the recommended way to run if you have available disc space.

While, in this second method, the Bloom filters are still consulted during the query (because it's much faster than reading the index), there are no further downloads during the query. All index chunks are already present on the machine. This method is preferred if one is studying an address that appears frequently on the chain or many different addresses with varying usage patterns.

chifra scrape: building the index from scratch

Disc footprint:	Large, ~100 GB at the time of writing – same size as method 2
Query speed:	Fast queries – same as method 2
Download time:	"a few" days depending on the speed of node the machine
Burden size:	Full burden – same as method 2
Sharing:	Full sharing possible – same as method 2
Security:	Most secure, but not as secure as reviewing the open-source code first
RPC endpoints:	Generally, will not work with shared endpoints – you will by rate limited
Ongoing burden:	The end user must run 'chifra scrape' to maintain front-of-chain index

The third and final method is to build the index yourself. This is the most secure method, particularly if you review the source code first. One accomplishes this third method by running chifra scrape (which is the same command one must use to stay up to the head of the chain). Running against a remote RPC endpoint will most likely not work. TrueBlocks hits the node as hard as it possibly can. This method has the same disc usage and query characteristics as method 2. The only benefit is that you've built the index yourself.

Pinning by Default

In the currently available version of the Unchained Index, the system does not pin the downloaded or produced index by default. If you wish to pin the index, you may do so with `chifra chunks manifest --pin`. There are other options to allow you to pin the manifest remotely and rewrite the contents of the manifest to the data folder.

In future versions, pinning will be enabled by default as the chain is scraper. This will be an important day for TrueBlocks as it will allow TrueBlocks to finally become a truly decentralized method to produce, publish, and share an immutable index.

Pinning by default has the intended consequence that as users acquire and retain portions of the index for their own selfish reasons, they are sharing those portions with others. This happens without "extra effort" on the part of the end user—an important consideration in distributed systems. In other words, sharing happens as a by-product or natural outcome of the system. This is by design. We believe requiring "extra effort" from end users is a good way to make the system fail.

Obviously, most users will retain those portions of the index they need for their own purposes. And, while each chunk contains the user's individual records, they contain many additional records as well. "Pinning by default" takes advantage of this fact. It is a perfect example of "You scratch my back, I'll scratch yours."

The Unchained Index is available through a smart contract. And, the system is purposefully designed to distribute the index. It works this way because we wished to create a system with positive externalities—that is, a system in which each new user makes the system better as he or she joins, as opposed to placing an increasing burden on the system.

Conclusion of this Section

We've spent time explaining the Unchained Index because we hope it helps explain some of the engineering decisions we describe below. The system we describe is not a traditional "web2-database-running-in-the-cloud" type of thing.

In the next sections, we detail the Unchained Index smart contract. Following that, we describe the file format of the manifest. Following that, we describe the file formats for each of the four primary binary databases.

The stylized Solidity and GoLang code presented below will not run. We simplified it to aid in our explanations. See the GitHub repos for the actual source code.

The Unchained Index Smart Contract

The Unchained Index smart contract has been deployed on Ethereum Mainnet at address 0x0c316b7042b419d07d343f2f4f5bd54ff731183d.

This smart contract is purposefully permissionless. The reason for this and the workings of the code are explained in the comments.

The contract has not changed since the previous version of this specification, and we should note that by placing the contract's address into this document and the hash of this document into the manifest, we are purposefully making the index data reproducible.

```
pragma solidity ^0.8.13;

// The Unchained Index Smart Contract
contract UnchainedIndex_V2 {
    // The address of the account that deployed the contract. Used only
    // as the recipient of donations. This value may be modified.
    address public owner;

    // A map pointing from the address publishing a record to the record
    // itself. The record is a further map associating the database's name
    // with an IPFS hash of the full contents of the database. For example,
    // the mainnet database points to the manifest containing hashes for
    // every portion of the index. End users are encouraged to query the map
    // for a publisher they trust. Any publisher may write any number of records.
    mapping(address => mapping(string => string)) public manifestHashMap;

    // The constructor preserves the deploying address as the owner. It then
    // initializes a single record for the mainnet database pointing to the
    // manifest hash of an empty file. Two events are emitted.
    constructor() {
        // Store the deployer address for later use (see below)
        owner = msg.sender;
        emit OwnerChanged(address(0), owner);

        // Store a record, published by the deployer, indicating that the
        // manifest for mainnet is the empty file.
        manifestHashMap[msg.sender][
            "mainnet"
        ] = "QmP4i6ihnVrj8Tx7cTFw4aY6ungpaPYxDJEZ7Vg1RSNSdm"; // empty file
        emit HashPublished(
            owner,
            "mainnet",
            manifestHashMap[msg.sender]["mainnet"]
        );
    }
```

```
    // The publishHash function is purposefully very simple and permissionless.
    // Anyone may call this function. Anyone may also query the contract for
    // records published by any publisher. End users must determine for themselves
    // who to believe. We suggest it's us, but it's your choice.
    function publishHash(string memory chain, string memory hash) public {
        manifestHashMap[msg.sender][chain] = hash;
        emit HashPublished(msg.sender, chain, hash);
    }

    // We are happy to accept your donations in support of our work.
    function donate() public payable {
        // Only accept donations if there's an address to accept them
        require(owner != address(0), "owner is not set");
        payable(owner).transfer(address(this).balance);
        // Let someone know…
        emit DonationSent(msg.sender, msg.value, block.timestamp);
    }

    // The 'owner' address serves only the purpose of accepting donations.
    // If, at a certain point, we decide to disable or redirect donations
    // we can set this to a different address.
    function changeOwner(address newOwner) public returns (address oldOwner) {
        // Only the owner may change the owner
        require(msg.sender == owner, "msg.sender must be owner");
        oldOwner = owner;
        owner = newOwner;

        // Let someone know…
        emit OwnerChanged(oldOwner, newOwner);
        return oldOwner;
    }

    // Emitted each time a manifest hash is published
    event HashPublished(address publisher, string chain, string hash);

    // Emitted when the contract's owner changes
    event OwnerChanged(address oldOwner, address newOwner);

    // Emitted when a donation is sent
    event DonationSent(address from, uint256 amount, uint256 ts);
}
```

In the next section we define the binary file formats for the various databases used in the system.

File Formats

The Manifest File

The manifest file is updated with the chifra chunks manifest --pin command. It is a simple JSON object storing five data fields:

1) the version of this document which describes everything one needs to reproduce the index;
2) the name of the blockchain that the manifest indexes;
3) the IPFS hash of this exact PDF document;
4) the configuration settings used to create the index and
5) a list of chunks and associated Bloom filters detailing the entire index.

Note that the version string stored in the Manifest is the same version string of this document (trueblocks-core@v2.0.0-release).

A representative example of the JSON object is shown here:

```
{
  "version": "trueblocks-core@v2.0.0-release",
  "chain": "mainnet",
  "specification": "Qmart6XP9XjL43p72PGR93QKytbK8jWWcMguhFgxATTya2",
  "config": {
    "appsPerChunk": 2000000,
    "snapToGrid": 100000,
    "firstSnap": 2300000,
    "unripeDist": 28
  },
  "chunks": [
    {
      "range": "015013585-015016368",
      "bloomHash": "QmREw5qaoucbVvEQzF71D44rXKzax9YgKuEEhZYHAYFZF5",
      "indexHash": "QmTbFshRSdBFoC6AvBgzdRJ6Vgb9cVL3yTprYQ24XqHTqx"
    },
    {
      // and so on...
    }
  ]
}
```

The algorithm that produces the chunks is described in the section called *Building the Index and Bloom Filters*.

Like the index chunks, the manifest is pinned to IPFS as it's being created. The hash of that file, and thereby the hash of each chunk, is enshrined in the Unchained Index smart contract.

Once published, a number of important aspects of the index become true:

1) The publication record cannot be undone—the IPFS hash of this particular manifest will be on the chain forever readable by anyone with access to the chain;
2) If someone reads the manifest, he or she may download each individual chunk *and* this specification. In this way, the end user may read the index without asking any third party for permission. With the specification, the entire index may be reproduced if desired;
3) The publisher (in this case us, but it may be anyone) has no further ongoing cost of publication other than ensuring that the files are pinned (which they are because we want to provide this data to our paying customers). The cost to maintain the data, once pinned, is near zero. The publisher need not run a traditional web2 API. IPFS handles the data delivery seamlessly.

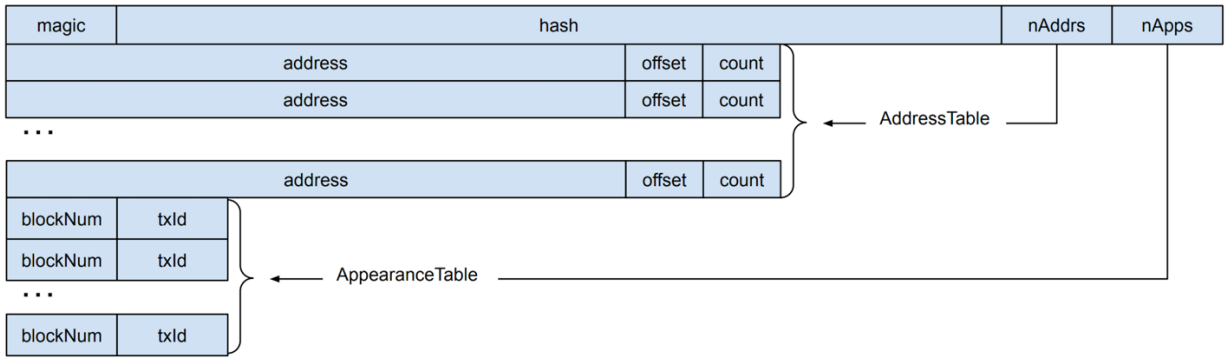
Over time, as more and more users download and pin more and more portions of the index—which happens through normal end-user usage—the resiliency and speed of the system increase. This is in proportion to the number of users. The more users using the system—the more resilient the system becomes. This is the essence of the Web 3.0 ethos. Positive externalities. This is a classic case of a positive sum game and the "if we all build it, we can all come" mentality.

As mentioned, the Manifest lists the hashes of each individual chunk whose file format is described next.

The Index Chunk File

Next, we describe the format of the index chunk files graphically as well as GoLang structures. Following that, we detail the algorithm one would use to read this file. There are approximately 4,000 index portions covering the 19,000,000 Ethereum mainnet blocks at the time of this writing and the same number of associated Bloom filters.

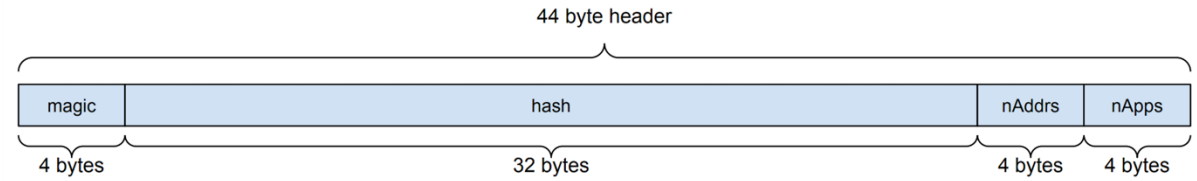
The binary *Index Chunk* file consists of a single fixed-width header followed by two fixed-width tables. Graphically, the file looks like this:



The same file format expressed as a Golang structure is here:

```
// Binary format of the Index Chunk file.
type IndexChunk struct {

    // A fixed-width record containing versioning information and two record
    // counts (nAddresses and nAppearances) one each for each of two tables.
    Header          HeaderRecord

    // A fixed-width table containing nAddresses AddressRecords
    AddressTable    []AddressRecord

    // A fixed-width table containing nAppearances AppearanceRecords
    AppearanceTable []AppearanceRecord
}
```

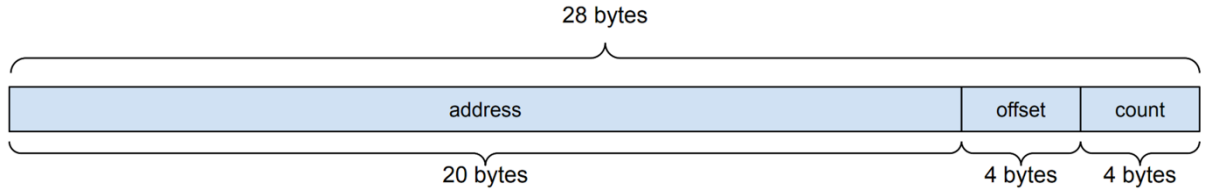
The *HeaderRecord*, which contains versioning information and counters for two tables, is expressed graphically as:



and rendered in Golang as follows:

```
// The structure of the Header record (44 bytes)
type HeaderRecord struct {

    // '0xdeadbeef' indicates that this is a known file format
    Magic         [4]byte

    // A byte string representing the keccak256 of the version string
    // (trueblocks-core@v2.0.0-release) of this specification.  This value
    // ensures that anyone receiving this file knows how to read it.
    Version       [32]byte

    // A count of the number of records in the AddressRecord table
    nAddresses    uint32

    // A count of the number of records in the AppearanceRecord table
    nAppearances  uint32
}
```

The *AddressTable*, which follows immediately after the Header, contains *header. nAddrs* fixed-width records of type *AddressRecord*. Each record relates to the *AppearanceTable* via the *Offset* value, which points to the position in the *AppearanceTable* for this address's appearance list.



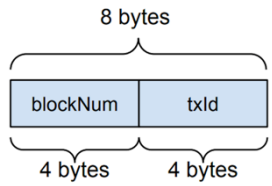
For each address found in the block range, the *AddressTable* stores the address and two unsigned integers. The first integer is the *Offset* into the *AppearanceTable* where this address's appearance list begins. The second integer, *Count*, is the number of appearance records to read.

In Golang code, the *AddressRecord* looks like this:

```
// The AddressRecord structure
type AddressRecord struct {

    // a 20-byte Ethereum address
    Address [20]byte

    // The offset into the appearance table to the address's first record
    Offset  uint32

    // Number of records in the appearance table to read
    Count   uint32
}
```

The *AppearanceTable*, which follows immediately after the *AddressTable*, consists of an array of *header.nApps* fixed-width *AppearanceRecords*. The records consist of two 32-bit unsigned integers representing the *blockNumber* and *transactionIndex* of the appearance.

This structure is graphically presented here:



and rendered in Golang code here:

```
// An appearance is a <blockNumber><tx_id> pair. One for each time an address
// appears anywhere in the chain data.
type AppearanceRecord struct {

    // The block number for the appearance
    BlockNumber      uint32

    // The transaction id for the appearance
    TransactionIndex uint32
}
```

The algorithm for reading an *Index Chunk* file is presented next. Note, that we've removed error processing for clarity.

```
// Search a single chunk for an address
func GetAppearances(addr address, chunkFilename string) []AppearanceRecord {

    // Open the file
    fp := os.Open(chunkFilename)

    // Read the header
    header := HeaderRecord{}
    binary.Read(fp, binary.LittleEndian, &header)

    // Where do the tables start?
    offsetToAddrTable := os.Seek(fp, io.SeekCurrent)

    // Conduct a binary search on the address table
    found := binarySearch(fp, addr, offsetToAddrTable)
    if found == nil {
        // Address not found, return empty array
        return []AppearanceRecord{}
    }

    // Where the appearance table resides
    offsetToAppsTable := offsetToAddrs +
                         (header.nAddresses * os.Sizeof(AddressRecord)))

    // Go to the start of this address's appearance records
    fp.Seek((offsetToAppsTable + found.Offset, io.SeekStart)

    // We know how big the results array has to be…
    apps := make([]AppearanceRecord, found.Count)
    binary.Read(fp, binary.LittleEndian, &apps)

    return apps
}
```

Performance Note: In the actual code, this function accepts an array of addresses. In this way, we avoid spinning through the chunks repeatedly for each address. In the actual code, we return an array of arrays of *Appearances*. We've removed this feature for clarity.

Performance Note: Because the *GetAppearances* reads from disc, it is desirable to avoid running it if at all possible. This is the reason the system uses Bloom filters.

The Bloom Filter File

As described above, the Unchained Index is a collection of binary files called *chunks*. Taken as a whole, the *chunks* can be seen as a naturally-sharded database. Each chunk contains a near-equal number of records. In this way, we ensure a fair distribution of the index via content-addressable storage such as IPFS while imposing a minimal cost of publication on publishers.

In order to optimize searching the chunked index, we produce a second collection of binary files called Bloom filters. Each *chunk* has an associated Bloom filter.

A Bloom filter is a probabilistic data structure that encodes set membership in a very compact way. On average, each *chunk* is around 25 megabytes big and contains approximately 2,000,000 appearance records (for Ethereum Mainnet, indexes for other chains will have different characteristics). The associated Bloom filters are, on average, one megabyte big. Twenty-five times smaller than its *chunk*.

The algorithm we use to build the chunks and Bloom Filters is described in the section *Building the Index and Bloom Filters*. In this section, we describe how to read the Bloom filter files and document the files' format.

Note that each Bloom filter (as well as each chunk) is independent of all the others. This means that querying against the Bloom filters (and the chunks) may proceed concurrently. In the code below, we've removed concurrency in order to make the discussion easier to understand.

We will start by describing the algorithm used to read the Bloom filter. Following that, we describe the file format.

```
// Optimize address queries by consulting a Bloom filter for set membership of
// an address prior to reading the much larger Index Chunk.
func GetAppearancesUsingBlooms(addrs []address) (apps [][]AppearanceRecord) {

    // Get a list of all Bloom filters
    listOfBloomFilters := <list of bloom filter files>

    // For each Bloom filter...
    for _, bloom := range listOfBloomFilters {

        // Prepare a place to store a list of hits
        hits := make([]address)

        // For each address...
        for _, addr := range addrs {
```

```
            // ...if the address hits the Bloom filter...
            if bloom.IsElementOf(addr) {
                // ...take note...
                hits = append(hits, addr)
            }
        }

        // Did any addresses hit on this Bloom filter?
        if len(hits) > 0 {
            // Yes. We've hit at least one bloom. This indicates that the
            // address "may be" in the chunk. We must explicitly check...

            // ...make sure the chunk is local (i.e., download from IPFS if not)
            establishChunk(bloom.Filename)

            // For each address, get that address's appearances (if any)
            for _, addr := range hits {
                apps = append(apps, GetAppearances(addr, bloom.FileName)
            }
        }
    }

    // Return the results
    return apps
}
```

establishChunk consults the local hard drive for the *chunk*. If the *chunk* is not present (because the user used `chifra init`, for example), the routine consults the Manifest in order to decide which IPFS hash to download. The full binary *chunk* is then downloaded from IPFS.

TrueBlocks uses a multi-part Bloom filter called an *Adaptive Bloom Filter*. The "parts" of the multi-part filter are bit arrays encoding the set membership of addresses in the chunk. Each bit array is 1/8 of one megabyte wide (an arbitrarily chosen width balancing file size with the filter's effectiveness). The method by which we insert addresses into the Bloom filters is described in the section called *Purposeful Sloppiness* below.

The word "Adaptive" here refers to the fact that we grow the Bloom filter as the set membership for each element is inserted, making the filter as large as it needs to be in order to maintain a pre-defined "maximum expected false positive rate." Different block ranges contain a varying number of addresses. If we did not use an adaptive Bloom filter, the Bloom filters would either be under-populated or over-saturated—in fact, this is exactly what happens in the Ethereum node software where the Bloom filters are ineffective.

We "grow" the Adaptive Bloom filter by adding additional bit arrays (parts) as needed to ensure a consistent false positive rate. Consult the source code for further information.

As the expected number of false positives becomes larger due to inserting additional addresses and finally overtops the pre-determined false-positive rate, we add an additional bit array to the Bloom. In this way, the expected false positive rate of the Bloom never exceeds the pre-determined value. We do this in an effort to ensure that each address is as likely to hit an arbitrary Bloom filter as any other. This promotes a more even distribution of the downloaded chunks.

Bloom filter files have a header similar to the one noted above for the index portions, with the exception that the *magic number* field is only 16-bits wide and there are no counters. It does have the same keccek256 of the version string.

A Golang version of the Bloom filter data structure is presented here:

```
// The structure of one Bloom filter
type BloomFilter struct {

    // The number of bit arrays in this file
    Count       uint32

    // An array of bit arrays, the first Count-1 of which are "full"
    BitArrays   []BitArray

}
```

The *Count* field at the head of the file records the number of BitArrays in the file. A single BitArray structure look like this.

```
// An arbitrarily chosen width of each bit array — 1/8 of a megabyte
type BitArrayWidth ((1024 * 1024) / 8)

// A Bloom filter consists of one or more BitArrays. A BitArray stores the
// actual bit-by-bit array as well as an unsigned 32-bit integer (nInserted)
// used for data analysis and debugging. The actual bit array is BitArrayWidth bytes
// wide.
type BitArray struct {

    // The number of addresses inserted into Bits
    nInserted   uint32

    // The bit-by-bit array representing the set membership in this chunk
    Bits        [BitArrayWidth]byte
}
```

A careful reader will notice a number of things:

1) There is no versioning information in the Bloom file. This is by design so as to keep the file as small as possible. Versioning can be found in the associated *chunk*.

2) The *Count* field is redundant as the BitArray structure is fixed width. One could have deduced the *Count* value from the file's size divided by the record size.

3) The *nInserted* field increases the size of the BitArray and serves no direct purpose other than debugging / data analysis. It could have been removed and will be in the future.

The *IsElementOf* function now becomes easier to understand:

```
// Consult the Bloom filter for set membership of an address.
func (bloom *BloomFilter) IsElementOf(addr address) bool {

    // Convert the address into a bit array using the same function used to
    // convert addresses into bit arrays during construction
    bitsLit := addressToBitArray(addr)

    for _, bitArray := range bloom.BitArrays {
        // If all bits are lit, the address "may be" a member
        if (bitArray & bitsLit) {
            return true
        }
    }

    // The address is not present in the chunk
    return false
}
```

The construction of the Bloom filter and index chunks remains to be described. This is done in the section below called *Building the Index Chunk and Bloom Filter*.

Summary

In summary, the algorithm used to query the Unchained Index for a list of appearances for a given address is:

for each address of interest
  for each bloom filter…
    consult the bloom filter to see if the chunk may *contain the address*
    if the chunk may *contain the address,*
      download the chunk if it's not already present
      extract the appearances of this address from the chunk (if any)

In the next section, we describe a few ancillary databases used by the TrueBlocks system. Technically these databases are not required for the operation of the Unchained Index, but are published to the Unchained Index smart contract, and they greatly improve the performance of the system, so we describe them next.

The Names Database File

The names database is not technically part of the Unchained Index, but it is useful. Periodically, we publish the IPFS hash to the Unchained Index smart contract. It is very useful when presenting information about the history of an address.

Publishing the *namesDB* hash to the smart contract allows our end-user software to acquire the names database without relying on a third party. This lowers our cost of publication. The names database is a binary file consisting of nearly 14,000 fixed-width records. The number of records in the file is easily calculated from the file's size.

The binary format of the names database follows. (**Note:** In version 2.0.0 of the system and prior, the names database is not published to the Unchained Index. It is stored in the project's GitHub repo. This location and format of this file may change in the future.)

```
// An array of fixed-width name records
namesDb := []NameRecord

// The number of records can be calculated using the file's size
nRecords := fileSize(<path to names database>) / sizeof(NameRecord)
```

Each *NameRecord* takes on the following fixed-width format:

```
type NameRecord struct {
    // A user-defined tag
    Tags            [31]byte

    // The address to which this name resolves
    Address         [43]byte

    // A name or label for the address
    Name            [121]byte

    // The symbol (if any) for the token (automatically assigned if found on-chain)
    Symbol          [31]byte

    // An attempt to record where the name was first acquired
    Source          [181]byte

    // An arbitrary description of the address, including perhaps a URL
    Description     [256]byte

    // For ERC-20 tokens, the decimals for the token
    Decimals        uint16

    // An internal, opaque bit array used for flags including deletion status
    Flags           uint16
}
```

The Timestamp Database File

The IPFS hash of a second ancillary data is published to the Unchained Index smart contract. We call this database the Timestamps Database. For each chain, this database is published at "chainName-ts". For example, mainnet-ts is the database name for the Ethereum Mainnet's timestamp database.

The need for a timestamp database becomes apparent as soon as one tries to query a block by date (or, equivalently timestamp). The RPC does not include such queries resulting in the need to scan the chain for the result.

An external timestamp database speeds up that query significantly. This Timestamps Database is created during the process of building the index, and its location is published to the smart contract as mentioned above.

The structure of the binary timestamps database is very simple:

```
// An array of fixed-width timestamp records
timestampDb := []TimestampRecord

// The number of records can be calculated using the file's size
nRecords := fileSize(<path to timestamps database>) / sizeof(TimestampRecord)
```

A single TimestampRecord structure looks like this:

```
type TimestampRecord struct {

    // The block number for this timestamp
    BlockNumber   uint32

    // The timestamp at that block
    Timestamp     uint32
}
```

The following invariant is true for every record in this database:

```
bn == timestampDb[bn].BlockNumber
```

Finding a timestamp given a block is accomplished as follows:

```
ts := timestampDb[bn].Timestamp
```

Finding a block number given a timestamp is accomplished using a binary search:

```
bn := binarySearch(&ts, timestampDb, nRecords, sizeof(TimestampRecord))
```

Implementation Note: We note that the Timestamps Database could have been half as big if we had removed the block number from the file. The block numbers are sequential, start with zero, and contain no gaps—a self-contained index. We choose, however, to include block numbers in the database as an aid for debugging and checking the database's integrity. We reserve the right to change the format of this file in future versions.

Conclusion

This ends the section of the paper describing file formats. Next, we describe how the system builds each of the binary files it produces.

Building the Index and Bloom Filters

Above, we defined the various binary file formats of the databases that make up the Unchained Index and a number of algorithms used to read and query the index.

In this section, we describe the algorithms used to create and publish the index itself. We start with a discussion one of the underlying data structures in the system: the *AppearanceRecord*. We then proceed to discuss how we extract appearances from the raw blockchain data. We conclude with a discussion about the engineering decisions we made in order to preserve immutability, reproducibility, and content-addressability as we discussed at the start of this document.

Definition: Address Appearances

We define an *Appearance* to mean any location in the history of a blockchain where an address "appears." More specifically, an *appearance* is a <*blockNumbzer.transactionId*> pair.

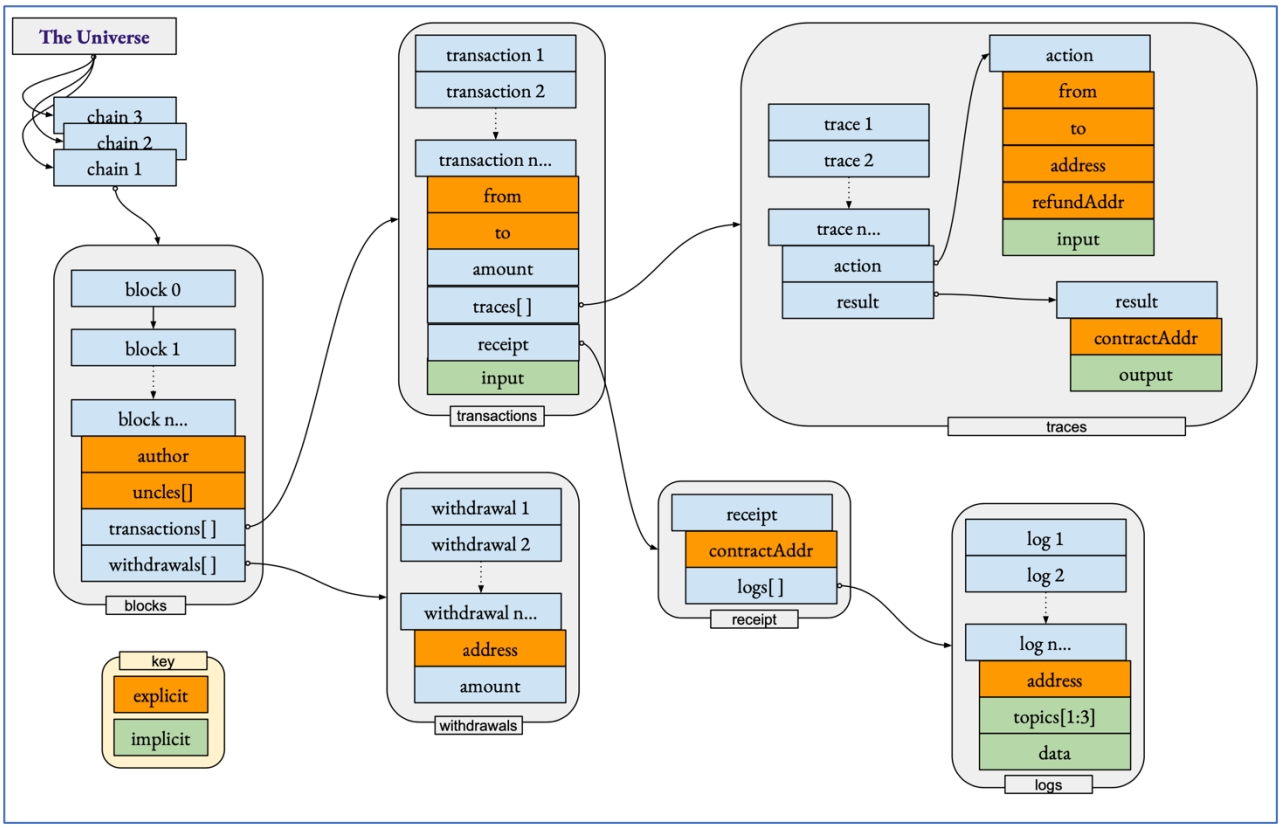
The algorithm we use to identify appearances is "purposefully sloppy" (for reasons similar to why bloom filters permit false positives). We wish to include byte patterns that "appear" to be addresses but may not be, rather than not include obscure addresses that are legitimate.

Unfortunately—and very importantly—it is not even theoretically possible to find every address appearance. Every array of 20 bytes is a valid Ethereum address. Desiring to maintain an index of a manageable size, we were led to the idea of "baddress" which we described below.

On the following page is a graphic showing a typical blockchain's data structure (if there is such a thing). We've highlighted places where an address may "appear." Some of these locations are obvious, such as the *from* and *to* address of transaction. We call these appearances "explicit" and they are colored orange in the following graphic.

Other appearances are not so obvious, such as those found in the input data of a transaction, a log topic, or a trace's output data. We call these appearances "implicit" and color them green in the following graphic.

We discuss this distinction between implicit and explicit appearances in the section called "*Purposeful Sloppiness*."



The orange items shown above are what we would call *explicit* appearances. These locations are well-defined in the documentation of the RPC as being addresses. Appearances of this type are trivial to find. We add them directly to our growing index as the scraper scans the chain.

There are other locations in the data where address appear. Some of them are "semi-obvious", such as log topics for certain events. Many indexers (including one's we've tested) find these appearances. For example, the ERC20 token *Transfer* and *Approve* events contain addresses in well-known locations. These appearances are also easily extracted.

However, we find the "ad-hoc" method of identifying these appearances in-effective. It requires specific knowledge of the byte-stream. One must know where to look making automation impossible. For well-established standards, such as ERC20, perhaps this is okay, but for arbitrary smart contracts it's not scalable. As new protocols appear, this type of specific knowledge about the data is best avoided.

Even if it were possible to maintain specific knowledge about the data (it's not), doing so precludes recognizing appearances in the long tail of unknown, obscure, or new protocols. (This goes against our desire to be more inclusive rather than less.)

Looking more closely at the "implicit" appearances mentioned above (those colored green), one may finally understand the true reason why producing complete and accurate transactional histories for EVM chains is difficult.

In these green areas of the data, address appearances are obscured. The appearances are hidden inside the byte data. Is it still possible to find them?

Our desire to capture as many appearances as possible forces us to operate with minimal "special knowledge" about the byte stream. *It is not even theoretically possible* to identify every appearance in the general case. Our indexer, as odd as this may seem, takes advantage of certain aspects of the byte stream[1] that help us to be as inclusive as possible:

1) The Solidity compiler zero-left-pads many values to 32 bytes, including addresses, numerals, and even Booleans,

2) Ethereum addresses are 20 bytes long, meaning those that are left-padded with zeros start with 12 leading zeros in the byte stream, and

3) Almost all numeric values encountered in the byte stream are smaller than (that is, they have more leading zeros, almost all addresses).

(The third point above hints at the idea of a "baddress" described below.)

We remarked earlier that these observations were "odd," and they are. We acknowledge that, but we desired an algorithm that was as inclusive as possible. Yes. Using the above observations in our indexing algorithms will include bytes that are not in reality addresses. Yes. The algorithm will miss some addresses. But when one operates in a world where it is not even theoretically possible to do a perfect job, one's algorithms must be *purposefully sloppy*.

The upshot is that for *implicit* locations in the byte stream, our algorithm is forced to use certain heuristics. We explain these heuristics next.

Notes on Per Block Data

Performance note: We note that appearances buried in a given canonicalized block are independent of other appearance found in other blocks. In the following discussion, we describe how the algorithm processes a single block. The reader is encouraged to consider the obvious opportunities to make this processing concurrent. (Our actual code is heavily concurrent.) Concurrency considerations have been removed in the following description to make things easier to understand.

The processing described in this section of the paper is at the heart of the system. The algorithms require per-block processing in order to build chunks. Riding on top of the per-block processing is a process called the *Consolidation Loop*, which is described in the section following this section on per-block processing.

---

[1] Yes – this seemingly contradicts our previous comment about avoiding "special knowledge" of the byte stream. We justify this by noting that this second instance of "outside knowledge" the knowledge is not about a particular smart contract protocol such as ERC20—the "knowledge" we take advantage of here related to the way the Ethereum blockchain itself operates. (That is, the fact that addresses are frequently left padded to 32-bytes by Solidity.) For this reason, this method works across any protocol and thereby avoids destroying the long tail.

Another performance note: These algorithms are very aggressive in the way they query RPC endpoints. The algorithms work against any RPC (either local or remote), but they are significantly more effective when running against a locally running node. This fact is because of three reasons:

1) Most RPC providers do not support delivering the RPC's *trace_* namespace. Of those that do, the cost is prohibitive;

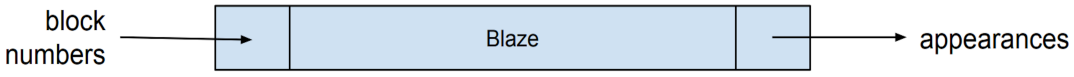
2) If one uses a remote RPC provider, one will be rate-limited. The algorithms described in this section are very aggressive;

3) Why do we have to pay for free data?

We highly recommend running your own node. We find Erigon and Reth to be quite usable and reliable. Both synchronize quickly to the head of the chain, both are fast at delivering RPC requests locally, both support the *trace_* namespace, and both take up much less hard drive space than the alternatives.

Extracting Addresses Per Transaction Per Block

The algorithm we use to process (or scrape) blocks is a pipeline called Blaze. At the highest level, Blaze "shoves" *blockNumbers* into the head of the pipe and, as if by magic, *appearances* emerge from the other end of the pipe.



Blaze consists of two distinct internal channels. The first channel:

a) ingests blockNumbers
b) queries the RPC for that block's data, and
c) produces a stream of "traces, logs, and withdrawals."

The second channel considers each byte of the block data (using the heuristics mentioned above), searching for appearances.

The overall effect of the pipeline is that block numbers go in end and address appearances come out the other.

Detailing now the actual code, we see that Blaze begins by creating two channels:

```
// Blaze - process a range of blocks to extract address appearances
func Blaze(br BlockRange) []AppearanceRecord {

    // Create a pipeline to process block numbers and query the node.
    blockChannel := make(chan uint64)

    // Create a second pipeline to process the retrieved data which is stored
    // in a structure called ScrapedData which contains the block number, the
    // block's timestamp and the Log, Trace, and Withdrawal data for the block.
    addressChannel := make(chan ScrapedData)
```

Next, Blaze creates as many go routines as the user instructs it to create in order to process the collected data. Different blockchains have different characteristics, therefore, the software allows the end-user to configure the process from the command line.

```
    // Set a number of processor pipelines
    var blockWg sync.WaitGroup
    blockWg.Add(opts.nBlockProcessors)
    for i := 0 ; i < opts.nBlockProcessors ; i++ {
        go processBlocks(blockChannel, addressChannel, &blockWg)
    }

    // Set a number of address processor pipelines
    var addrWg sync.WaitGroup
    addrWg.Add(opts.nAddressProcessors)
    for i := 0 ; i < opts.nAddressProcessors ; i++ {
        go processAddresses(addressChannel, &addrWg)
    }
```

Now that the pipelines are set up, the next step is to insert block numbers into the top of the pipeline. The caller provides the *start* and *stop* blocks for this round.

```
    // Shove the blocks into the block pipeline
    for bn := opts.Start ; bn < (opts.Start + opts.Count) ; bn++ {
        blockChannel <- bn
    }
```

And finally, Blaze closes the channels (flushing them) and waits until the pipeline finishes processing.

During processing, Blaze writes individual files containing the *AppearanceRecords* found in each block. The *Consolidation Loop* (described below) post-processes this data upon return.

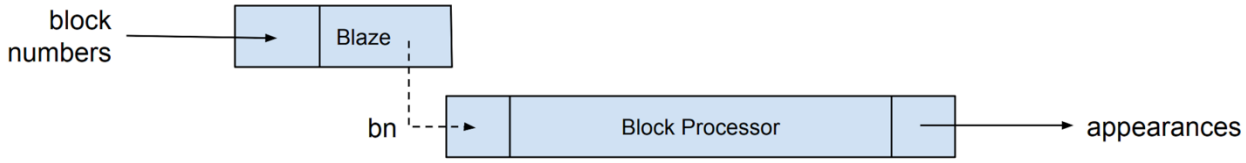
```
    // When we're finished shoving blocks, close the channels and wait for
    // them to finish processing.
    blockChannel.Close()
    blockWg.Wait()

    addressChannel.Close()
    addrWg.Wait()

    // return to caller
    return nil
}
```

Blaze process takes as input a block range. It is the responsibility of the caller to determine which blocks to process. During regular processing, the caller of Blaze is the *Consolidation Loop* which is described below.

The output of Blaze is written to a file inside the address pipeline—that is, the algorithm "streams" the results choosing not to return the data in an array. This allows us to process a very large number of blocks at a time (including a single block—a feature we use for debugging). This "steaming" aspect to the code is one of the primary reasons that the algorithm continues to work on small machines.

The Block Processor

Looking now inside the first pipeline, we see the block processor (the *processBlocks* routine). The block processor queries the node for trace, log and withdrawal data given a block number. It then collates that data into a more useful structure (called ScrapedData) and inserts this collated data into the address processor (the *processAddresses* routine).



The block processor ranges over the block channel, waiting until block numbers appears.

Each time a block number appears, the block processor makes three RPC requests for data and combines the result into a *ScrapedData* structure. It then shoves that combined data into the address processor. When the *blockChannel* is closed, the loop exits and the *WaitGroup* is marked as done. Blaze then returns control to the caller.

```
// Wait until block numbers appear in the block channel and then query
// the node for the block's timestamp, its logs, its trace data, and its
// withdrawls. Combine the results into a structure for easier processing
// and pass the combined result into the address processor channel.
func processBlocks(blockChannel chan int, addressChannel chan ScrapedData,
                   blockWG *sync.WaitGroup) {

    // Range over the block channel. Note that these values will be
    // non-sequential and may be processed concurrently.
    for bn := range blockChannel {

        // Query the node and combine results for further processing
        combined := ScrapedData{
            BlockNumber: bn
            TimeStamp: client.GetTimestamp(bn)
            Withdrawal: client.GetWithdrawals(bn)
            Traces: client.GetTraces(bn)
            Logs: client.GetLogs(bn)
        }

        // pass it on to the next piece of the pipeline...
        addressChannel <- combined
    }
}
```

Implementation Notes:

1) This code is concurrent-aware. Any number of go routines may process this data concurrently and independently of other blocks;

2) In future implementations, we intend to process groups of blocks at a time as opposed to individual blocks. This would allow us, for example, to use **eth_getLogs** with a block range instead of a single block. This would be much faster. Processing multiple blocks at a time would require us to insert block ranges as opposed to block numbers into the block processor channel.

3) **Possible EIP:** It would be beneficial if the node software provided timestamp information with the **trace_block** response. This would eliminate one third of all queries against the node.

4) **Possible EIP:** It would be very beneficial to indexing if the node software provided the above routine as part of the RPC. A single call such as *eth_getAppearancesInBlock* would solve the problem for all indexers.

Next, we detail the ScrapedData structure (called "combined" in the above code). This structure combines the received information from the RPC commands (*trace_block*, *eth_getLogsByNumber*, and *eth_getBlockByNumber*). This convenience structure makes subsequent processing easier.

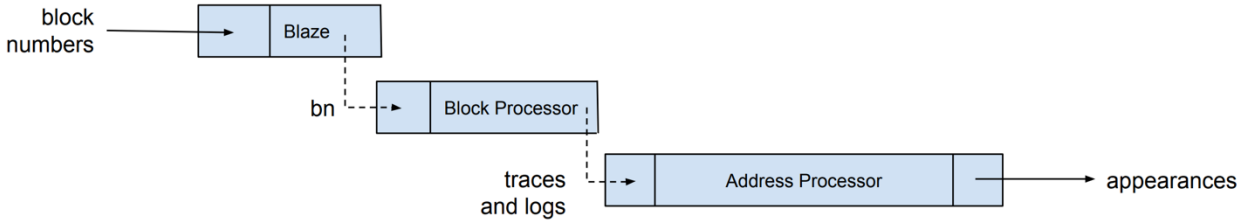
```
// The ScrapedData structure
type ScrapedData struct {

    // The block number for the data stored in this structure
    BlockNumber   uint32

    // The timestamp at that block
    Timestamp     uint32

    // The withdrawal data as returned by the RPC's eth_getBlockByNumber routine
    Withdrawals   []Withdrawal

    // The trace data as returned by the RPC's trace_block routine
    Traces        []Trace

    // The log data as returned by the RPC's eth_getLogs routine
    Logs          []Log
}
```

The data types *Withdrawal*, *Trace*, and *Log* are defined in the Ethereum RPC documentation.

The Address Processor

Looking more deeply into the pipeline, we see that the *block processor* passes the above data into the *address processor* which we describe next.

Conceptually, this step in the pipeline looks like this:



The address processor accepts *ScrapedData* structures as they are inserted into the *addressChannel* and finds "every appearance of any address anywhere in the byte stream."

For each identified address, three values are written to disc for later processing by the *Consolidation Loop*. These three values are a 20-byte address, a 32-bit integer representing the *blockNumber*, and a 32-bit integer representing the *transactionIndex*. In other words, an *AppearanceRecord*.

When the channel is closed, the wait group is marked as being completed.

```
// The Address processor
func processAddresses(addressChannel chan ScrapedData, addrWg *sync.WaitGroup) {

    // range across the scraped data in the channel
    for sd := range addressChannel { // sd means ScrapeData

        // provide a map to store unique addresses used when building
        // the Index Chunk
        addressMap := make(map[AppearanceRecord]true)

        // extract address appearances from the withdrawal data, store to map
        extractFromWithdrawals(sd.blockNumber, sd.Withdrawals, addressMap)

        // extract address appearances from the log data, store in the map
        extractFromLogs(sd.blockNumber, sd.Logs, addressMap)

        // extract address appearances from the trace data, store in the map
        extractFromTraces(sd.blockNumber, sd.Traces, addressMap)

        // write the address map to the output medium
        writeAddresses(sd.blockNumber, addressMap)
    }
}
```

We describe the particular processing of withdrawal, trace, and log data below.

eth_AddressesPerBlock

Possible EIP: The extraction of address appearances from a given block would make an obviously beneficial EIP that would benefit the entire space. If the node software itself had a routine called *eth_AppearancesPerBlock*, anyone could easily build a full index of every appearance of every address anywhere on the chain. We encourage anyone with the wherewithal to write such an EIP to do so.

Extracting Addresses from Withdrawals

Extracting addresses from the Withdrawals data is a simple process given the simple nature of the withdrawals data in each block. Since the advent of the Shanghai fork, each block contains an array of *withdrawals*. Each withdrawal has a explicit data item called *address* which is easily extracted.

Extracting Addresses from Traces

The trace data is more complicated. The data returned by the RPC's *trace_block* function contains a field called *type*. The *type* field distinguishes between various types of trace data.

For example, traces related to smart contracts calling into each other, traces related to self-destructs and contract creations, traces related to mining rewards are all types of traces. The address processor uses *type* to figure out where to look for *explicit* address locations, which, as we noted above, are easy to find.

The address processor then proceeds to process locations in the byte stream where *implicit* address appearances are found.

In our Go code, we define the various types traces can take on:

```
// Possible values for a trace's Type field
type TraceType uint8

const (
    // Generated when one smart contract calls another
    Call TraceType = iota

    // Generated for mining rewards (there are three subtypes of this
    // type with values of "block", "uncle", and "external")
    Reward

    // This trace type is generated during a self-destruct transaction
    SelfDestruct
```

```
    // This trace is generated during the creation of a smart contract
    // (CreateFailed is identified if the deployment fails but the tx does not.)
    Create

    // Sometimes, the trace data contains incorrectly formatted data (for
    // example a miner reward given to address 0x0. While this is clearly
    // possible, we don't want to identify these values as appearances.
    // We use the following trace type to identify these situations.
    Misconfigured
)
```

The differing trace types carry information that contains addresses appearances in differing locations. In some cases, those locations are *explicit*—in the sense that these values are stored in locations that are explicitly documented as being addresses. These addresses are "easily-found" (orange in the above image).

For other trace types, such as smart contract interactions, addresses may appear *implicitly* in raw byte data such as the *input* and *output* fields of a trace. When an *explicit* address appearance is found, we easily add it to the growing appearance map. When the addresses appearance is *implicit*, we must be creative—using heuristics to find appearances as we've described in the section below called *Purposefully Sloppy*.

The Trace data structure is made up of two sub-structures called Trace.Action and Trace.Result (see the RPC documentation). We summarize the various trace types, address locations, and *explicit* / *implicit* aspect of the trace in this table.

Type	Location	Aspect
Call	Action.From	*explicit*
	Action.To	*explicit*
Reward		
Block	Action.Author	*explicit*
Uncle	Action.Author	*explicit*
External (Gnosis?)	Action.Author	*explicit*
SelfDestruct	Action.Address	*explicit*
	Action.Refund	*explicit*
Create	Action.From	*explicit*
	Result.Address	*explicit*
CreateFailed	Receipt.ContractAddr	*explicit*
Create	Input (contract byte data)	*implicit*

Type	Location	Aspect
Any	Input (function call data)	*implicit*
Any	Output (function output data)	*implicit*

The address appearance extraction carried out for the rows in the table labeled *implicit* are described in the section called *Purposeful Sloppiness*. Because the log data contains a similar distinction between *explicit* appearance locations and *implicit*, we explain the processing of the Log data first.

Extracting Addresses from Logs

Ethereum's Log data is far simpler than its Trace data.

Ignoring the fact that the log data contains smart contract-specific information, which we want to avoid using for the reasons mentioned above concerning the long tail, every Ethereum Log has the same format.

Every Log has an *address* field which is the address of the smart contract that generated the log. Every log has between one and four 32-byte *topics*. The first topic never contains addresses—it is the hash of the generated event signature.

The remaining three topics may contain address appearance but always in an *implicit* way, never *explicitly*. The remaining field, *data*, contains an arbitrarily long byte stream (whose length is usually modulo 32 bytes long but may be any length). This *data* field may contain *implicit* appearances. It never contains *explicit* appearances.

Location	Aspect
Address	*explicit*
Topic[0]	never contains addresses
Topic[1:3]	*implicit*
Data	*implicit*

For the *explicit* appearance in the *Address* field, we add it to the map. For the remaining *implicit* fields, like the *implicit* fields in Traces, we can now explain the processing necessary to extract appearances from this more complicated data.

Purposeful Sloppiness

Did you ever wonder why it's nearly impossible to do accurate off-chain accounting on the Ethereum blockchain? Wouldn't it make sense that an 18-decimal place accurate ledger that comes to perfect agreement every 12 seconds on hundreds of millions of address, should be able to be accounted for automatically? Shouldn't tax reporting be a push button?

We (at TrueBlocks) know why off-chain accounting doesn't work—people are not extracting address appearances at *implicit* locations.

That's not quite true—some people do—but they do so by using ABI files which are sparsely available at best. This is the long tail problem we mention above. The use of specific ABI files for specifically chosen "popular" smart contracts is a mistake in our opinion for a number of reasons:

1) Who's to choose which smart contracts are popular and should be included;

2) Over time adding new smart contracts to the list becomes more and more onerous as many, many systems depending on this "list of known addresses" must be maintained;

3) How is maintaining such a list done fairly and how does the entire ecosystem resist the inevitability of this list being "captured" and made "pay-to-play".

Hint: Keeping a list of popular smart contracts does not work.

In this section, we discuss the method by which we avoid creating a list of known smart contracts in aiding us extracting address appearances. A few notes before we start:

1) A perfect algorithm is not possible. Every 20-byte string is a valid Ethereum address. For example, the value "true", as a 20-byte string is 0x0000…0001. 0x0000…0001 is also a valid Ethereum address—albeit an unusual one (and an reserved one, but this can be ignored). Likewise, the value of -1, when represented as a 20-byte string (0xfff.ffff) is also a valid address.

2) In order to avoid relying on external information (and therefore information that can be censored and perhaps captured), we desire an algorithm that relies on only locally available information—that is information that is directly in the byte stream.

3) It is possible (and quite effective) to do this if one has a few heuristics at hand.

4) It is better to be over-inclusive than under-inclusive. While it is not possible to create a perfect algorithm, one can get arbitrarily close if one is willing to trade off the size of the index on disc. A smaller index will include less appearances. A more inclusive index will be larger. We've chosen to be rather more inclusive than less.

5) Being more inclusive allows the Unchained Index to reconcile about 5% more transactions than any other system we know of. This is enough to reconcile nearly every address we've tested. The results of these tests will be presented in a separate report.

Heuristics for Identifying Appearances

Knowing that a perfect solution is impossible, what heuristics might one use to accomplish the desired task? (Reminder: our task is to find every appearance of every address in order to reconcile every address's accounting off-chain.)

Below is a list of considerations and heuristics we use to identify appearance in those locations in the Ethereum data we've labeled *implicit* above.

1) A large part of the Ethereum byte stream is aligned to 32-byte boundaries. This happens for various reasons (including the compiler) and becomes quite handy;

2) Addresses, at least presently, are 20 bytes long. Because the byte data is frequently aligned on 32-byte boundaries and much of the data is left padded with zeros, this means an address has at least 12 leading zeros. Any 32-byte chunk of *implicit* data that does not start with 12 zeros is considered not to be an address appearance. Frequently these are strings or contract byte-code.

3) Much of the Ethereum byte stream that contains numbers is four eight-byte integers packed into a 32-byte string or a single eight-byte integer packed into 32 bytes.

4) Boolean values are (oddly) packed on their left with 31 zeros.

5) Addresses less than 0x00000….ffff (65,5350) is reserved for pre-compiles.

We use these observations to produce an admittedly odd but quite effective algorithm for identifying address appearances. In a forthcoming paper[2], we discuss the results of using these methods when compared to other sources of Ethereum data, such as popular web-based APIs, including Covalent and EtherScan.

Hint: TrueBlocks identifies about 15% more appearances than either of these methods and of those, about 5% are material. Material means that one or more asset balance changes. This is why other methods find off-chain reconciliation so difficult. They are missing transactions. As you might surmise—it's not possible to reconcile an accounting if one is missing transactions.

While running the two routines detailed above (*extractFromTraces* and *extractFromLogs*) our processing may encounter many *implicit* locations for an address appearance. For *explicit* locations, we simply add the address to the growing appearance map. For *implicit* locations (all of which are 32-bytes wide), we pass the 32-bytes value to a routine called *isPotentialAddress*.

That routine is detailed next:

[2] "How Accurate is TrueBlocks" to be published on our blog: https://trueblocks.io/blog.

```
func potentialAddress(addr string) bool {

    // Any 32-byte value smaller than this number (including precompiles)
    // are assumed to be baddresses. While there are technically a very
    // large number of addresses in this range, we choose to eliminate them
    // in an effort to keep the index small.
    //
    // While this may seem drastic—that a lot of addresses are being excluded,
    // the number is actually a quite small number--less than two out of
    // every 1000000000000000000000000000000000000000000000000 20-bytes strings
    // are excluded, and almost every one of these are actually numbers such
    // account balance or number of tokens transferred. It's worth it.
    small := "0000000000000000000000000000000000000000ffffffffffffffffffffffff"
    //        -------+-------+-------+-------+-------+-------+-------+-------+
    if addr <= small {
        return false
    }

    // Any 32-byte value with less than this many leading zeros assumed to be
    // a baddress. (Most addresses are 20-bytes long and left-padded with zeros
    // Note: we're processing these as strings, so 24 characters is 12 bytes.
    largePrefix := "000000000000000000000000"
    //              -------+-------+-------+
    if !strings.HasPrefix(address, largePrefix) {
        return false
    }

    // A large number of what would normally be considered valid addresses
    // happen to end with eight zeros. We're not sure why, but we identify
    // these as badresses as well in a final effort to lower the size of
    // the index. We've seen no obvious ill-effects from this choice.
    if strings.HasSuffix(address, "00000000") {
        return false
    }

    return true
}
```

As we noted, this is an admittedly odd algorithm. Yes—it uses a seemingly arbitrary set of heuristics to identify addresses. No—it's not future-proof given that the packing algorithms that produce this data may change—unlikely, but they may evolve. We acknowledge these shortcomings but re-iterate that this method is "purposefully sloppy." The task is impossible—even theoretically—but this should not stop us from producing useful data—data that can be used to reconciled accounts off-chain.

Instead of arguing against the efficacy of this algorithm, we should be exploring ways to make these *implicit* address appearance—of which this algorithm identifies a massive number—unnecessary. The problem stems from a lack of recognition by the core developers of the Ethereum nodes that the problem of identifying appearances even exists. This document may serve to start the process of identifying the problem.

Baddresses

We mentioned above the idea of a "baddress," which we briefly define here. A "baddress" is any 20-byte string that could potentially be a valid address but which we choose to exclude from our index. Note that every 20-byte string is a valid address.

We identify *badresses* in consideration of the size of the index on disk. We want to be inclusive enough to find the great majority of real addresses but not so inclusive as to explode the size of the index on disc.

Therefore, a "baddress" as any 20-byte string that is less than "`0x0000000000000000000000000000000000000000ffffffffffffffffffffffff`" or ends with eight zeros ("`00000000`"). This is an arbitrarily selected definition.

Non-Transaction Transactions

TrueBlocks's primary use case is off-chain, historical state reconciliation (including accounting). For this reason, we add certain records to the index that are not properly transactions in the normal EVM transaction sense.

For example, the block reward paid at the end of each block is not a regular Ethereum transaction. Neither is the uncle reward nor the nephew reward nor withdrawals. However, these "pseudo-transactions" alter account balances. Wishing to account for such non-transaction transactions properly, we add pseudo appearances to the index.

We arbitrarily assign the following special-case *transactionIndex* values for these eight different types of transactions. This allows us to process them more easily during export.

Type	Value	Notes
BlockReward	*bn.99999*	
UncleReward	*bn.99998*	
MisconfigReward	*bn.99997*	In some early blocks, it was possible to assign the zero address for block reward recipients.
ExternalReward	*bn.99996*	Seen only on Gnosis chain
Withdrawal	*bn.99995*	
NephewReward	*bn.99994*	
TxFeeReward	*bn.99993*	
EsInternalTx	*bn.99992*	EtherScan internal txs. *chifra slurp* only.

The Consolidation Loop

This section discusses the consolidation phase of the algorithm, that is, how the system determines when to create a new index chunk, how we best choose differing configuration values for different chains, and how the algorithm creates our enhanced, adaptive Bloom filters.

As a very meager first pass at explaining this important process, we can say that it is a continually running process that periodically wakes up and processes any new blocks that may have appeared since the last time it ran.

During the initial phase of scraping a new chain, the process runs continually with no delay between runs. Once caught up to the head of the chain, the process sleeps as long as the block time for the chain so as to place as small a burden on the machine as possible.

The process remembers, each time it runs, where it last left off. In this way, the algorithm can pick up where it left off the next time it starts. The process calls Blaze with the pre-configured (or user-supplied) number of blocks to process. Once Blaze is finished, the Consolidation Loop decides if it's time to create a chunk. This decision is different for each chain. On mainnet, we've chosen to use a certain number of appearance records (2,000,000) as the break-off point for a chunk (not forgetting the snap-to-grid feature mentioned below). This produces a new chunk about twice a day on Mainnet.

On the Gnosis chain, 2,000,000 records would take many days to produce (due to its much lower usage). On some private chains, 2,000,000 appearances may never happen. Each chain must be evaluated separately.

A decent approximation of how often to create a chunk is to divide the number of seconds in a day by the block time and create a chunk after that many blocks. Be careful, though—this does not always work. During the 2016 dDos attack on the Ethereum mainnet, millions of appearances were created in very few blocks.

The consolidation loop is described on the next page.

```
func ConsolidationLoop(start, end uint64, outputLocation string, pin bool) {

    // Run forever...
    for (true) {

        // Call Blaze to process a range of blocks. Blaze will write the results
        // into a “staging” folder if the blocks are more than six blocks old
        // and those blocks will no longer be consulted. If the blocks are
        // less than six blocks old, Blaze writes them into an “unripe” folder
        // where they may be used, but the user is cautioned that the blocks
        // blocks may re-organize.
        Blaze(start, end, outputLocation)

        // Check to see if one of two conditions are true:
        // 1) enough (2,000,000) appearances have been written
        // 2) check to see if we’ve crossed the snap-to marker
        if <enough records are present> or <snap-to-grid> {

            // write the Chunk to the hard drive, produce the Bloom filter
            // and optionally pin the chunk and the Bloom filter to IPFS,
            // update the Manifest and pin that as well.
            writeChunk(outputLocation, pin)

            // Remove written appearance records from the stage
            removeWrittenRecords()

            // Pin chunks and Bloom filters, update Manifest, pin
            // Manifest and optionally push IPFS of Manifest to smart contract
            pinAndPublishManifest()

        }
    }
}
```

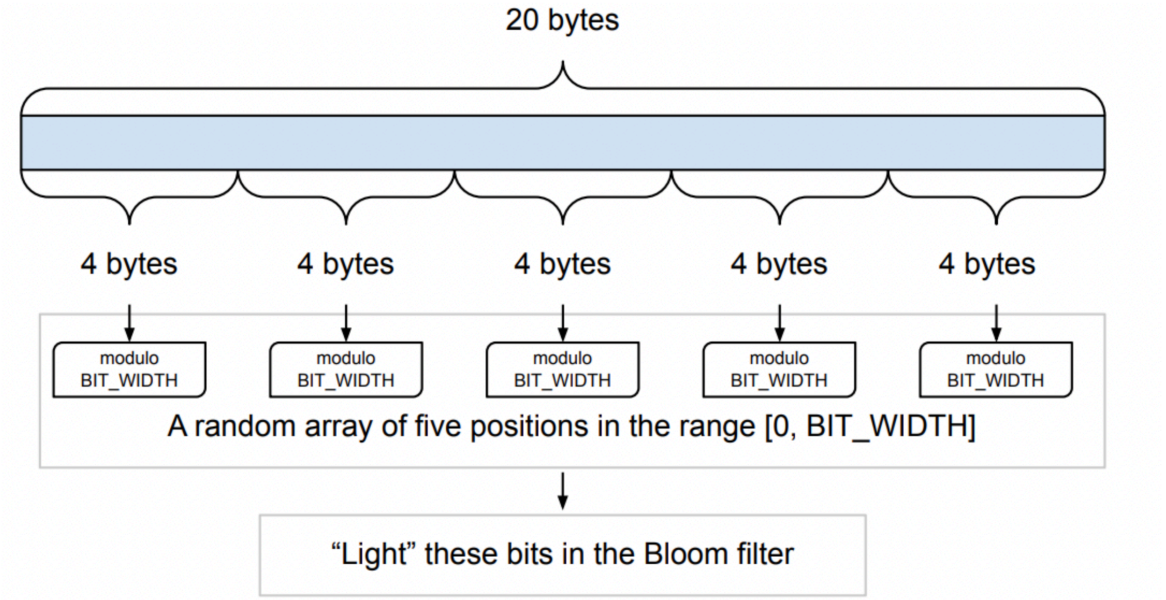
“Enough records are present” and “snap-to-grid” are chain dependent and are left to be described elsewhere.

The writing of the data to a chunk is carried out in a way opposite to that described above for reading. In essence, open the file, write the header, write the address table, and finally write all the appearance records after relating the values to the address records.

The creation of the Bloom filter is rather simple as well. Simply spin through the list of addresses being written to the chunk, convert each address to a bit array as describe below in the *address2BitArray* function remembering to add a new bit array to the Bloom filter each time the

expected false positive rate overtops the pre-selected value (also a per-chain configuration item), and proceed.

The *address2BitArray* function "chops" the 20-byte address into five 4-byte sections, which produce five 32-bit integers modulo the width of the bit array.



The unsigned 32-bit integers are in the range [0, BitArrayWidth]. The algorithm "lights" those five bits in the current bit array. If more than the pre-determined number of addresses have been added to the current bit array (for Ethereum mainnet, this number is an arbitrarily chosen `50,000 addresses`), a new bit array is appended, and subsequent addresses light bits in that array. The number of addresses chosen to determine when to adapt the size of the Bloom filter requires further study.

After picking off "enough records" as many times as necessary (depending on the busyness of the chain, multiple chunks may be encountered in a single loop), the *ConsolidationLoop* removes those addresses from the stage, leaving the remainder of the appearances for the next run. The consolidation loop may (in the future) also pin the new Index Chunks and their associated Bloom filters to IPFS, updates the Manifest, pin the Manifest to IPFS, and, then post the IPFS hash of the Manifest to the smart contract. These features are left for a future version.

Snap-to-Grid and Correcting Errors

We mentioned above a snap-to-grid configuration item. This value is present primarily to aid in debugging, and our wish is that in a future version, it will be removed. The purpose of this function is to provide break points (or snap-to points) in the data. Because the data is produced against a continuing stream of data, and because we name the chunks by block number to ease processing, we find that if an error occurs early in the list of chunks and we wish to correct that

error, we must reproduce the entire stream of chunks. Adding break points allows us to avoid re-generating the entire index if there is an error early on.

If, instead, we break the stream of chunks periodically at a snap-to-grid block, we can correct only that portion of the data up until the next snap. On the mainnet, we find a value of 100,000 blocks produces a reasonable breaking point without creating too many extra files.

The snap-to-grid feature serves no other purpose.

A Justification for Chunking

We've been asked why we choose to chunk our data and whether it wouldn't be more efficient to store the entire index in a regular Web 2.0 database. The answer to this is more complicated than it might seem, therefore, a short explanation is in order.

The TrueBlocks system, and in particular *chifra*, is primarily a local-first command line tool. While *chifra* does provide an API server that matches all of the command line options, its first and most important use case is as a command line tool to aid in data science.

Chifra was also designed, from first principles, to run locally—that is, on an end user's machine. It is possible to make very specific performance enhancements on an application if one is building a purely local application. Choices one might never make for a web server application.

A perfect example of this is opening and reading local binary files directly. In any web 2.0 web server environment, one would never read local binary files with each invocation, and, in fact, in our server version, we have plans to avoid exactly this situation. When one is running on a command line against local files, however, the performance bottleneck becomes opening large files. In the web server environment, this cost is amortized over every request. In the local command line environment, this cost applies to every invocation.

Additionally, another important consideration is that we wanted to create a "naturally sharded and easily shared index." Breaking the index into immutable chunks, inserting those chunks into IPFS, and storing references to those IPFS files in a Manifest, makes this possible. As each end user expresses interest in different parts of the index through their regular usage, the chunked Unchained Index delivers only those portions of the index the end user needs. And, once downloaded and pinned to the end user's machine—where the speed of local data becomes apparent—the end user can be "enlisted" in sharing the data through pinning by default.

For this and many other more subtle reasons, we create the index in a chunked manner.

Conclusion of This Section

Thank you so much for reading this far. This project has taken a massive amount of effort, and your simple act of reading goes a long way to make it all worth it.

Please do us the great courtesy of giving us your feedback and helping us to improve this document. Anything from issue creation suggested edits, ideas to improve the processes described, all the way up to suggestions to fund our work, and/or donations would be greatly appreciated.

Querying the Index

chifra list

This section describes the algorithms used to querying for a list of appearances for a particular address: *chifra list <address>*.

We leave this section to be completed later.

chifra export

This section describes the algorithms used to extract full transactional histories for a particular address: *chifra export <address>* as well as all the options available for this very powerful tool.

We leave this section to be completed later.

Conclusion

Thank you for reading this far in our document. We've tried our best to describe the TrueBlocks Unchained Index system as best we can. The primary goal of this document is to describe the indexing process well enough for others to implement it themselves. The best source of information about the system is the source code itself which is fully open-source and available for review at our GitHub repositories (see below).

Fin

There's one thing left to explain. Why did we build it? Unfortunately, a sixty-page paper devoted to this question would be reasonable, but suffice it to say that we want to "re-change" the world. One of us (Rush) played a vanishingly small part in "changing the world" back in the 1990s during the birth of the world-wide-web. He wishes to make amends.

Humans have more than 25 years of experience living in a world dominated by websites. We are only now coming to understand the effects of decisions made in the 1990s when the web stack was being formed. Back then, everyone was wildly enthusiastic about the architectural decisions being made, but we missed something that now seems obvious. We missed the fact that censorable, capturable worldwide information systems are—not surprisingly—censorable and capturable. And—again, not surprisingly—that at this point, these systems have been fully captured.

By who? Q-Anon? An evil cabal of evil people? Advertisers? Who cares? It doesn't matter. What does matter is that the web has been captured by people with incentives opposite to those of us all. The web has been captured by *non*-sheeple. It's time for the sheeple to take back their own system.

The architectural decisions made in the 1990s and early 2000s were constrained by the capabilities of the hardware that was then available. Website rate limiting was born out of a necessity to protect servers from overuse. Logging in to websites was born out of a necessity to rate limit. The first websites we ever used had no concept of "logging in." Privacy invasion is a natural, unintended consequence of logging in. Only retroactively does this seem obvious.

Since that time—25 years ago—the capabilities of computer hardware advanced by 10 orders of magnitude. The software stack—consisting of web servers and user-based web clients—has not changed at all. Except for the fact that it's gotten more and more invasive. The massive gains in computer hardware capability have gone almost exclusively toward deeper and deeper privacy invasion. Not surprisingly—the people running the servers own the data.

It is our belief that the advent of distributed blockchain node software (such as Bitcoin) which, literally, allows any user to see anything was a first example of a system that takes advantage of the 10 orders of magnitude increase in hardware capability on behalf of the end user. Blockchains pull back power from the massive worldwide information and privacy invasion machine known as the Internet…supposedly.

TrueBlocks is an attempt to show by example that indexing and searching this already decentralized data may also be fully decentralized. We believe that the only reason existing blockchain data access systems, such as EtherScan, Covalent, QuickNode, Dune, and The Graph, exist is that people are lazy. They don't yet realize that it is possible to avoid web-based APIs. They don't yet understand that it is possible to build a system that is not capturable.

We tried to show in this document that if indexing were part of the node's processing and if the indexing were natively produced to be distributed through a content-addressable data store such as IPFS, the cost of publishing the same would be negligible, particularly if the system automatically invites end users to share in the burden of carrying the data. TrueBlocks does this by "pinning by default" and creating a "naturally sharded and shared chunked index."

We look forward to continuing our work and expanding our understanding of the new paradigm. Help us by contributing to our repos and/or donating to our GitCoin grant (see below).

Hopefully, this time we can all build a worldwide information system that doesn't suck.

/Fin

Appendix A: GitHub repositories

Core and SDKs

- https://github.com/TrueBlocks/trueblocks-core

Documentation

- https://trueblocks.io/docs/chifra [command line]
- https://trueblocks.io/api [api]
- https://github.com/TrueBlocks/trueblocks-docs

Explorer

- https://github.com/TrueBlocks/trueblocks-explorer

Docker

- https://github.com/TrueBlocks/trueblocks-docker

Appendix B: Social Media and Websites

TrueBlocks Home
- https://trueblocks.io

Twitter
- https://twitter.com/trueblocks

Discord
- https://discord.com/invite/kAFcZH2x7K

GitCoin grant
- https://gitcoin.co/grants/ (search for TrueBlocks)

Tokenomics
- https://www.tokenomics.io/gitcoin

References

Adaptive Bloom Filters
- Adaptive Bloom Filters for Identifying Transaction of Interest (Thomas Rush, 2017)

Bloom Filters
- https://en.wikipedia.org/wiki/Bloom_filter

RPC Documentation
- https://ethereum.github.io/execution-apis/api-documentation/