# A Specification for the Unchained Index

trueblocks-core@v0.40.0

Thomas Jay Rush
TrueBlocks, LCC
June 2022

Table of Contents

## Introduction

Immutable data—such as that produced by blockchains—and content-addressable storage—such as IPFS—have gotten married, and they've had a baby called the "Unchained Index."

Immutable data and content-addressable storage are deeply connected.

After all, without a suitable storage medium for immutable data, how can it possibly be immutable? And, if one modifies immutable data—first of all, it's not immutable, and secondly its location on IPFS changes. The two concepts are as connected as the front and back sides of a piece of paper. One cannot pull them apart—and even if one were able to pull them apart—rending the paper front from back, one would end up with two, slightly thinner, pieces of paper. There's no way around it.

This document describes the Unchained Index, a computer system that purposefully takes advantage of 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), but the examples herein focus on the Ethereum blockchain's mainnet.

### The Format of the Paper

This document begins by reviewing the Unchained Index. Following that are detailed descriptions of the binary file formats used by the system. The paper concludes by describing the algorithms to create 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. See this website (https://unchainedindex.io) for more information.

By querying a smart contract, the system obtains the IPFS hash of a manifest that points to the entirety of the full index. End users may subsequently query this index to obtain a list of "everything that ever happened" to an address. This allows a user to reconstruct the history of his account(s) without the aid of a third party (assuming they are running their own Ethereum and IPFS nodes). Furthermore, the manifest includes enough information to rebuild the index from scratch.

In the following sections of the paper, five binary file formats are described:

1) Manifest – a JSON object that carries enough information to reconstitute the index;
2) Index Chunk – a single portion of the index consisting of approximately 2,000,000 appearance records and covering a certain block range;
3) Bloom Filter – a Bloom filter encoding set membership of each address in the associated Index chunk covering the same block range;

4) Names Database – a somewhat unrelated collection of named address labels used to better articulate the query results. This small subset of known accounts contains about 13,000 records; and
5) Timestamp Database – a flat-file binary database used to optimize timestamp lookups.

You may skip ahead to the File Formats section below if you wish.

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 a traditional database, the Unchained Index is not stored in a single monolithic file. Instead, it is a collection of much smaller binary files ("chunks") and their associated Bloom filters. Breaking the index into smaller chunks is done by design in order to take advantage of content-addressable storage systems such as IPFS. This design allows for natural distribution of the index while requiring no "extra effort" from the end-user. We call this aspect of the system, "naturally sharded and easily shared." The design also imposes a near-zero cost of publication on its "publishers" of which there may be many.

Because the index is chunked, end-users are able to acquire only those portions of the index they need. "Need" being expressed naturally as a result of an end user's queries. As the end-user explores the history of an address—that is, he exhibits his own natural interest in an address—the Unchained Index delivers only that portion of the index required to fulfill the specific query. This has the happy consequence that "light" users (i.e. users interested in only a few addresses or lightly used addresses—that is, most of us) carry a light burden, while "heavy" users (those interested in large smart contracts or doing data analytics, for example) require a larger number of chunks to satisfy their queries. As a result, heavy users are able to share more chunks, thereby carrying a heavier burden. This is by design, and we think, fair.

We go a step further and pin by default. As a side-effect, the system enlists the end-user in sharing the downloaded chunks with others without "extra effort". Over time, the system becomes distributed as each chunk becomes increasingly more 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 throughout the system, (c) more resistant to censorship, (d) more difficult to capture, and (e) imposing a lessening burden on the publisher as the end users themselves are sharing in the burden of publication.

*Reproducible*

The content-addressable nature of the storage also aids in making the Unchained Index reproducible. A primary data structure in the system is called the Manifest (the JSON format of which is described below). As each chunk is produced, the block range covered by that chunk, the IPFS hash of the chunk, and the IPFS hash of the chunk's Bloom filter are appended to the manifest and the manifest itself is written to IPFS. The IPFS hash of the manifest is then enshrined in the Unchained Index smart contract (which is also detailed below).

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@v0.40.0-beta").
2. The manifest also records the IPFS hash of this exact PDF document. In this way, end-users who have access to the manifest have a full specification of the system used to create it. It is expected this specification will not change frequently.
3. The keccak_256 of the version is inserted into each binary chunk of the index prior to publishing it to IPFS. In this way, if the user obtains any portion of the index, he/she knows exactly which specification under which the portion was written.
4. The IPFS hash of the manifest is periodically posted to the Unchained Index 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 a later point, if a user wishes to verify the contents of a 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 themselves against thier own blockchain node. This, presumably, produces the exact same result.
6. We consider it the responsibility of the end-user to satisfy themselves as to the veracity of the data in the index. We make it easy for them to get the data, we do not claim it's correct.

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." If the end user wishes to have proven data, she has all the tools she needs to prove the data herself.

*A Further Note on Reproducibility*

TrueBlocks creates this index data for our own purposes. We want our end-user-focused software to work properly. In this sense, we are motivated to produce accurate data, and we are quite certain that the data we produce is accurate. While we purposefully allow others to use the data by exposing it in the smart contract, we reject any sense of responsibility to vouch for the data. The data is correct because our software demands that it be correct. Others may use it if they wish—but beyond that, we make no other representations.

## A Short Digression on Our 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 member of a set." This fits perfectly in our design. For each chunk, the system produces an associated Bloom filter. Upon first use of the system, the end user may download only the Bloom filters (about 2.5 GB). Alternatively, they may, if they wish, download both the Bloom filters and all of the index

chunks (about 80 GB). As a final alternative, the end user may choose to 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, 2-3 GB |
| Download time: | 15-20 minutes |
| Query speed: | Slower the 1st time one queries an address, then 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 '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 only 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 him." In other words, the system places a burden commiserate with the user's behavior. Users who interact infrequently with the chain, get only a small amount of data (in proportion to their usage). Queries against addresses that interact very frequently with the chain—such as popular smart contracts—hit on nearly every Bloom filter. In this case, the user downloads 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 are downloaded during the query. Subsequent queries for the same address, however, 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, ~75 GB at the time of writing |
| Query speed: | Very fast queries on all addresses as there is no additional downloading |
| Download time: | ~1-3 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 with chifra init –all the entire Unchained Index (including all of the chunks and all of the Bloom filters) is downloaded. This process may take hours to complete depending on the end user's connection. 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 to avoid reading the chunk if possible), there are no further downloads during the query. All index chunks are already present on the machine. If one is studying an address that appears frequently on the chain or many different addresses with varying usage patterns, this method is probably the best.

*chifra scrape: building the index from scratch*

| | |
|---|---|
| Disc footprint: | Large, ~75 GB at time of writing – same size as method 2 |
| Query speed: | Fast queries – same as method 2 |
| Download time: | 2-3 days depending on speed of node software and 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, building the index yourself, is the most secure, particularly if you review the source code first. One accomplishes this third method by running chifra scrape run (which is the same command one must use to stay up to the head of the chain). If you've reviewed the source code and concluded that it does what it says it does, and you're running the scraper in a secure environment against your own locally running node, this is the most secure method to obtain the index. 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 build 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, although you may enable pinning if you wish.

In future versions, pinning will be enabled by default as will running an embedded IPFS node. 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 happy 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 "off-fall" of the system. This is by design. Requiring "extra effort" from an end user almost guarantees the long-term failure of the system.

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

We purposefully built this system to naturally distribute the index (which, remember, is available to anyone through the smart contract). We wanted to create a system with positive

externalities—that is, we wanted to design a system in which each new user makes the system better as opposed to placing an increasing burden on the system.

## Conclusion

We've spent time explaining the Unchained Index because this may explain some of the engineering decisions we've made in the data.

In the next sections of the document, we detail, first, the Unchained Index Smart Contract, then the file format of the Manifest, then the file formats for each of four binary files. Each format is presented in its own section. In the following, we present this information as stylized Solidity or GoLang source code to ease explanation.

## Smart Contract

### The Unchained Index Smart Contract

```
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 for donations. May be modified.
    address public owner;

    // A map pointing from the address that wrote a record to the record
    // itself. A record is an entry in a map pointing from a chain to
    // the current IPFS hash of the manifest representing the index
    // of addresses for that chain. End users are encouraged to query
    // this map for a publisher that they themselves trust. Any publisher
    // may write any number of records.
    mapping(address => mapping(string => string)) public manifestHashMap;

    // The contract's constructor preserves the deploying address for the
    // contract as the owner. It also initializes a single record for the
    // mainnet chain 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 primary function of the contract, this routine allows anyone to
    // publish a record to the smart contract. End users may choose to use
    // any record they desire. TrueBlocks makes no representation as to the
    // quality of any data published through this smart contract, however,
    // because this data is used by our own applications, it satisfies us.
```

```
    // The publishHash function is purposefully permissionless. Anyone
    // willing to spend the gas may publish a hash pointing to any IPFS
    // file. Anyone may also query the contract for records published
    // by any publisher. This is by design. End users must determine for
    // themselves who to believe. We suggest it's us, but who knows?
    //
    // This function writes a record to the map and emits an event. Note that
    // any data published by any publisher is permitted.
    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);
}
```

## File Formats

### The Manifest File

The manifest file is produced each time a new index chunk (and Bloom filter) is produced. It is a simple JSON object that stores five things: (1) the version of this document that describes everything one needs the index from scratch; (2) the name of the blockchain that this manifest indexes; (3) the IPFS of this document; (4) the IPFS hash of a zipped tar file made from a directory containing various off-chain databases at the time of publication; and (5) a list of chunks and associated Bloom filters detailing the entire index. Note that the version string is of this document, not the TrueBlocks/trueblocks-core repo.

The JSON format of the Manifest file

```
{
  "version": "trueblocks-core@v0.40.0",
  "chain": "mainnet",
  "schemas": "Qmart6XP9XjL43p72PGR93QKytbK8jWWcMguhFgxATTya2",
  "databases": "Qmart6XP9XjL43p72PGR93QKytbK8jWWcMguhFgxATTya2",
  "chunks": [
    {
      "range": "015013585-015016368",
      "bloomHash": "QmREw5qaoucbVvEQzF71D44rXKzax9YgKuEEhZYHAYFZF5",
      "indexHash": "QmTbFshRSdBFoC6AvBgzdRJ6Vgb9cVL3yTprYQ24XqHTqx"
    },
    {
      // and so on...
    }
  ]
}
```

The Manifest is produced each time a new chunk is produced. The algorithm to produce the chunks is described below. After producing the manifest, it is formatted with the command line tool **jq** and stored in a text file. That text file is added to IPFS and the IPFS hash of the file is then (periodically due to cost) published to the smart contract. (In our case, the publication is completed by the `trueblocks.eth` wallet which is also the contract's deployer.)

Once published, a few things become true:

1) The publication cannot be undone—this version of the Manifest will be on-chain forever readable by anyone with access to that blockchain,
2) Anyone who reads the manifest may download this specification and any chunks and Bloom filters referenced by the manifest,
3) The publisher (us) has no further on-going costs of publication other than pinning the files on IPFS (which carries a near-zero cost).

Over time, as more and more users download and pin more and more portions of the index—which happens by default—the resiliency and speed of the system increases in proportion to the number of users. This is a classic case of positive externalities and “If we all build it, we can all come.”

## The Index Chunk File

We describe the format of the index chunk file as a GoLang structure. Following that is the source code (in GoLang) one would use to read and write these files. There are ~2,750 individual chunks in the Ethereum mainnet index at the time of this writing and the same number of associated Bloom filters.

The binary file consists of a single fixed-width header containing versioning information and two 32-bit integer counters recording the number of records found in each of the two fixed-width tables that follow the header.

The GoLang structure of an Index Chunk file

```
// Each binary chunk contains a single header row followed by two related
// fixed-width tables. The first table, addresses, relates to the second,
// appearances.
type IndexChunk struct {
    Header          HeaderRecord
    AddressTable    []AddressRecord
    AppearanceTable []AppearanceRecord
}
```

The Header record takes the following form

```
// The first 44 bytes of the file contains the header which contains version
// information and two 32-bit integer counters detailing how many records
// are in each of the two database tables.
type HeaderRecord struct {

    // ‘0xdeadbeef’ indicates that this is a known file format
    MagicNumber   [4]byte

    // The version string of this specification. This value ensures that anyone
    // receiving this file knows how to read it. For all chunks up to and
    // including block 13,000,000, this field contains all zeros. For chunks
    // covering blocks after 13,000,000, this value is the keccak256 hash
    // of the version string of this specification.
    Version       [32]byte

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

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

The Addresses table, which follows the Header, contains a table of fixed-width AddressRecords. The number of records in the table is detailed in the header. The address table relates (using the offset and count fields) to the position of an address's appearance records in the appearance table.

```
// For each address found in the block range, this table stores the address
// and two unsigned integers. The first integer is the offset into the
// appearance table where this address's records begin. The second integer
// records the number of records to read.
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 appearance table consists of an array of AppearanceRecords. An AppearanceRecord which is described in more detail below is a record of two 32-bit unsigned integers showing the block number and transaction index where each address in the chunk appears.

```
// 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 TrueBlocks search algorithms try to avoid reading the chunk file. This is why we create the Bloom filters which allow us to quickly determine if an address appears in the chunk. Only when the Bloom filter hits on an address must we open and read the chunk file.

The algorithm for reading a single chunk is presented below. Note, that we've removed error processing for clarity.

Algorithm for querying an address against a single chunk file

```
// 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
}
```

**Note:** Because this function opens a file from disc, it's desirable to avoid running it if at all possible. Enter Bloom filters…

### The Bloom Filter File

As described above, the Unchained Index is a collection of "chunks" of a regular database's index file. Each chunk is a near-equal-sized piece of the index so as to promote fair distribution of the index via a content-addressable store while imposing a minimal cost on the publisher. Another important part of the system, however, is the following fact. As each "chunk" is created, a second file called a Bloom filter is created. The algorithm used to build the Bloom filter is described in the section Building the Index and Bloom Filters.

The Bloom filter is a probabilistic data structure that encodes set membership in a very compact way. Each chunk, on average, is 25 MB big and contains approximately 2,000,000 appearance records. The associated Bloom filters are, on average, one MB big. Each Bloom filter is independent of all the others which means the search may proceed concurrently, however, the code below, we've removed concurrency in order to make the algorithm easier to understand.

Algorithm for searching Unchained Index using Bloom filters

```
// 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 {

        // Store a list of which addresses hit the Bloom
        hits := make([]address)

        // For each address...
        for _, addr := range addrs {
            // If the address hits the Bloom...
            if bloom.IsElementOf(addr) {
                // Make a note...
                hits = append(hits, addr)
            }
        }
```

```
        if len(hits) > 0 {
            // ...we download the chunk here if it's not already present...
            for _, addr := range hits {
                apps = append(apps, GetAppearances(addr, bloom.FileName)
            }
        }
    }
    return apps
}
```

Because the **IsElementOf** function depends on the binary format of the Bloom filter, we describe the file format of the Bloom filter first.

TrueBlocks uses a multi-part Bloom filter that we call an Adaptive Bloom filter. The "parts" of the multi-part filter are bit arrays encoding the set membership of the addresses in the chunk. Bit arrays are 1/8 of one megabyte big (an arbitrarily chosen width).

The method by which we insert addresses into the Bloom filters is described below in the section called "Purposeful Sloppiness".

The word "Adaptive" above refers to the fact that we grow the Bloom filter as large as it needs to be to maintain a pre-define "maximum expected false positive rate." We "grow" the filter by adding additional BitArrays. Consult the source code for further information.

In essence, as the expected number of false positives becomes larger due to adding additional addresses and finally overtops a pre-determined rate, we append another bit array to the Bloom. In this way, the expected false positive rate does not go over a pre-determined value. In this way, each address is as likely as any other to hit the Bloom if the address is present.

The format of the binary Bloom filter file

```
// The full Bloom filter structure
type BloomFilter struct {

    // The number of bit array structures
    Count       uint32

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

}
```

The *Count* field records the number of BitArrays. The BitArray structure is described next.

The format of the BitArray structure

```
// an arbitrarily chosen width of each bit array
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 BitArray is BitArrayWidth 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 a conscious choice made 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. We 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.

In summary of the algorithm to query the Unchained Index for appearance of a given address:

```
for each address of interest
  for each bloom filter…
    consult the bloom filter to see if it contains the address
    if yes,
      download the chunk if it's not already present
      extract the appearances in this chunk for the given address (if any)
```

Next, we describe a few ancillary databases used by the TrueBlocks system. Technically these data are not required for the operating of the Unchained Index, but the Manifest does reference them, so we describe them here.

## The Names Database File

The names database is not technically part of the Unchained Index, per se, but it is useful. We publish the IPFS hash to the names database into the manifest, thus its inclusion here.

Carrying the namesDB hash in the manifest allows end-user software to acquire the database without a third party. This lowers the cost of publication. The file is a binary file consisting of fixed-width records. The record count is easily calculated from the file's size.

Format of the names database

```
// The binary format for the names database
namesDb := []NameRecord

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

The structure of a NameRecord

```
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 need for a timestamp database becomes apparent as soon as one tries to query the node for timestamps. The RPC does not include such a query resulting in the need to scan the chain for the result. An external database of timestamps speeds up that query many times over. This database is created during the scraping process and its location is published to the smart contract as part of the manifest.

The structure of the binary timestamps database is

```
// The binary format for the timestamps database
timestampDb := []TimestampRecord

// The number of records in the database is calculated using the database's size
nRecords := fileSize(<path>) / sizeof(TimestampRecord)
```

A single TimestampRecord structure looks like this

```
// A single timestamp record in the timestamps database
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

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

Finding the timestamp for a given 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))
```

We note that the timestamps database could have been half as big if we had removed the block number from the file. The block number is sequential, starts with zero, and contains no gaps—a perfect index). We choose, however, to include block numbers in the data as an aide in debugging and checking of the databases's integrity. We reserve the right to change this format in future versions.

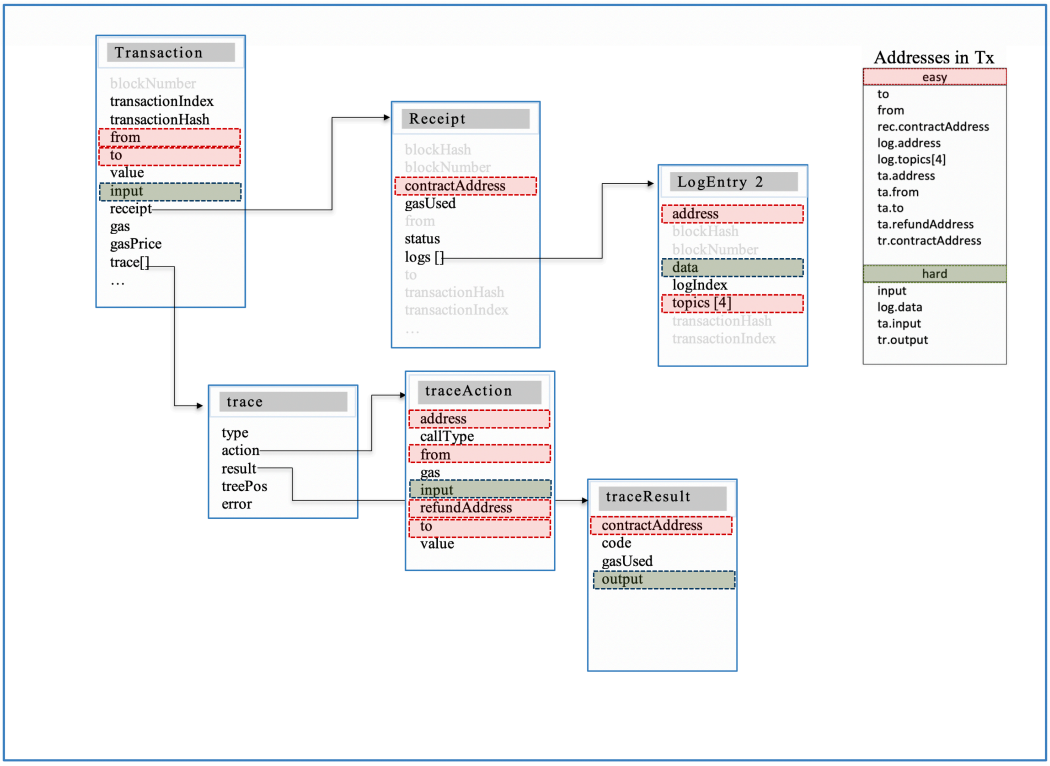
## Building the Index and Bloom Filters

Above we've described the binary file formats of the various databases that make up the Unchained Index as well as 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. We start with a definition of the primary data structure in the index: the appearance. We then proceed to discuss how we extract appearances from the blockchain data. We conclude by justifying a few engineering decisions we've made that were forced upon us due to our desire to preserve our ability to preserve data immutability, reproducibility, and content-addressability.

### Definition: Address Appearances

We define an *Appearance* broadly to mean anywhere a particular byte pattern is encountered on chain that "appears" to be an address. Our algorithm is "purposefully sloppy" in what we consider an appearance (in much the same was as a bloom filter will exhibit no missed set memberships, but may return false positives. We do this because we would rather include byte patterns that "appear" to be an address but are not, rather than not include actual appearances. It turns our this is not possible, which leads to the idea of a "baddress" which is described below.

Looking closely at a typical blockchain's block data, one may identify up to 16 different locations in the data where an address may appear. This is most easily described with an image. (Note The image shows 14 locations. Block and uncle miners are not represented and this is why we say 16 locations and not 14.)



The pink items shown above are what we would call “easy-to-find” locations. These are places in the data where addresses are obvious. Some addresses (such as **to**, **from**, **miner**, and **log emitters**) are defined in the data as addresses.

Other obvious locations, such as certain logs, are easily inferred. For example, ERC20 Transfer and Approve logs (identifiable by topic) contain known locations for addresses. While this method is effective, it is not extensible as it requires knowledge of the format of the bytes. As new protocols appear, this specification would have to change. Even if that were possible, it introduces an aspect of the algorithm that would necessarily destroy the long tail of unknown on-chain behavior. (Something we specifically wanted to avoid.)

Looking at other areas of the data (colored green above) on encounters the true reason why identifying appearances is difficult. These areas of the data, while frequently containing actual appearances of addresses, are pure byte data.

Because we want to avoid destroying the long tail and capturing as may appearances as we can, our scraping process purposefully operates with minimal external context. We desire to identify appearances without knowing anything about the format of the data. While this is not even theoretically possible in the general case (every byte pattern under or equal to 20-bytes wide is a valid addresses), we take advantage of two aspects of the Ethereum byte stream that are produced by the way the Solidity programming language works:

1) Addresses are 20 bytes long,
2) Nearly every number encountered in the operation of Ethereum is smaller than nearly every address (this touches on the idea of a “baddress”.

The above comments are obscure and better explained below in the “Purposeful Sloppiness” section.

(Yes – we recognize †he above context refer to external context produced by Solidity, but this external context is applicable to all smart contracts and is not relative to particular smart contracts such as the ERC20 data above. In this way, it avoids destroying the long tail.)

As we are scraping the byte data of blocks, we use certain heuristics to identify appearances in these “hard-to-find” (that is, green) locations.

### Notes on Per Block Data

We note that the appearance data buried in a given block (once canonicalized) is independent of the appearance data contained in any other block. In this section we describe how the Unchained Index processes a single block. The reader is encouraged to consider the obvious opportunities to make this processing concurrent. (We do this heavily in our implementations.)

The processing described here is at the heart of the Unchained Index. The chunking algorithm requires this per-block processing in order to build the chunks in a process we call

"consolidation." We describe the process of consolidation after describing the per-block processing.

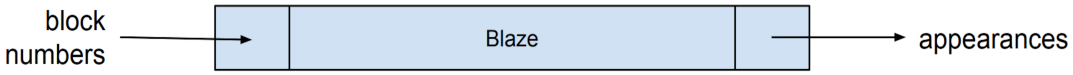
A further note before proceeding: These algorithms only work if one is running the node software locally for two reasons:

1) Most remote Ethereum node data providers do not support providing trace data and if they did, it would cost an exorbitant amount of money.
2) In order to complete the processing in a timely manner, the algorithm hits the node very heavily. This will quickly get you rate limited when using a remote provider.

We highly recommend Erigon.

### Extracting Addresses (Per Transaction) Per Block

The following algorithm is contained in the Blaze section of the Unchained Index codebase. Blaze is a "pipeline" in the old-fashioned command line sense of the word as well as in the new-fashioned GoLang sense of the word (i.e., channels). Blaze, which is dependent only on the node's RPC endpoints, "shoves" block numbers into one end of the pipe and, almost by magic, address appearances emerge from the other.




Let's dig into Blaze. Note that we've simplified this code from the actual implementation in order to ease explanation. Please consult the source code for full details of the algorithm.

While conceptually Blaze is a single pipeline, internally we need to channels. The first processes blocks and produces a stream of 'candidate addresses." The second considers (using the heuristics mentioned above) each byte stream parsing out appearances. The effect is as the image above shows: blocks go in, address appearances come out.

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

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

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

We next create a number of go routines to process data inserted into the pipelines. Because different blockchains have different characteristics these values are provided as command line options.

```
    // 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)
    }
```

The next step is to insert blocks into the top of the pipeline.

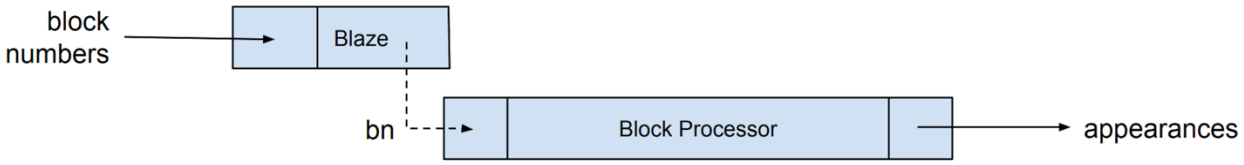
```
    // Shove the blocks into the block pipeline
    for bn := opts.Start ; bn < (opts.Start + opts.Count) ; bn++ {
        blockChannel <- bn
    }
```

And finally, we close the channels and wait until the pipeline finishes processing. When Blaze returns a file has been written containing all the AppearanceRecords found in the block range processed. The Consolidation Loop will post process this data.

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

    addressChannel.Close()
    addrWg.Wait()

    return nil
}
```

Blaze is called with a block range. It is the responsibility of caller to determine which block range to process. In this case, the caller is being called by the Consolidation Loop which is described below. The output of Blaze is written inside the pipeline—that is, the algorithm "streams" the results. This allows for arbitrarily-sized block ranges (including one block) and does not require the algorithm to store results in memory.

Next, we peer into the block processor (the processBlocks go routine). The block processor queries the node given block numbers, collates the data into a more useful format (ScrapedData) and "shoves" that collated data into the address processor (processAddresses).




The block processor ranges over the block channel waiting until block numbers appear. Once they do, it makes three RPC calls and combines the result into a ScrapedData structure. It then shoves this combined data into the address processing channel. When the blockChannel closes, the wait group is marked as completed and the function returns.

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

    // Range over the block channel. Note values are non-sequential and
    // are processed concurrently.
    for bn := range blockChannel {

        // Query the node and combine results for further processing
        combined := ScrapedData{
            BlockNumber: bn
            TimeStamp: client.GetBlockTimestamp(bn)
            Traces: client.GetBlockTraces(bn)
            Logs: client.GetBlockLogs(bn)
        }

        // Do further processing...
        addressChannel <- combined
    }
}
```

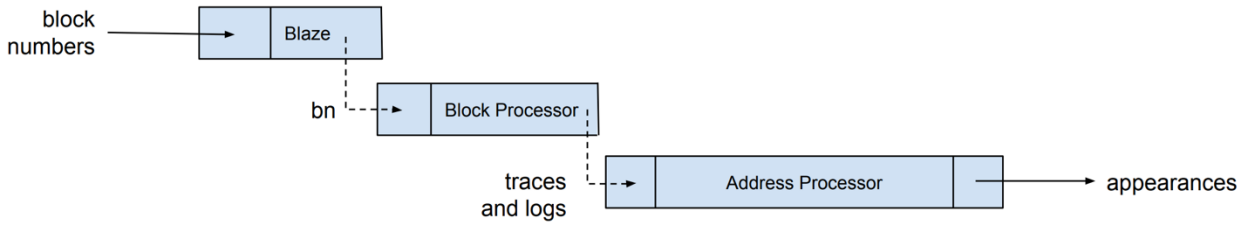
A few notes on this function:

1) This code is purposefully concurrent-aware. Any number of go routines can process this data independently of other blocks;
2) In future implementations, we intend to process groups of blocks at a time. This would allow us to use **eth_getLogs** with a block range in which case it is much faster. Here, we process one block at a time. Processing multiple blocks at a time would require us to insert block ranges into the channel instead.

Next, we detail the ScrapedData structure which is called “combined” in the above code. This structure combines the received information from two RPC commands (**trace_block** and **eth_getLogs**). This makes further 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 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 block processor passes the above data on to the address processor which we describe next. Conceptually, we are here




The address processor accepts ScrapedData structures as they are inserted into the addressChannel and finds “every appearance of every address anywhere in the block.” For each identified address, the block number and transaction id are recorded and written to disc for later processing by the Consolidation Loop. When the channel is closed, the wait group is marked as being done.

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

    // range across the scraped data in the channel
    for scrapedData := range addressChannel {
```

```
        // range across the scraped data in the channel
        addressMap := make(map[AppearanceRecord]true)

        // extract address appearances from the trace data
        extractFromTraces(sc.blockNumber, sc.Traces, addressMap)

        // extract address appearances from the log data
        extractFromLogs(sc.blockNumber, sc.Logs, addressMap)

        // write addresses to the output medium
        writeAddresses(sc.blockNumber, addressMap)
    }
}
```

We describe the processing of trace and log data below.

### eth_AddressesPerBlock

It does not escape our notice that the above routines could be a very useful addition to the RPC specification. If the node software itself had a routine called eth_AddressesPerBlock, 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 Traces

The trace data contains a field called Type. The Type field takes on the following values

```
// 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 (subtypes of "block", "uncle", and
    // "external")
    Reward

    // Generated due to a self-destruct call
    SelfDestruct

    // Generate during the creation of a smart contract (CreateFailed if
    // deployment transaction fails)
    Create
)
```

The different types of traces carry data that contains addresses in differing locations. In some cases, those locations are explicit – in the sense that those values are stored locations that are explicitly documented as being an address. These addresses are "easily-found" (pink in the above image).

For other trace types, the addresses are represented implicitly in raw byte data such as input and output fields for function calls. When an explicit address is found, we quickly add it to the growing appearance map. When the addresses we are processing are implicit, we must be creative—using an algorithm we've described as "purposefully sloppy."

The Trace data structure is made up of two sub-structures called the Trace.Action and the Trace.Result (see the RPC documentation). We summarize the various trace types, address locations, and explicit / implicit aspect of the trace data 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 |
| Any | Input (function call data) | Implicit |
| Any | Output (function output data) | Implicit |

The extraction aspects labeled as Implicit above are described in more detail in the section called "Purposeful Sloppiness."

### Extracting Addresses from Logs

This section describes the GoLang implementation of the block scraper called Blaze.

### Purposeful Sloppiness

This section discusses our special algorithm to identify address appearances and why we call the "appearances".

### Baddresses

This section discusses the idea of a "baddresses," which is ignored by the index and why they are important.

### Snap-to-Grid and Correcting Errors

This section discusses the 'snap-to-grid' feature of the indexer and why it is important.

### Consolidation Phase

This section discusses the consolidation phase of the algorighm, how we determine the number of records at which to consolidate, how to best choose that value for differing chains, and the algorithm used to create our enhanced, adaptive Bloom filters once a chunk is created.

### A Justification for Chunking

In which we explain why we've chosen to keep the files chunked. (Because of wanting to store on IPFS) and we further explain that in our implementation we can re-combine the chunks into larger files and/or store the entire bloom array in memory.

### Conclusion

This section completes the discussion.

## Querying the Index

*[This section is not complete.]*

### chifra list

This section describes the algorithm used to allow querying for an address: **chifra list <address>**.

### chifra export

This section discusses the various options available for **chifra export <address>**.

## Conclusion

A concluding paragraph.

## Supplementary Information

*[This section is not complete.]*

Website
GitHub repositories
- Core
- Explorer
- Documentation
- Docker Core
- Docker Monitors

GitCoin grant
Tokenomics