A Specification for the Unchained Index

version: trueblocks-core@v0.40.0

Table of Contents

Introduction

Immutable data—such as time-ordered logs produced by blockchains—and content-addressable storage systems—such as IPFS—are intimately related.

Without a suitable storage medium to store immutable data, how can it possibly be immutable? Furthermore, if one modifies immutable data, its location on a content-addressable storage medium changes. The two concepts are as closely connected as the front side and the back side of a piece of paper. One simply cannot take them apart—and even if one were able to split a piece of paper by rending it front from back, one would simply end up with two, slightly thinner, pieces of paper.

The concepts are metaphysically interconnected.

This document describes a certain aspect of the TrueBlocks system called the Unchained Index. The Unchained Index purposefully takes advantage of this tight coupling between immutable data and content-addressable storage.

The mechanisms described in this paper are applicable to any time-ordered log, not just blockchains, but the example herein are focused on Ethereum's mainnet.

The Format of the Paper

This document begins by describing certain aspects of the Unchained Index. Following that are detailed descriptions of the binary file formats of four different file types used by the system. In coordination with an Ethereum smart contract where the IPFS hash of a manifest is periodically published, end users may read the contract, download the manifest and read it to find the IPFS hashes of each portion of the index. This allows any user to reproduce the index without the aid of a third party (assuming they are running their own Ethereum node).

The four file formats are:

1. *Index Chunk* – a portion of the index of appearances database consisting of at least 2,000,000 records;
2. *Bloom Filter* – a Bloom filter encoding set membership of every address in the Index chunk covering the same block range;
3. *Names Database* – a collections of names for a very small subset of known addresses (about 13,000 names); and
4. *Timestamp Database* – a flat-file binary database making timestamp lookups significantly faster than querying the node.

Each file format is specified below. You may skip ahead to the File Formats section below if you wish.

A Short Digression on the Unchained Index

The Unchained Index is a naturally-sharded, easily-shared, reproducable, and minimal-sized immutable index for EVM-based blockchains. See this website (https://unchainedindex.io) for more information.

Naturally Sharded, Easily Shared

Unlike a traditional database, the index produced by 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 designed to take advantage of content-addressable storage systems such as IPFS. This design allows for broad distribution of the index while imposing a minimal burden on the end-user and a near-zero cost of publication on its "publishers".

Because the index is chunked, end-users may acquire and later share (i.e. pinning by default) only those portions of the index they need. "Need" in this case happens naturally as a result of an end-user's behavioiur. As the end-user's queries for address—that is, he exhibits a natural interest in some addresses and not others, the Unchained Index is able to deliver only that portion of the index needed to fullfil the secific query. This has the happy outcome that users with small needs (i.e. he/she is interested in only a few addresses with a small number of appearances) carry a small burden. Users interested in heavily used addresses will require more chunks to satisfy their queries, and as they will share those chunks, they carry a heavier burden. This is as it should be.

As a side-effect of using content-addressable storage, the system enlists the end-user in sharing (i.e. pinning) the results with other users. Over time, the system becomes sharded and each chunk becomes increasingly more available because more and more users are sharing. As the system matures, the index becomes shared among community members making it (a) more resilient and resistant to censoring, (b) higher-performant as more copies are available throughout the system, (c) more difficult to capture, and (d) requiring a lessending burden from the publisher.

Reproducable

Content addressability also aids in making the Unchained Index reproducable. One of the primary data structures in the system is called the "manifest" (the format of which is also described below). As each chunk of the index is produced, the block range that chunk covers, the IPFS hash of the chunk, and the IPFS hash of the chunk's Bloom filter are appened to the manifest. After appending, the manifest itself is written to IPFS and the IPFS hash of that version of the manifest is enshrined in a smart contract called the Unchained Index (details of which are also presented below).

The manifest contains other information that makes the Unchained Index “reprodcuable” in the following sense:

1. The manifest contains the version string of this specification (currently “trueblocks-core@v0.40.0-beta”).
2. The IPFS hash of this specification document is also included in the manifest so that end-users have all the information they need to read the binary files. It is expected that this specification will change infrequently.
3. The keccak_256 of the version string is inserted into each binary chunk of the index prior to publishing the chunk to IPFS. In this way, the user of the index chunk knows exactly which specification it is a part of.
4. The IPFS hash of the manifest is posted to the Unchained Index smart contract, thereby enshrining it forever on the blockchain. Once published, the publisher may no longer take the information back. It is available to anyone for as long as the blockchain continues to run.
5. Later, if a particular user wishes to verify the contents of the index (or any portion), that user may read the smart contract, download the manifest, download this document and the tagged version of the source code, and re-run the code against the same blockchain (which will presumably produce the same results).
6. We consider it the responsibility of the end-user to satisfy themselves as to the varacity of any data produced by this system. Having said that, we make it as easy as we can for the user to do exactly that.

Because the manifest contains all the information necessary 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 that our data is correct. If the end user wishes to have proof that the data is correct, the end user has all the tools he/she needs to do so.

TrueBlocks is not creating this data for any purpose other than our own. We want our software to work. In that sense, we are motivated to produce excellent data. We are quite certain that the index data we produce is correct. While we purposefully built the system to allow others to use the data, we reject any sense of responsibility to prove that it’s correct. It’s correct because our software demands that it be correct. Others may use it if they wish—but it doesn’t matter to us if no-one does.

A Short Digression on 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 with our design for the Unchained Index. For each chunk, the system produces an associated Bloom filter. Upon first use of the system, end users may download only the Bloom filters (about 1.5 GB). They can, if they wish, alternatively choose to download not only the Bloom filters but the index chunks as well, however this places a burden of about 125 GB on the end user. As a further option, the user may wish to create the index themselves. If they have their own locally-running node, this is the best way to be sure to get valid data.

These three methods are explained here: https://trueblocks.io/docs/install/get-the-index.

Method 1 – Downloading only the Bloom filters from IPFS

Disc footprint:	Small, 1-2 GB
Query speed:	Slower for 1st time queries on a given address, then as fast as other methods
Download time:	15-20 minutes
Hard drive space:	In direct proportion to the user's query patterns
Sharing:	Shares Bloom filters and downloaded index portions through pinning by default
Security:	Data is created by TrueBlocks, less secure than producing it yourself
RPC endpoints:	Works with remote RPC endpoint, but much prefers local RPC endpoints
Ongoing burden:	The end user must run scraper to maintain 'front of chain' index

When initialized with `chifra init`, the TrueBlocks system downloads only the Bloom filters. 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 ever acquires index chunks that "matter to him." In other words, the system is "fair." Users who interact infrequently with the chain, get only a small amount of data (in proportion to their usage). Queries for addresses that interact very frequently such as popular smart contracts—that is they appear in nearly every block—will hit on nearly every Bloom filter. In this case, the user would download a much larger portion of the full index.

In this mode, an initial query for a new address may take a few moments (as the full index chunks are downloaded), but subsequent queries for the same address will be as fast as the other methods. Unless one is querying a huge collection of different and changing addresses, this slower initial query may be worth it, as this method imposes the smallest disc footprint.

Method 2 – Downloading Bloom filters and full index from IPFS

Disc footprint:	Large, ~120 GB at time of writing
Query speed:	Very fast queries as there is no downloading at time of query
Download time:	~1-3 hours depending on internet connection speed
Burden size:	The full index is stored on the end user's machine
Sharing:	Full sharing of the entire index (good citizen award!)
Security:	The data is produced by TrueBlocks – not as secure as building oneself
RPC endpoints:	Works with remote RPC endpoint, but much prefers a local endpoint
Ongoing burden:	The end user must run scraper 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 download. 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 the Bloom filters are still consulted during the query (because it's much faster to avoid reading the full chunk if possible), there are no futher downloads during the query. The chunks are already present. If you're studying an address that appears frequently or you're studying many different addresses with varying usage patterns, this method is probably the best.

Method 3 – Building the index from scratch

Disc footprint:	Large, ~120 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 – same as method 2
Security:	Most secure, but not as secure as reviewing the open source code as well
RPC endpoints:	Generally won't work with remote endpoints – you will get rate limited
Ongoing burden:	The end user must run scraper to maintain 'front of chain' index

The final method to acquire the index is to build it yourself. One does this with `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 version. If you're running against a remote RPC endpoint, you will be rate limited because TrueBlocks hits the node as hard as it can. This method has the same disc usage and query characteristics as method 2. In that sense, it's only benefit is that you build the index yourself.

Pinning by Default

In currently available version of the Unchained Index, the system does not pin the downloaded or produced index by default, although, you may enable this feature if you wish.

In future versions, pinning will be enabled by default. This will be an important day for TrueBlocks as it will, for the first time, become a truly decentralized method of producing and publishing an index. Pinning by default has the happy property that, as users acquire and retain the index (or portions thereof), they are sharing the index with other users. This will happen with "extra effort" from the end user—in other words, sharing happens as a by-product of the use of the system.

Obviously, acquiring and retaining those portions of the index the is interested in are in that user's self-interest. The user will retain this data because they need them. Pinning allows the user to share those portions with no extra effort. Each chunk contains the records of interest to that user, but they contain many other records as well—records that other users will need. It's a perfect example of "You scratch my back, I'll scratch yours."

All of this is by design. We purposefully built a system that naturally distributes the index (which, remember is available to anyone without censure through the smart contract). We want purposefully created a system that has positive externialities—that is, new users make the system better.

Conclusion

We've spent a little bit of time explaining the system, however this document is intended to specify the binary file formats of the files that are produced and stored in the manifest file that is published to the Unchained Index smart contract.

In the next section we describe the algorithms used to build the index, consolidate chunks and subsequently query the index for address histories.

In the remainder of the document in the following section, we detail first the Unchained Index Smart Contract, then the file format of the Manifest, then each of four file formats for the Index Chunk, the Bloom Filters, the Names Database, and the Timestamps Database. Each format is presented in its own section. We present this information in the form of highly commented Solidity or GoLang source code as this is as tight a representation as we can think of.

Smart Contract

The Unchained Index Smart Contract

```
pragma solidity ^0.8.13;

// The Unchained Index Smart Contract
contract UnchainedIndex_V2 {
    // The address of 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.
    // A record is an entry in a map pointing from a chain to the current
    // IPFS hash of the manifest representing the latest index for that chain.
    // End users are encouraged to query this map for any publisher that they
    // trust. We make no representation as to the quality of the data produced
    // by any particular publisher including ourselves. Notwithstanding this,
    // by querying the ‘owner’ the user may find those records published by us.
    mapping(address => mapping(string => string)) public manifestHashMap;

    // The contract’s constructor preserves the deploying address for the contract
    // as the owner (see below). It also initializes a single record pointing the
    // Ethereum mainnet’s manifest hash to 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(
            msg.sender,
            "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 chose 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 satifies us.
    //
    // Note: this function is purposefully permissionless. Anyone who is
    // willing to spend the gas may publish a hash pointing to any IPFS
    // file. Also anyone may query that hash by any given publisher. This
    // is by design. End users themselves must determine who to believe.
    // We suggest it's TrueBlocks, but who's to say?
    //
    // This function writes a record to the map and emits an event.
    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(owner, msg.value, block.timestamp);
    }
```

```
    // The 'owner' address serves only the purpose to accept donations.
    // If, at a certain point, we decide to disable or redirect donations
    // we can set this to the zero 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 a simple JSON object that stores five things: (1) the version string of this document which describes everything one would need in order to build the index contained within itself; (2) the name of the blockchain to which this manifest applies; (3) the IPFS of these current document; (4) the IPFS hash of a zipped tar ball made from a directory containing various off-chain databases; and (5) a list of chunk descriptors detailing the entire IPFS manifest of the index chunks and bloom filters defined below.

The JSON object has this format:

```
{
    "version": trueblocks-core@v0.40.0-beta,
    "chain": "[mainnet|sepolia|gnosis|…]",
    "schemas": "<IPFS hash to this file>",
    "databases": "<IPFS hash of gzipped tar ball of off-chain data>",
    "chunks": [
      {
        "range": <block range inclusive for this chunk>,
        "bloomHash": "<IPFS hash of bloom filter covering range>",
        "indexHash": "<IPFS hash of index chunk covering range>",
      },
      {
        "range": …and so on…
      }
    ]
}
```

This file is produced each time a new chunk (and its associated Bloom filter) is produced or as we call it "consolidated." The algorithm to produce the chunks and their Bloom filters is described above. After producing the above Manifest, the file is formatted with **jq** and stored in a regular test file. That text file is added to IPFS. The IPFS of the manifest is then (periodically due to cost considerations) published to the above smart contract. (In our case, this publication is completed by the `trueblocks.eth` wallet which is also the contract's owner.)

Once published, a few things become true: (1) that publication cannot be undone—the record of this version of the Manifest will be on-chain forever readable by anyone with access to the chain; (2) anyone who reads the manifest may download this document and all of the chunks and Bloom filters, and (3) the publisher (us) has no further on-going costs other than pinning the files on IPFS (which carries a near-zero cost—less costly than hosting a regular website.

Over time, as more an more users use various portions of the index as described above, the number of copies of those portions increase, and because our users pin those portions by default the resiliency and speed of the system increases in direct proportion to the number of users. 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) that one might use to read these file. There are currently about 2,750 indivual chunks in the Ethereum mainnet index.

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

The GoLang structure for the file as a whole looks like this:

```
// The binary chunk file contains a singe header record and two
// arbitrarily related fixed-width tables of addresses relating
// to appearance records
type IndexChunk struct {
    Header           HeaderRecord
    AddressTable     []AddressRecord
    AppearanceTabls  []AppearanceRecord
}
```

The header record has the following fields:

```
// The first 44 bytes of the file containing versioning information
// and two counters detailing how many records are in the two
// fixed width tables.
type HeaderRecord struct {

    // 0xdeadbeef indicates a known file format
    MagicNumber  [4]bytes

    // The version string of this specification. This value
    // ensures that anyone receiving this file knows its
    // format and may therefor read the file
    Version      [32]bytes

    // 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 contains the number of addresses detailed in the header each with the following structure. The address table relates into the position of the appearances records in that table.

```
// For each address found in the block range represented by
// this chunk, this table stores the address and two integers.
// The first points to the offset in the appearance table
// where this address's appearance records begin. The second
// integer records the number of records.
type AddressRecord struct {

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

    // The offset into the appearance table for the address
    Offset  uint32

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

}
```

The appearance table records <blockNumber><tx_id> pairs for every address in two 32-bit integers.

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

    // The block number for the appearance
    BlockNumber      uint32

    // The transaction id for the appearance
    TransactionInfex uint32

}
```

Generally, the search algorigthms try to avoid reading this file. In fact, this is exactly the reason for the Bloom filters which allow us to much more quickly determine if the address appears in the chunk. But, if the Bloom hits, then we must search the chunk file for the address. Here, we also avoid reading the entire file into memory, choosing instead to memory map the file and conduct a binary search for the address. This algorithm is presented next. Error processing is squelched.

```
func getAppearances(addr, fn string) []AppearanceRecord {

    // Open the file for reading
    fp:= Open(fn)

    // Read the header - fp remains at start of address table
    header := ReadHeader(fp)

    // Where do the tables start?
    addrsTable := sizeof(header)
    appsTable := addrsTable + (header.nAddrs * sizeof(AddressRecord)

    // Conduct a binary search on the address table
    found := binary_search(addrsTable, appsTable, address, test)
    if !found {
        return []AppearanceRecord{}
    }

    // Seek to location in appearance table of offset
    fp.Seek(appsTable + found.Offset)

    // Read and return that many records
    apps := make([]AppearanceRecord, found.Count)
    fp.Read(appsTable + found.Offset, &apps)
    return apps
}
```

The Bloom Filter File

Some text

Some text

The Names Database File

The names databases is not part of the Unchained Index, per se, but it is useful. We publish the IPFS hash to the current names database as part of the manifest. This allows our end-user software to download the names database from IPFS and share it via pinning. This lowers our cost of publication for our software. On disc, the file is sorted by the Address field.

The names database is a fixed-width binary data file. The number of records in the file can be calculated by dividing the file's size in bytes by the width (in bytes) of each record.

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

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

A NameRecord is a single row in the names database table:

```
// A single row in the names database file
type NameRecord struct {
    // A user defined tag for this record
    Tags        [31]byte

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

    // A name given to this address
    Name        [121]byte

    // For ERC-20 tokens, the symbol for the token
    Symbol      [31]byte

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

    // An arbitrary description for the record
    Description [256]byte

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

    // An arbitrary bit array used for flags
    Flags       uint16
}
```

We leave it as an excersize for the reader to open and process this simple array.

The Timestamp Database File

The need for 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 GoLang structure for the binary timestamps database file is:

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

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

A single timestamp record has this format:

```
// 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 always true:

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

One may find the timestamp for a given block with:

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

One may find a block number given a timestamp using a binary search:

```
bn := binary_search(&ts,
                    timestampDB,
                    nRecords,
                    sizeof(TimestampRecord),
                    searchFunc)
```

We note that the timestamps database could have been half as big if we removed the block number (which is sequential and starts at with zero). We could have used the block number directly as an index into a 32-bit-width-record timestamp array. We choose, however, to include block number in the data as an aide in debugging and checking of the databases's integrity.

Building the Index and Bloom Filters

Definition: Address Appearances

In this section, we define an address appearance.

Appearances per Block

In this section we discuss how to extract all addresses found in a given block. This is the heart of the algorigthm. Because the list of appearances contained in the canonical blocks is independent of any other block, this can be made highly concurrent (and is in our code).

Extracting Addresses at High Speed

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

Purposefull Sloppiness

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

Baddresses

This section discusses the idea of a "badaddress," which are 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.

Conclusion

This section completes the discussion.

Querying the Index

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 to **chifra export <address>**.

Conclusion

A concluding paragraph.

Supplimentary Information

Website
Github repo
GitCoin grant
Tokenomics website
Docker version
Account Explorer