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 Contracts

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 is:

```
{
  "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 in the section called Building the Index and Bloom Filters. The Manifest is stored in a text file that is added to IPFS and the IPFS hash of that file is (periodically due to cost) published to the smart contract. (We publish the hash using the `trueblocks.eth` wallet which is also the contract's deployer.)

Once published, a number of important things become true:

1) The publication cannot be undone—the IPFS reference to that Manifest will be on-chain forever readable by anyone with access to the chain;
2) Anyone who reads the Manifest may download all the chunks *and* this specification which contains enough information to faithfully reproduce the chunks, if desired;

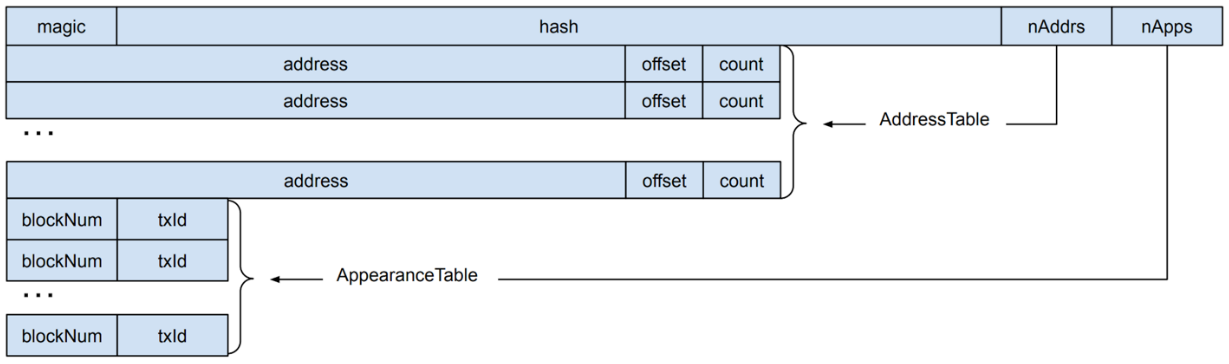
3) The publisher (that is, TrueBlocks) has no further on-going cost of publication other than pinning the files on IPFS (which carries a near-zero cost). We do not have to run a Web 2.0 API to deliver this data.

Over time, as more and more users download and pin more and more parts of the index—which happens by default through usage—the resiliency and speed of the system increases. This increase in resiliency is in proportion to the number of users. The more users—the more resilient. This is the essence of Web 3.0. It’s a classic case of positive sum externalities and “If we all build it, we can all come.”

The Index Chunk File

We describe the format of the index chunk file graphically as well as GoLang structures. Following that, we detail the algorithm one might use to read this file. There are approximately 2,800 chunks covering the 15,000,000 Ethereum mainnet blocks at the time of this writing and the same number of associated Bloom filters.

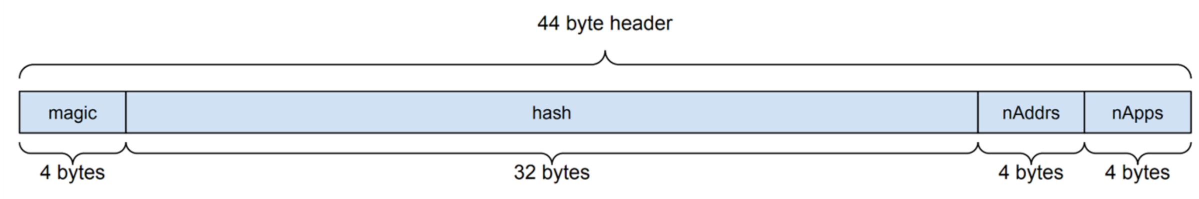
The binary Index Chunk file consists of a single fixed-width header followed by two fixed-width tables. Graphically, it looks like this:



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 rows
    AddressTable    []AddressRecord

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

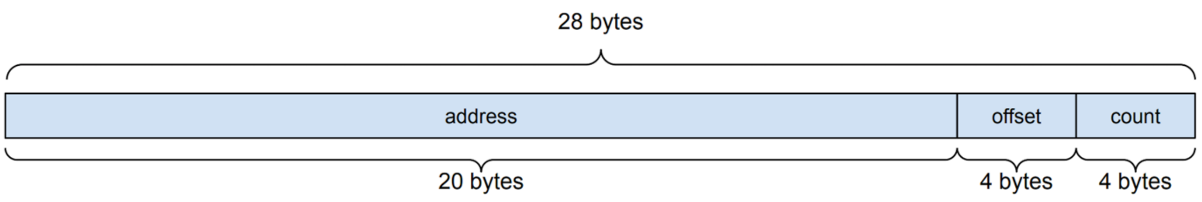
The HeaderRecord, which contains versioning information and counters for two tables, is expressed graphically as:



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

    // The version string of the specification used to create this file.
    // This value ensures that anyone receiving this file knows how to
    // read it. For all chunks up to and including block 13,000,000, the
    // value of this field contains all zeros. For chunks covering blocks
    // ranges 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 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 into the AppearanceTable via the Offset field which points to the position in the AppearanceTable of this address’s appearances.

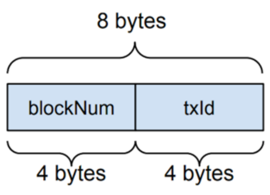


For each address found in the block range, the AddressRecord table stores the address and two unsigned integers. The first integer is the Offset into the AppearanceRecord table where this address’s appearance records begin. The second integer, Count, is the number of appearance records to read.

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. The 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 implementation of the function, it accepts an array of addresses as opposed to a single address at a time. We do this to avoid spinning through the chunks repeatedly for multiple addresses. In the actual implementation, we return a corresponding array of arrays of Appearances instead. We've removed that feature here to ease explanation.

Performance Note: Because the GetAppearances function reads from disc, it is desirable to avoid running the function if at all possible. Thus the reason why the system requires Bloom filters…

The Bloom Filter File

As described above, the Unchained Index is a collection of binary "chunks". Taken as a whole, the chunks can be seen as a normal, everyday database index file. Each chunk contains a near-equal number of records. In this way, we can ensure a fair distribution of the index via content-addressable storage, while imposing a minimal cost of publication on the publisher.

In order to optimize searching the chunked index, we produce a second binary file, called a Bloom filter, for each chunk.

A Bloom filter is a probabilistic data structure that encodes set membership in a very compact way. Each chunk is, on average, 25 megabytes big and contains approximately 2,000,000 appearance records. The associated Bloom filters are, on average, one megabyte big.

The algorithm used to build the Chunks and Bloom Filters is in the section Building the Index and Bloom Filters. In this section, we detail how to read the Bloom filter document the Bloom filter's file format.

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

We will start by describing the algorithm used to read the Bloom filter. After 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 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 it is not present (because the user used `chifra init` for example), the routine consults the Manifest in order to know which IPFS hash to download. The full binary chunk is then downloaded from IPFS.

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 addresses in the chunk. Each bit array is 1/8 of one megabyte wide (an arbitrarily chosen number of bits 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".

The word "Adaptive" refers to the fact that we grow the Bloom filter as we populate it making it as large as it needs to be to maintain a pre-define "maximum expected false positive rate." Different ranges of blocks contain a widely-varying number of addresses which means that if we did not use an adaptive Bloom filter the filters would be either under-populated or over-saturated—in fact, this is exactly what happened in the Ethereum node software where the Bloom filters are ineffective. We "grow" the filter by adding additional bit arrays 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 rate, we add an additional bit arrays 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 if the address is present. This promotes a more even distribution of the Chunks.

A Golang version of the Bloom filter 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 structures 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 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.

Summary

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 databases are not required for the operation 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 our end-user software to acquire the database without a third party. This lowers our cost of publication. The names database file is a binary file consisting of fixed-width records. The record count is easily calculated from the file's size.

The binary format of the names database follow. (**Note:** this format 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 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

In addition to the names database, a second ancillary data is defined in the Manifest. This is the Timestamps Database.

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

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

// The number of records can be calculated using the file's size
nRecords := fileSize(<path>) / 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 aide in debugging and checking of the databases's integrity. We reserve the right to change this format 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've defined the various binary file formats of the databases that make up the Unchained Index as well as a number of algorithms used to read and query the index.

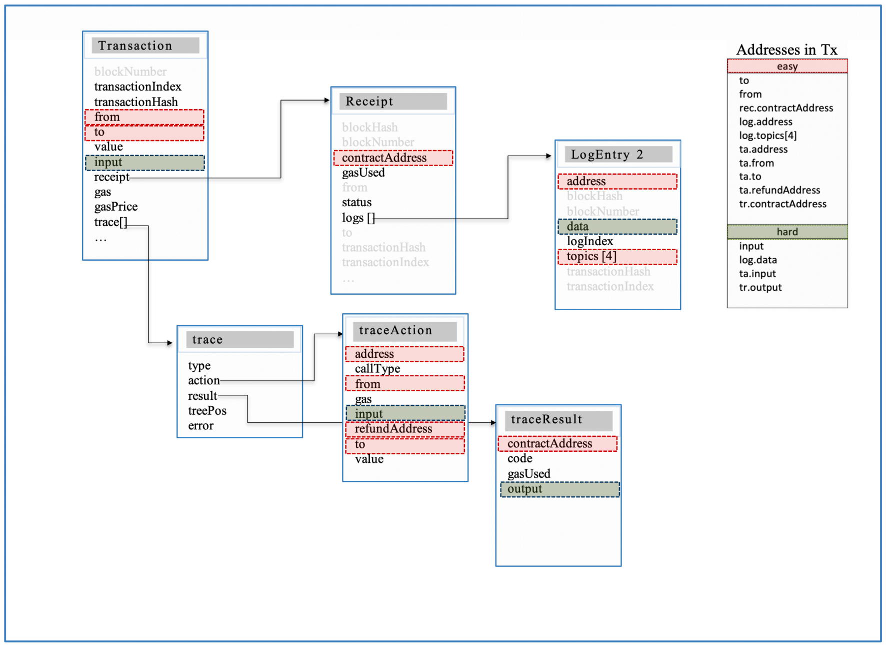
In this section we describe the algorithms used to create and publish the index. We start with a discussion of the primary data structure in the index: the AppearanceRecprd. We then proceed to discuss how we extract appearances from the blockchain data. We conclude by justifying a few engineering decisions we've made that were forced upon us due to our desire to preserve data immutability, reproducibility, and content-addressability.

Definition: Address Appearances

We define an *Appearance* broadly to mean any location in a blockchain's history where a particular byte pattern is encountered that "appears" to be an address. More specifically, and *Appearance* is a <blockNum, txId> pair.

Our algorithm is "purposefully sloppy" in identifying appearances (for reasons similar to the reason bloom filters permit false positives, but are able to return false if an item is not in the set). We would rather include a byte pattern that "appears" to be an address but is not, rather than not include an actual appearance. Unfortunately, it turns out that this is not even theoretically possible, which leads to the idea of "baddress" which is described below.

Looking closely at a typical Ethereum transaction, one may identify up to 16[1] different locations where an address may "appear."



[1] 14 locations are shown in the image, however, in actuality, the block data includes two additional locations—the uncle miner and the block miner. Thus, the total is 16 not 14 as shown.

The pink items shown above are what we would call “easy-to-find” or Explicit locations. These are locations in the data where the field is identified as being an address (such as, **to**, **from**, **miner**, and **log address**). These locations are trivial to extract.

Other obvious locations, such as certain logs topics, are easily inferred. For example, ERC20 Transfer and Approve logs (identifiable by topic) contain known locations for addresses. While this method may seem effective, it requires specific knowledge about the format of the topics. As new smart contract protocols appear, this type of specific knowledge must be maintained. Even if that were possible to maintain this knowledge (it’s not), this way of processing introduces an aspect that would necessarily destroy the long tail of unknown protocols. (Something we specifically want to avoid.)

Looking at other areas of the data (colored green above) one encounters the true reason why identifying appearances is difficult. In these areas of the data, which frequently contain actual appearances of addresses, the appearances are Implicit. These appearances are “obscured” inside of byte data and we do not, and cannot, know the format.

Our desire to capture as may appearances as we can forces our scraping process to operate with minimal “external knowledge” about the byte stream. While *it is not even theoretically possible* to identify every appearance in the general case (every 20-byte pattern from *0xfff..fff* to *0x000…000* is a valid address), we can take advantage of two aspects of the byte stream[2] that help us identify addresses:

1) In Ethereum, addresses are 20 bytes long and (for the most part) are left-padded with zeros to 32-bytes;

2) Most numeric values encountered in the byte stream are smaller than most addresses (this observation touches on the idea of a “baddress”, which is discussed below).

The above comments are obscure and require better explained. Please see the “Purposeful Sloppiness” section below. For the Implicit locations of the byte stream, we are forced to use certain heuristics when identifying appearances. That section explains those heuristics.

Notes on Per Block Data

Performance Note: We note that the appearance data buried in a given canonicalized block is independent of the appearance data contained in any other block. In the following discussion, 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 implementation.)

[2] Yes – this contradicts the previous comment mentioning that we want to avoid “outside knowledge” of the byte stream. We justify this choice by noting that this second version of “outside knowledge” is not relative to particular smart contract protocol such as ERC20—this outside knowledge is specific to the way the Ethereum blockchain itself operates. For this reason, it works across any protocol and thereby avoids destroying the long tail.

The processing described in this section of the paper is at the heart of the Unchained Index. The chunking algorithm requires per-block processing in order to build chunks. On top of this per-block processing is a process called the Consolidation Loop. We describe the process of consolidation after describing the per-block processing.

Performance Note: These algorithms are very aggressive in the way they query the node software. They only work if one is running a local node. This is for two reasons:

1) Most node providers do not support delivering Ethereum's trace data. Of those that do, the cost is prohibitive;

2) If you're using a remote node, you will almost certainly get rate limited even if you're paying for access;

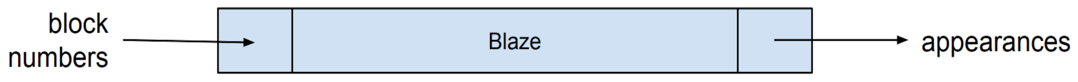
3) Why should you have to pay for access to globally available data?

We highly recommend running the node software called Erigon. It is fast at syncing to the chain, fast at delivering RPC requests locally, takes up a lot less hard drive space, and is easy to install and maintain.

Extracting Addresses Per Transaction Per Block

The following algorithm is contained in the Blaze section of our codebase. Blaze is a "pipeline" in the GoLang sense of the word (i.e., that is, it uses channels and is highly concurrent).

Blaze is dependent only on the node's RPC endpoints. The processing "shoves" block numbers into one end of the pipe and, as if by magic, AppearanceRecords emerge from the other end of the pipe.



While conceptually Blaze is a single pipeline, internally we need two primary channels. The first processes block numbers, queries the chain, and produces a stream of "traces and logs" for further processing by the second channel. The second considers (using the heuristics mentioned above) each byte of the byte stream parsing out appearances. The overall effect is pipe where block numbers go in and address appearances come out.

A discussion of Blaze starts on the next page.

At its topmost level, the Blaze algorithm begins by creating two channels:

```
// 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 as many go routines as the user tells us to create in order to process the data being inserted into the pipelines. Different blockchains have different characteristics, therefore the software allows the end-user to provide the number of processors on 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)
    }
```

The next step is to insert block numbers into the top of the pipeline. The caller specifies which blocks to start and stop at.

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

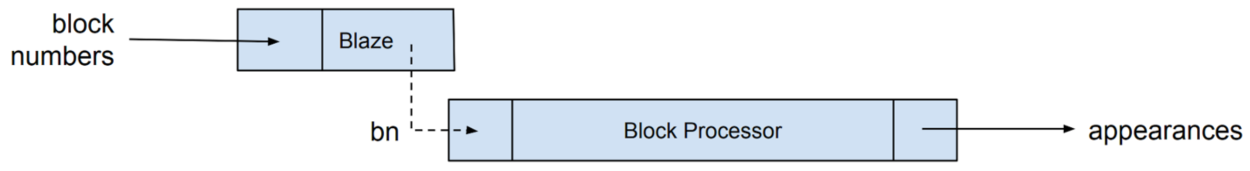
And finally, we close the channels (flushing them) and wait until the pipeline finishes processing. After Blaze completes, a file has been written containing all the AppearanceRecords found in this range. The Consolidation Loop post processes this data upon return.

```
    // 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 is called with a block range. It is the responsibility of caller to determine which block range to process. During regular processing, the caller 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 allows the algorithm to continue to operate on very small machines.

The Block Processor

Next, we look into the block processor (the **processBlocks** routine). The block processor queries the node for trace and log data given a block number. It then collates that data into a more useful structure (called ScrapedData) and "shoves" the collated data into the address processor (the **processAddresses** routine).



The block processor ranges over the block channel waiting until a block number appear. Once a block number appears, the block processor 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 is closed, the loop exits and the wait group is marked as done and the function return 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, and its trace data. 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.GetBlockTimestamp(bn)
            Traces: client.GetBlockTraces(bn)
            Logs: client.GetBlockLogs(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.

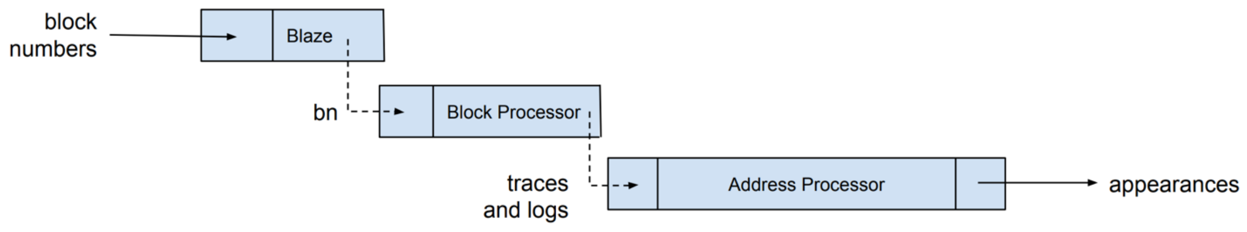
3) **Possible EIP:** It would be beneficial if the node software provided timestamp information with the **trace_block** response. This would eliminate one third of all queries against the node.

Next, we detail the ScrapedData structure (called "combined" in the above code). This structure combines the received information from the RPC commands (**trace_block** and **eth_getLogs**). This convenience structure 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 Address Processor

The block processor passes the above data on to 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 channel, addressChannel, and finds “every appearance of every address anywhere in that 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 block number, and a 32-bit integer representing the transaction index. In other words, an AppearanceRecord.

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

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

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

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

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

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

        // write the map (i.e., the Index Chunk) to the output medium
        writeAddresses(sc.blockNumber, addressMap)
    }
}
```

We describe the processing of trace and log data below.

eth_AddressesPerBlock

Possible EIP: 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 return by **trace_block** contains a field called *Type*. The *Type* field distinguishes between various types of trace data. For example, traces related to one smart contract calling another, traces related to self-destruct activity, traces related to mining. The address processor uses these trace types to figure out where to look for Explicit addresses which are very easy to find. It 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
)
```

The differing traces carry information that contains addresses appearances in differing locations. In some cases, those locations are Explicit—in the sense that these values are stored locations that are explicitly documented as being addresses. These addresses are "easily-found" (pink 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. 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 Result.Address	Explicit Explicit
CreateFailed	Receipt.ContractAddr	Explicit
Create	Input (contract byte data)	Implicit
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 does not contain addresses. 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 modulo 32-bytes). This field may contain Implicit appearances, but never Explicit appearances.

Location	Aspect
Address	Explicit
Topic[0]	-
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 literally 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 14 seconds, should be able to be accounted for automatically? Shouldn't tax reporting be trivial?

I know why it doesn't work—no other system extracts address appearances from the data locations we've identified above as Implicit. That's not quite true—they do—but they do so using ABI files which are sparsely available at best. This 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 included;

2) Over time add new smart contracts 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 this list from being "captured" and made "pay-to-play".

Hint: Keeping a list of known smart contracts does not work. We need a different solution.

In this section of the document 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 cannot exist. Every 20-byte string is a valid Ethereum address. For example, true, as a 20-byte string is 0x0000…0001. This is 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 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 3% more transactions than any other system we know of. This is enough to reconcile nearly every address we've tested. 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 we use to accomplish our task as best we can. (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 very large part of the Ethereum byte stream is aligned to 32-byte boundaries. This happens for various reasons and becomes quite handy;

2) Addresses, at least presently, are 20-bytes long. Because the data is frequently aligned on 32-byte boundaries and are left padded with zeros, this means an address must have 12 leading zeros. Any 32-byte chunk of Implicit data that does not start with 12 zeros is rejected as a potential address appearance.

3) Much of the Ethereum byte stream that contains numbers is four eight-byte integers packed in 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) Any address 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 (*How Accurate is TrueBlocks*) 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 15% of "missing" transactions, about 5% are material. Material meaning that one or more asset balances change. This is why other method find off-chain reconciliation so difficult. They are missing transactions. As you might surmise—it's not possible to reconcile if you're missing transactions.

While running the two routines detailed above (**extractFromTraces**, and **extractFromLogs**) our processing may encounter many "potential" locations for an address appearance (these are the locations labeled as Implicit above).

For any Explicit location, 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** (which is named to indicate that it is impossible to be sure if this is an address or not.)

That routine is detailed next:

```go
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 10000000000000000000000000000000000000000000000 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 := "00000000000000000000000000000000000000ffffffffffffffffffffffffff"
    //        -------+-------+-------+-------+-------+-------+-------+-------+
    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 an seemingly arbitrary set of heuristics to identify addresses. No—it’s not terribly future proof given that the packing algorithms that produce this data might change—unlikely they will change, but they may evolve. We acknowledge these shortcomings, but, we re-iterate this this method is “purposefully sloppy.” The task at hand is impossible—even in the theoretical sense—but this doesn’t stop us from producing data that can be reconciled 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—is not needed. The comes from a lack of a recognition by the node software engineers that the problem of identifying appearances even exists. This, at least, is a start to identify that problem.

Baddresses

We mention 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 (which is every 20-byte string), but which we choose to exclude from our index. We do this for consideration of the size of the index on disc. We want to be inclusive enough to find the great majority of addresses, but not so inclusive so as to explode the size of the index unnecessarily.

We identify a "baddress" as any 20-byte string that is less than "`0x000000000000000000000000000000000000ffffffffffffffffffffffff`" or ends with eight zeros ("`00000000`"). This is an arbitrarily selected definition.

The Consolidation Loop

This section discusses the consolidation phase of the algorighm, that is, how we determine to consolidate 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 process any new blocks that may have appeared since the process last ran. During the initial phase scraping a new chain, the process runs continually. 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, it can pick up the next time around at that point. The process calls the Blaze process with the pre-configured (or user supplied) number of blocks to process. Once Blaze is finished, the Consolidation Loop decide if it's time to create a chunk. This decision is different for each chain. On Ethereum 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 the Gnosis, this many records would take many days to produce. On some private chains, this many 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 than many blocks. Be careful though—this does not always work. During the 2016 dDos attack on Ethereum mainnet, many millions of appearances were created in very few blocks. This tends to skew the results which is why we break off after a certain number of appearances.

The consolidation loop follows the following trajectory:

```
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 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 is 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 also pins the new Index Chunk and its associated Bloom filter to IPFS, updates the Manifest, pins the Manifest to IPFS, and, finally, posts the IPFS hash of the Manifest to the smart contract.

Snap-to-Grid and Correcting Errors

We mention above a snap-to-grid configuration item. This value is present primarily to aide 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 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 re-produce the entire stream of chunks.

If, instead, we break the stream of chunks periodically at a snap-to-grid block, we can correct only that part of the broken data up until the next snap. On Ethereum 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 wouldn’t it 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 command line tool. While it does support a API server that matches all of the command line options through the API, it’s first and most important use case is as a command line tool to aide in data science. It is also designed, from first principals to be run locally—on the end user’s machine. It is possible to run 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. An 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 bottle neck becomes open 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.

Additional to the above another very important consideration is that we wanted to create and "naturally sharded and easily shared index." Breaking the index into chunks, inserting those chunks into IPFS, 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 own natural interest, the chunked Unchained Index is capable of delivering only those portions of the index the end user needs. And, once downloaded to the end user's machine—where the speed of local data become appearance—the end user can be "enlisted" in sharing the data through pinning by default.

For this any 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 curtesy 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 apprecated.

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 list <address>** as well as all the options available for this very powerful tool. We leave this section to be completed later.

Conclusion

We're done. Thanks for listening.

Supplementary Information

Website
GitHub repositories

- Core
- Explorer
- Documentation
- Docker Core
- Docker Monitors

GitCoin grant
Tokenomics

List of references:

- Adaptive Bloom Filters
- Bloom Filters from Wikipedia
- RPC Documentation
-