A Language-Agnostic Compression Framework for the Bitcoin Blockchain

Orestes Papanastassiou[1] and Alex Thomo[1]

[1] University of Victoria, BC, Canada
[2] {orestespap,thomo}@uvic.ca

Abstract. This research addresses the growing interdisciplinary interest in Bitcoin by proposing a versatile compression framework for transforming raw blockchain data into a streamlined compact format suitable for high-performance analysis. Our approach focuses on developing a language-agnostic API, ensuring accessibility across programming languages. Beyond data extraction, our framework outputs the Bitcoin user transaction graph, facilitating network analysis, forensics, and pattern detection. Processed data are exported to the HDF5 file format for compatibility with mainstream analysis tools. A proof-of-concept CPython implementation demonstrates the framework's feasibility, showcasing its real-world applicability for data-driven investigations in Bitcoin research.

Keywords: bitcoin · blockchain · cryptocurrencies · big data

1 Introduction

The Bitcoin blockchain[3] is a decentralized, distributed ledger that records Bitcoin transactions. This ledger is public and immutable, making it a valuable source of information for various research and analytical purposes. However, its sheer size presents significant challenges to researchers and analysts. The ability to extract, store, and analyze blockchain data efficiently is desired by academics and industry.

This research addresses a critical gap in the current landscape by proposing an abstract algorithmic framework for the extraction and compression of Bitcoin's transaction ledger and user transaction graph. The data are converted into a form that we call *normal form*, which adheres to a data-oriented design [3].

Data-oriented design is a design paradigm applied to algorithms and data structures to maximize the use of the CPU and GPU cache. This involves minimizing the size of the working set (program input, i.e. a matrix) to fit as much data as possible into the cache, using a contiguous memory layout whenever possible, and designing algorithms so that data reside in the cache until they are no longer needed by the program [8]. A cache line is the currency between the CPU and the main memory [5]. Reading from memory is a significant bottleneck for modern CPUs, thus high-performance data analysis necessitates fitting as much

[3] https://bitcoin.org/bitcoin.pdf

data as possible into consecutive cache lines so that the size of the subset of the working set retrieved every time the CPU reads data from main memory is maximized.

Our normal form makes it possible for the entire transaction ledger or user transaction graph to fit into the main memory of a 64GB commodity research machine, with $O(1)$ indexing of the underlying data for fast lookups. The framework facilitates the extraction of the raw blockchain data and transforms them into our normal form with a minimal memory footprint, such that the entire blockchain can be extracted on a commodity machine with just 32GB of memory.

To maintain the framework's language-agnostic theme end-to-end, output data are exported to the HDF5 file format [6], which can be processed by any mainstream programming language and data analysis software. Furthermore, the output can be seamlessly stored in a conventional SQL/No-SQL DBMS. We also present a proof-of-concept implementation in Python to demonstrate the feasibility and efficiency of our approach. We chose Python for our proof-of-concept implementation because of its interdisciplinary popularity and the challenges associated with implementing the framework efficiently using this language. The main challenge is CPython's inefficient memory consumption; the increased productivity and simplicity offered by its object-oriented and dynamically typed nature come with trade-offs.

Our contributions can be summarized as follows.

- **Algorithmic Framework/API:** We introduce a comprehensive language-agnostic algorithmic framework/API designed to efficiently extract and compress Bitcoin's transaction ledger and user transaction graph. This framework is crucial for handling the massive scale of blockchain data, which contains over 800 million transactions as of January 1st, 2023[4].
- **Cache Optimized Data Layout with O(1) Indexing:** Our *normal form* data structure facilitates $O(1)$ indexing for high-performance data analysis. The entire Bitcoin transaction ledger or user transaction graph can be accommodated in the main memory of a commodity research machine.
- **HDF5 Format for Cross-Platform Compatibility:** Output data are saved in the HDF5 file format, facilitating the use of the processed data with various mainstream programming languages and data analysis tools, and can be seamlessly stored in both SQL and No-SQL databases.
- **CPython Proof-of-Concept Implementation:** To validate our work, we present a CPython-based proof-of-concept that overcomes the language's memory hurdles.

2 Related Works

BlockSci [7] is a comprehensive state-of-the-art blockchain exploration tool implemented in C++ with a CPython interface for seamless high-level querying of

[4] `https://blockchain.com/explorer/charts/n-transactions-total`

the data. Similar to our work, BlockSci's data layout is designed to maximize the utility of the CPU cache. The project is no longer maintained, which makes it challenging to install and use because of compatibility issues with newer versions of Linux and CPython. Because it is a packaged querying tool, there is little flexibility in changing the set of transaction attributes and migrating the data to a different DBMS.

BTCSpark [11] is built on top of Apache Spark and Spark SQL, a robust and distributed data processing framework, to address the challenges associated with large-scale Bitcoin data analysis. This approach is particularly valuable in scenarios where the volume and complexity of Bitcoin data require efficient and seamless distributed processing.

BitSQL [9] is an SQLite and MariaDB-based blockchain querying tool. The data layout is similar to the one proposed by this work, but redundantly stores both raw transaction IDs/addresses and their integer hashes, leading to an output that is double in size compared to ours. Furthermore, our research dedicated a significant amount of time to integrating various SQL/NoSQL DBMSs into the pipeline and failed to replicate the claimed performance figures.

[2] contributes a general extraction framework for the Ethereum and Blockchain blockchains. The framework extracts raw blockchain data, as well as off-chain data (i.e. exchange rate, user IP addresses), and stores them in an SQL or NoSQL database. The purpose of this framework is to facilitate individual as well as cross-chain data analytics.

3 Background

3.1 Bitcoin transactions

The Bitcoin blockchain records transactions in a sequence of blocks, created by miners in a peer-to-peer network. Miners compile transactions into blocks, earning new Bitcoins and fees as rewards.

A Bitcoin transaction consists of an arbitrary number of inputs and outputs. The currency of the Bitcoin blockchain is the set of unspent transaction outputs. Each transaction output specifies a value of Bitcoins to be spent and contains a script (*ScriptPubKey*) dictating the conditions under which these Bitcoins can be spent. An unspent output is spent when referenced by a transaction input alongside the appropriate unlocking script (*scriptSig*). Bitcoin users/entities are an abstraction of output script addresses.

In a valid transaction, the sum of the Bitcoin value of the inputs must be greater than or equal to the sum of the value of the outputs - users cannot spend more money than they have. When the input value is greater than the output value, miners can claim the remainder as a fee. If the total input value is greater than the debt that needs to be settled, the remaining Bitcoins can be reclaimed by appending an output(s) to the transaction's output set referencing the desired amount and address. This is known as a *change* output.

The first transaction in a block is called *Coinbase* and can be used by miners to claim the newly minted Bitcoins and any aggregated transaction fees. *Coinbase* outputs create new Bitcoins, and thus increase the money supply.

3.2 User transaction graph and address clustering

A directed graph $G = (V, E)$ consists of vertices V and directed edges E, where each edge points from one vertex to another. The blockchain can be represented as a directed multigraph, where directed edges represent transaction outputs and vertices/nodes represent users. A multigraph is a graph that can have multiple edges between a pair of endpoints.

Bitcoin users are inferred via address clustering. The *common-input-ownership*[5] heuristic is the most widely used address clustering method (CIO henceforth) The heuristic explicitly assumes that a multi-input transaction is signed (via the *ScriptSig*) by the same user, even though it is technically possible for each input to be signed by a separate user [4]. CIO is a computationally cheap algorithm for address cluster inference and is supported by our framework.

3.3 Notations

We list some recurring notations throughout the paper. Whenever we use *list* the order matters, and whenever we use *set* the order does not matter.

1. $S(.)$ is the size of a data structure/type in bytes.
2. h: Block height. This is a block's ordered position (index) in the blockchain.
3. B_h: List of blocks at height h. A block contains a list of transactions.
4. T_h: List of transactions at height h, all the transactions in blocks in B_h.
5. A_h: Address set at height h, all the addresses in transactions in T_h.
6. U_h: Sub-list of T_h containing all the transactions of T_h with at least one unspent output.
7. b: Block : $b \in B_h$.
   (a) $b.h$: Height of block b.
   (b) $b.t$: Transaction list of block b; $b.t \subset T_h$.
      i. $b.t[i]$: i^{th} transaction in $b.t$.
8. t: Transaction.
   (a) $t.id$: Unique ID of the transaction.
   (b) $t.b$: index of t in block b; $0 \leq t.b < |b.t|$.
   (c) $t.h$: Transaction's block height; $0 \leq t.h < |B_h|$.
   (d) $t.norm$: Transaction's *normal form*.
   (e) $t.in$: List of transaction inputs (see below for the definition of inputs).
   (f) $t.out$: List of transaction outputs (see below for the definition of outputs).
   (g) $t.uout$: List of transaction unspent outputs.
   (h) $t.ts$: Transaction timestamp.
   (i) $t.insum$: Sum of inputs' referenced unspent output values. Each input references an unspent output from an older transaction.

[5] `https://en.bitcoin.it/wiki/Common-input-ownership_heuristic`

   i. $t.insum.f$: Denomination flag.
   ii. $t.insum.v$: Monetary value (Bitcoin or Satoshi, depending on f).
   (j) $t.outsum$: Sum of a transaction's output values. Same attributes as $t.insum$.
   (k) $t.fee$: Fee, if $t.insum > t.outsum$.
   (l) $t.norm$: normal form, list of integers (see Methodology).
9. in: Input.
   (a) $in.i$: i^{th} input in $t.in$.
   (b) $in.id$: Referenced output's transaction ID ($t^i.id$, $t^i \neq t$).
   (c) $in.outi$: Referenced output's position in its own transaction's (t^i) output list ($t^i.out[in.outi]$).
   (d) $in.addr$: Input address.
10. out: Output, $uout$: Unspent output.
   (a) $out.addr$: Output address.
   (b) $out.f$: Denomination flag.
   (c) $out.v$: Output Bitcoin value.
11. G_h: User transaction graph at height h.
   (a) V_h: Node set.
   (b) E_h: Edge set.

4 Methodology

4.1 Normal form, denomination flags, and MurmurHash

We transform raw Bitcoin transaction data into a list of unsigned 4-byte integers that we call *normal form*. This list contains input/output addresses, output values, sum of input values, and timestamp. The pipeline can be easily modified to include other attributes, such as output script type.

Let $|t.in| = n$ and $|t.out| = m$. The normal form structure is as follows:

1. The first two elements are n and m.
2. The following n elements are the *MurmurHash*-ed input addresses in the order they appear in the transaction.
3. The following m elements are the *MurmurHash*-ed output addresses ".
4. The following $2 \cdot m$ elements are $(flag, value)$ pairs, one pair per output address. $flag$ encodes $value$'s denomination (Satoshi or Bitcoin).
5. The last three elements are a $(flag, value)$ pair encoding the total monetary value of the inputs ($t.insum$), and transaction timestamp.

Transaction outputs are denominated in Satoshis ($1\ BTC = 100e6\ SAT$) on the blockchain. Outputs greater than $2^{32} - 1\ SAT$ exceed the limit of a 4-byte unsigned integer (*uint32*). For such large Satoshi values, we convert them into Bitcoin, round them to the fourth decimal, and scale them back to an integer. If still too large, the offset from $2^{32} - 1$ is stored. Negative fees due to rounding errors are set to zero.

Denomination flags indicate the transaction value's format and spent status. This facilitates the construction of the unspent transaction output set ($UTXO$).

Flags $(0, 3)$ represent SAT, $(1, 4)$ BTC, and $(2, 5)$ BTC as an offset from $2^{32}-1$. Unspent output flags are < 3; when an unspent output is spent, it is marked as spent by incrementing its flag by three. To convert back into SAT, for $flag \in (1, 4)$ multiply the output value by $1e4$, and for $flag \in (2, 5)$ add $2^{32} - 1$ and multiply by $10e4$.

Raw Bitcoin transaction IDs (*txid*) are 64-character long HEX strings. Addresses are *base58Check* encoded alphanumeric strings of 58 characters [1]. *txid* ($t.id$) can be represented as a HEX string of 64 characters (32 bytes) and an address can be represented as a 58-byte array of ASCII characters. A raw Bitcoin address and transaction ID occupy an entire cache line and half a cache line respectively (a cache line is usually 64 bytes). Our solution uses the *MurmurHash*[6] non-cryptographic hash function to convert addresses and transaction IDs into 4-byte unsigned integers.

4.2 Transaction indexing

Normal-form transactions are stored in a one-dimensional array of unsigned integers (tArr). To facilitate efficient traversal of the transaction array and $O(1)$ lookups we introduce two auxiliary offset arrays; toff and boff. toff maps an ordered transaction index (i) to the slice of tArr that contains ordered transaction i.

$$\forall i \in [1, |T_h|) : \text{tArr[toff[i]:toff[i+1]}) = T_h[i].norm \tag{1}$$

The i-th blockchain transaction's normal form is stored in the range [toff[i],toff[i+1]) of tArr. Similarly, boff enables transaction indexing by block height. Given a block height h, we can retrieve the block's transaction set from tArr as follows,

$$\forall h \in [1, |B_h|) : \text{tArr[toff[boff[h]] : toff[boff[h+1]]}) = B_h[h].t \tag{2}$$

We can also index transactions by block height and transaction block position $(t.h, t.b)$. For $a = \text{toff[boff[t.h]+t.b]}$, $b = \text{toff[boff[t.h]+t.b+1]}$,

$$tArr[a : b) = B_h[t.h].t[t.b].norm \tag{3}$$

4.3 Retrieving input addresses

The first stage parses the raw data and maps transactions to normal form. Let $|B_h| = N$ and $\forall b \in B_h : |b.t| = M$, we have the following intermediate representation,

$$\text{tArr} = [B_h[0].t[0].norm, \ldots, B_h[N-1].t[M-1].norm] \tag{4}$$

$$= [T_h[0].norm, \ldots, T_h[(N \cdot M) - 1].norm] \tag{5}$$

In our CPython implementation, we store tArr across multiple arrays/files on $5e6$ transaction intervals because of memory limitations. In the intermediate

[6] `https://github.com/aappleby/smhasher`

representation, inputs are represented as a triplet of integers compressed into one integer between stages one and two. For now, assume that each input is represented as an integer triplet (a set of three integers). For every $t \in T_h$ we have,

$$\forall i \in [0, |t.inp|) : (t^i.h, t^i.b, t.in[i].outi) \wedge (t^i \neq t) \tag{6}$$

The items in the triplet refer to the referenced output's transaction block height, position in the block, and the output's position in t^i respectively. This is used in the next stage to extract $in.addr, in.f$ and $in.v$. Given an arbitrary transaction id as input, the corresponding transaction can be retrieved in $O(1)$ via,

$$UMap_h(id) \rightarrow [t.h, t.b, |t.uout|],\ t = B_h[t.h].t[t.b] \tag{7}$$

$UMap_h$ is a key-value data structure that maps a transaction ID to a triplet containing the transaction's block height, position in the block, and number of unspent outputs. When a transaction is added to $UMap$, $|t.uout|$=$|t.out|$. When a future $in \in t'.in$ references $t.uout$, $|t.uout|$ is decremented by one. Every time block data are moved to secondary storage, $UMap$ is updated such that every t with $t.uout \in \varnothing$ is removed. The blockchain's $UTXO$ set at some height h can be reconstructed via $UMap_h$.

4.4 UMap as a CPython *dict*

As of v3.12, CPython's *dict* is based on a dynamic indexing array and a compact hash table[7]. Each entry in the hash table is represented by a hash (h) ($\leq$ 8-byte signed integer,), and two pointers (p) for the key-value (k, v) pair (8 bytes/-pointer). An entry's position in the hash table is stored in the dynamic array as an $\leq$ 8-byte unsigned integer (i).

In general, a map data structure's memory footprint is lower bounded by the size of its payload and upper bounded by the payload times the resizing factor[8]. For large payloads, the resizing factor for a Python dictionary is two, which we can use to approximate an upper bound. The following formula calculates a dictionary's payload ($P(k, v, i, h)$) size (n is the number of (k, v) pairs),

$$S(P) = n \cdot (S(k) + S(v) + 2 \cdot S(ptr) + S(h) + S(i)) \tag{8}$$

$$S(P(k, v, uint32, int64)) = n \cdot (S(k) + S(v) + 28) \tag{9}$$

We estimate a *dict*'s memory footprint to fall within this range,

$$S(P) \leq S(dict) \leq 2 \cdot S(P) \tag{10}$$

Considering the current size restrictions of the Bitcoin protocol $|b.t|$ and $|t.out|$ rarely exceed $10e4$, and is virtually impossible to exceed $1e6$. The average $|b.t|$ for the past six years has hovered around $2e3$, briefly reaching a peak of

[7] `https://mail.python.org/pipermail/python-dev/2012-December/123028.html`
[8] `https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf`

$\approx 3.4e3$ before reverting to the mean[9]. The record $|t.in|$ and $|t.out|$ stands at $20e3$ and $13e3$ respectively[10]. As of January 1st, 2023, $|B_h| \approx 770e3$, so $b.h$ can be represented by a seven-digit integer to accommodate block heights $< 1e7$.

EncodeT transforms a $(t.h, t.b, j)$ triplet tr into a 22-digit unsigned integer. *DecodeT* is the inverse map,

$$EncodeT(tr) \rightarrow ((1e7 + t.h) \cdot 1e7 + t.b) \cdot 1e7 + j \tag{11}$$

$$DecodeT(int) \rightarrow (t.h, t.b, j),\ int \in N \tag{12}$$

Where, $t.h = (int)\ div\ (1e14) - 1e7$, $t.b = ((int)\ mod\ (1e14))\ div\ (1e7)$, and $j = (int)\ mod\ (1e6)$. Depending on the context, j can represent one of three attributes. If tr describes an in, then $j \rightarrow in.i$ or $in.outi$, if tr describes an out, then $j \rightarrow out.i$, and if t is meant to keep track of $|t.uout|$, $j = |t.uout|$. The maximum value in *EncodeT*'s range is interpreted as $t.h = 9999999$, and $t.b = j = 999999$. Any unsigned integer in this range is represented as a 36-byte *int* instance in CPython, whereas an implementation representing the triplet as a *tuple* of three *int* objects requires 148 bytes of memory. As a result, our encoding consumes $\approx 50\%$ less memory resources than a *tuple*-based representation.

4.5 Handling hash collisions

Unambiguous resolution of hash collisions in $UMap$ leads to garbage output data. To avoid ambiguity, we leverage that the chronological traversal of transactions guarantees that previous transaction inputs do not reference the currently processed transaction's ID.

To add $t.id$ to $UMap$, we *MurmurHash* the ID, check the hash's membership in $UMap$'s keyset, and if the result is positive there is a hash collision. This means that an older transaction ID is mapped to the same hash. In this case, we add the raw transaction ID to $UMap$'s keyset, represented as a string of 64 HEX characters. A low hash collision rate means that the memory footprint of string keys is virtually zero. For any arbitrary transaction t' referenced by some input in another transaction t, we check the membership of $t'.id$ in $UMap.keys$. If it exists, t' is represented by $t'.id$ in $UMap$, and if it doesn't, it is represented by the *MurmurHash* of $t'.id$.

5 Algorithms

5.1 First Stage: Transforming raw transaction data into normal form

The pipeline begins by traversing the raw transaction data to convert them into normal form and create the (toff, boff, tArr) arrays and $UMap$ in the process. Each transaction t is temporarily stored in an array (*tempt*) that is unpacked

[9] `https://blockchain.com/explorer/charts/n-transactions-per-block`
[10] `https://coinmetrics.io/batching/`

into tArr before moving on to the next one. For each transaction input (in), $in.id$ (the equivalent of $t'.id$) is looked up in $UMap.keys$ according to the procedure described above.

The desired $(t'.h, t'.b, |t'.uout|)$ triplet is stored as an integer in $UMap[key]$, and is unpacked via $outT = DecodeT(UMap[key])$. The input is now represented by $EncodeT((outT[0], outT[1], in.outi))$, which is used in the second stage to retrieve the input's address and referenced output's Bitcoin value, both of which are stored in the referenced output. As per our pruning optimization, $UMap[key]$ ($|t'.uout|$) is decremented by one.

Next, each output address in $t.out$ is hashed and appended to $tempt$, followed by each output's $(out.f, out.v)$ pairs. The last three elements in the array are $t.insum.f = 0, t.insum.v = 0, t.ts$. t is added to the unspent transaction set, represented by $UMap$. The algorithm checks $t.id$'s hash membership in $UMap.keys$ to determine whether there is a hash collision or not. It then encodes the desired triplet $inInt = Encode(t.h, t.b, |t.uout|)$ and adds a new entry to $UMap$, $UMap[key] = inInt$. Depending on whether there is a hash collision or not, key is either $t.id$ or $t.id$'s hash.

5.2 Second stage: Resolving *uout* references

The list of files produced by the first stage is traversed to retrieve $in.addr$ and $t.insum$. The latter is necessary for the calculation of t.fee. If an input references an unspent output stored in another file (file$'$), the input is stored in a map ($lookupMap$) alongside other inputs with out-of-file references, so that their data are retrieved after tArr is traversed. The map is then grouped by file ID so that $uout$ references located in the same file are grouped, and thus each file is loaded once. Map keys are file IDs, pointing to (in, out) sequences. Both in and out are encoded as $EncodeT((t.h, t.b, j))$.

Each transaction (t) input is decoded to extract the corresponding $uout$ reference, represented by the compressed $(t'.h, t'.b, out.i)$ triplet. If $t'.h$ is in the file's block height range, then t' is in the same file. The referenced output (out) is retrieved from tArr using the toff and boff offset arrays, the corresponding $uout$ reference in $t.in$ is replaced by $out.addr$, $t.insum.v$ is incremented by $out.v$, $t.insum.f$ is assigned the appropriate flag, and $out.f$ is incremented by three to mark it as spent. If $t'.h$ is not in the file's block height range, then the input and the referenced output are appended to $lookupMap[t.h]$, and the corresponding $uout$ reference in $t.in$ is set to NULL. Once all transactions in the file are traversed, the missing data are retrieved by traversing $lookupMap$ and loading the appropriate files.

5.3 Third stage: Address clustering

This stage applies the CIO heuristic to the address set (A_h), implemented based on the Weighted Quick Union Find algorithm [12,13]. [10] implements CIO using a variation of *Union-Find*.

WQUF consists of two core operations; *Find* and *Union*. If an arbitrary element belongs to a set, *Find* returns the set's root, and if it doesn't, it returns the empty set operator. *Union* merges two distinct sets by attaching the root of the smaller set to the root of the larger set. It then increments the size of the larger set by the size of the smaller set to reflect the size of the expanded set and finally returns the root of the larger set.

5.4 Fourth stage: User transaction graph

For every $t \in T_h$, the cluster address (set root) of $t.in$ is retrieved via $parentMap[t.in[0].addr]$, and an outgoing edge is connected to each output cluster ($\forall out \in t.out : parentMap[out.addr]$). This process leads to the creation of $|t.out|$ number of edges. $parentMap$ maps a MurmurHash-ed Bitcoin address to its cluster address (set root).

A dummy *Coinbase* node (user) is added to the graph to represent transfers of newly minted Bitcoins and transaction fees. For each t, if $t.\text{f}ee > 0$, an edge is created between the input cluster and *Coinbase*, weighted by the transaction fee. The graph is implemented by a key-value data structure, where each key is a cluster ID that points to a list of outgoing edges,

$$edge = (parentMap[out.addr], out.\text{f}, out.v, t.ts) \tag{13}$$

6 Results

Experiments	**Small**	**Large**
Height range $h \in [0, H]$	$[0, 3e5]$	$[0, 769842]$
$\lvert A_H \rvert$	$3.733e7$	$1.084e9$
$\lvert TX_H \rvert$	$40e6$	$792e6$
CPU	i7-3770	Xeon E5-2620
DRAM	12GB	128GB
DIMMs	2x4GB, 2x2GB	8x16GB
L1	256KiB	384KiB
L2	1MiB	1.5MiB
L3	-	15MiB

Table 1. The first column lists experiment parameters, the second column represents Experiment Small, and the third Experiment Large.

6.1 Compression

We define compression rate as the percentage data size reduction by the compressed data (a) relative to the uncompressed data (b),

$$CR(a, b) = (1 - (S(a)/S(b))) \cdot 1e2 \tag{14}$$

The unclustered blockchain's (u) HDF5 output consists of tArr, toff, and $UMap$ (key-value pairs stored in a 1D array) and the clustered blockchain's (c) HDF5 output consists of the same arrays plus $parentMap$. In Experiment Large, the observed $S(u)$ (minus $UMap$), $S(c)$ (minus $UMap$, $parentMap$), $S(UMap)$, and $S(parentMap)$ are 53.8, 44.5, 1.2, and 6.3 GB respectively. With $S(Bitcoin)$ $\approx$ 446 GB on January 1st 2023, our normal form yields $CR(u, Bictoin) \approx 88\%$.

The graph ($\hat{G}_H$) is stored in three arrays; *nodes* contains cluster addresses, *edges* contains the outgoing edges for each cluster, and eoff contains the *edges* offsets for each cluster (identical to tArr and toff). The observed $S(\hat{G}_H)$ (minus $UMap$, $parentMap$) in B is $\approx$ 43 GB, with 4 GB evenly divided between *nodes* and eoff arrays, and 39 GB for *edges*. The total size (with the two maps) is $\approx$ 47.3 GB.

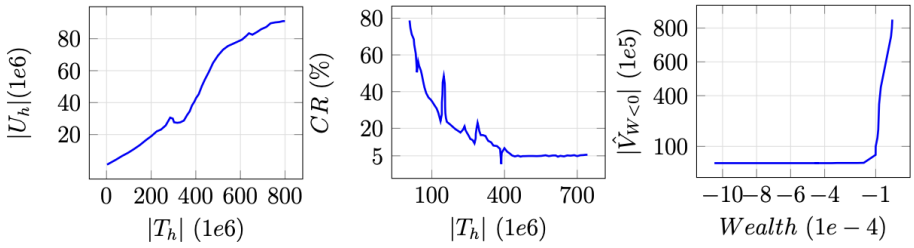

Fig. 1. [Left]: X and Y-axis represent the sizes of the transaction set ($|T_h|$) and unspent transaction set ($|U_h|$) respectively, as h grows to H. [Middle]: The graph shows the observed compression rate from pruning $UMap$ ($CR(UMap', UMap)$) as a function of the size of the transaction set $|T_h|$, as h grows to H. [Right]: Cumulative Distribution Function of negative wealth. The X-axis represents negative wealth values and the y-axis node count.

6.2 Memory

$UMap$ and $parentMap$ are the data structures with the largest memory footprint in the API. An alternative solution to *dict* would be a key-value pair DB, in which case a sample of the key-value pairs reside in the cache (DRAM in this case) and the rest in storage. We tried two state-of-the-art solutions, *RocksDB* and *Redis*, and in both cases, elapsed time increased at least three-fold. Approximately half of the queries are cache misses, and retrieving data from storage is orders of magnitude slower than main memory. The decision to store intermediate data ([toff,boff,tArr]) in *pickle* files has a similar reasoning. We tried two RDBMs, *MySQL* and *SQLite*, and elapsed time increased at least $\approx$ three-fold.

Fig. 6.1 [left] shows that $|U_h| \approx |T_h|/10$ throughout the experiment. For $n = |U_H| = |T_H|/10 \approx 90e6$, $S(P)$ yields 8.7GB, thus our estimate is $S(UMap_H) \in [8.7, 17.4]$ GB. The observed $S(UMap_H)$ is 13.1 GB, $\approx$50% greater than our lower bound, and $\approx$ to the median value of our estimated size range. As of

October 2023, $|T_{h=H}| \approx 900e6$, so our pruning optimization makes it possible to extract 90% of the unclustered transaction data with just 32GB of DRAM.

$UMap$ is pruned every time the extracted block data are stored in secondary storage and removed from the cache. Data are dumped in storage on $5e6$ transaction intervals, thus Fig. 6.1 [middle] contains 160 such measurements. The pruning optimization's returns are diminishing as h, and therefore $|T_h|$, increases. For $|T_h| \geq 400e6$, the achieved compression rate stays at 5%. At $p = (400e6, 40e6)$, $UMap$ is pruned an additional 80 times. However, had the pruning stopped at p, $|UMap.keys_{h=H}|$ would have grown to $40e6 \cdot (1.05)^{80} \approx 200e6$.

$parentMap.keys$ is the equivalent of A_H. To estimate its size at the end of B, we set $n = |A_H| \approx 1e9$ and get $S(parent) \in [92, 184]$ GB. The observed $S(parent)$ is 115GB, approximately 25% larger than our lower bound, and 17% less than the median value of our estimated size range.

6.3 Elapsed time

Experiment timings	**Small**	**Large**	t_L/t_S
Normal Form	2h 45min	67h 50m	25
UOUT References	12min	68h	340
Address clustering	2.5min	2h 30m	60
Graph creation	7min	3h 30m	30
Data export	3min	1h 15m	25
Total	3h 10min	5d 23h	

Table 2. First column lists experiment parameters, second and third columns list the results in Experiment Small and Large respectively, and fourth column lists Experiment Large/ Experiment Small timing ratios.

Table 2 lists the timings for each stage. Experiment Large's working set is $\approx$ 20 times larger than Experiment Small's, and since the framework's complexity is $O(n)$, we should expect timings in Large to be at least 20 times greater than Small. The empirical average $t_L/t_S \approx 45$ reflects the difference in processing power between the two machines and the growing number of CPU cache misses in Experiment Large due to the much larger working set (and an unexpected issue with stage two). We suspect that a critical factor in stage one's poor performance could be the third-party library used to read the raw blockchain data[11]. A main priority for our future work is further investigating the potential bottlenecks caused by this API.

Stage two's poor performance can be attributed to its worst complexity case turning out to be its average/expected complexity due to the inherently random nature of unspent output references. The worst case is that each file has at least one *uout* reference for each previous file. That proved to be the case for most of

[11] `https://github.com/alecalve/python-bitcoin-blockchain-parser`

our block files. As a result, most files had to access all previous files to retrieve input data, equivalent to performing a file scan each time.

6.4 Information loss from floating point arithmetic

Floating point arithmetic approximation error inevitably leads to nodes with negative wealth, spending more Bitcoins than they receive. We classify a user transaction graph $\hat{G}_h$ as a *valid economy* if it satisfies the following constraints:

1. $\forall n \in (\hat{V}_h - \{CB\}) : \hat{Wealth}_h(n) \gtrapprox 0$
   (a) $\hat{Wealth}_h(n) = inVal_h(n) - outVal_h(n)$
2. $Supply_h \approx \hat{Supply}_h$
   (a) $\hat{Supply}_h = outVal_h(CB) - inVal_h(CB)$
3. $Supply_h \approx Sum(\hat{Wealth}_h(n) \ \forall n \in \hat{V}_h)$

Where $inVal_h$ and $outVal_h$ represent the total value of a node's incoming and outgoing edges at h respectively. $Supply_h$ represents the Bitcoin supply in the blockchain at h, and $\hat{Supply}_h$ is the supply in $\hat{G}_h$. $\hat{Wealth}_h(n)$ represents individual node wealth, where the real precise value is unknowable. CB represents the Coinbase node. The first constraint states that a node cannot spend more Bitcoins than it owns. The following constraints can be deduced from the first one and all constraints are relaxed, therefore a node's wealth can be *slightly* negative.

Experiment Large	$G_{h=H}$	$\hat{G}_{h=H}$
$Supply_H$	19.249e6	19.237e6
$\lvert V_{H_{W<0}}\rvert / \lvert V_H\rvert$	0	0.1416
$min(Wealth_H)$	0	$-129e-3$
$Supply_H - Sum(Wealth_H)$	0	17.28

Table 3. First column lists the parameters of the experiment, second column lists the ground truth data (the blockchain), and third column lists the observed values.

Table 3 lists the necessary data to prove that our $\hat{G}_H$ satisfies all three constraints. All measurements are made at the end of Experiment Large, at height $h = H = 769,842$. The observed $\hat{Supply}_H / Supply_H = 99.94\%$ means that the Bitcoin supply in $\hat{G}_H$, based on $outVal_H(CB) - inVal_H(CB)$, reflects the real supply at height H. $\hat{G}_H$ then satisfies the second constraint.

The second statistic measures the proportion of $|\hat{V}_H|$ with negative wealth ($v \in \hat{V}_H : \hat{Wealth}_H(v) < 0$). Given that $|\hat{V}_H| \approx 490e6$, and $|\hat{V}_{H_{W<0}}| / |\hat{V}_H| \approx 14\%$, $\approx 69e6$ nodes in the graph have a negative net worth. This does not invalidate our user graph; Fig. 6.1 [right] shows the cumulative distribution function of negative user wealth at $h = H$, where 99% of negative $Wealth$ values fall in the $[1e-4, 0)$ range. The most negative observed $Wealth$ value is ≈ -0.13 Bitcoins. $\hat{G}_H$ then satisfies the first constraint.

The last statistic measures the difference between the supply in $\hat{G}_H$ and total user wealth. The value of the real figure is zero because user wealth cannot exceed $Supply_H$. In $\hat{G}_H$, this figure is ≈ 17 Bitcoins. In relative terms $(17/Supply_H) \approx 8.83e-57$, so the observed figure is very close to zero. $\hat{G}_H$ satisfies the last constraint and thus qualifies as a *valid economy*.

References

1. Andreas M. Antonopoulos. *Mastering Bitcoin*, pages 55–88. O'Reilly Media, Inc., 2 edition, 2017.
2. Massimo Bartoletti, Stefano Lande, Livio Pompianu, and Andrea Bracciali. A general framework for blockchain analytics. In *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pages 1–6. Association for Computing Machinery, 2017.
3. J. D. Bayliss. *The Data-Oriented Design Process for Game Development*, volume 55, pages 31–38. IEEE Computer Society, 2022.
4. Damiano Di Francesco Maesa, Andrea Marino, and Laura Ricci. Data-driven analysis of bitcoin properties: exploiting the users graph. *International Journal of Data Science and Analytics*, 6:63–80, 2018.
5. Ulrich Drepper. What every programmer should know about memory. `https://people.freebsd.org/~lstewart/articles/cpumemory.pdf`, 2007. Accessed: 2023-11-09.
6. Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, pages 36–47. Association for Computing Machinery, 2011.
7. Harry Kalodner, Malte Möser, Kevin Lee, Steven Goldfeder, Martin Plattner, Alishah Chator, and Arvind Narayanan. {BlockSci}: Design and applications of a blockchain analysis platform. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2721–2738. USENIX Association, 2020.
8. Markus Kowarschik and Christian Weiß. *An overview of cache optimization techniques and cache-aware numerical algorithms*, pages 213–232. Springer, 2003.
9. Hyunsu Mun and Youngseok Lee. Bitsql: A sql-based bitcoin analysis system. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–8. IEEE, 2022.
10. Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers 17*, pages 6–24. Springer, 2013.
11. Jeremy Rubin. Btcspark: Scalable analysis of the bitcoin blockchain using spark. `https://rubin.io/public/pdfs/s897report.pdf`, 2015. Accessed: 2023-11-09.
12. Robert Sedgewick and Kevin Wayne. *Algorithms*, pages 216–233. Addison-Wesley Professional, 4 edition, 2011.
13. Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.