A Language-Agnostic Compression Framework for the Bitcoin Blockchain

by

Orestes Papanastassiou
B.Sc., University of Macedonia, 2021

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER'S OF SCIENCE

in the Department of Computer Science

© Orestes Papanastassiou, 2023
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

A Language-Agnostic Compression Framework for the Bitcoin Blockchain

by

Orestes Papanastassiou
B.Sc., University of Macedonia, 2021

Supervisory Committee

Dr. A. Thomo, Supervisor
(Department of Computer Science)

Dr. S. Chester, Member
(Department of Computer Science)

ABSTRACT

The surging interdisciplinary interest in Bitcoin within both academic and enterprise realms underscores the need for a versatile framework that can efficiently transform the raw Bitcoin blockchain data into a streamlined format suitable for high-performance data analysis. This research proposes an abstract framework designed to convert this data into a compact, normal form, facilitating its utilization across various programming languages and data analysis tools.

Our approach centers on the development of a highly efficient, language-agnostic Application Programming Interface (API). This API is designed to be implementable by any Turing-complete programming language, ensuring broad accessibility and usability. Beyond mere data extraction, our framework extends its capabilities to assemble the Bitcoin user transaction graph, a fundamental resource for downstream analysis, including network analysis, forensics, and pattern detection.

To ensure compatibility and ease of integration with an array of programming languages and data analysis software, we export the processed data to the language-agnostic HDF5 file format, recognized and supported by mainstream data analysis tools. This strategic choice empowers researchers and analysts to harness the power of Bitcoin blockchain data without being constrained by software dependencies.

Furthermore, to demonstrate the practicality and efficiency of our proposed framework, we present a fully functional CPython implementation as a compelling proof-of-concept. This implementation showcases the feasibility and real-world applicability of our solution, opening doors to a wide range of data-driven investigations and applications within the realm of Bitcoin research.

As the interest in Bitcoin continues to evolve and expand, our research offers a comprehensive solution to the challenges of handling and analyzing the vast data included in the blockchain. By providing an accessible, language-agnostic extraction and compression framework, we contribute to the democratization of Bitcoin blockchain data analysis, enabling both novices and experts to uncover valuable insights and drive innovations within this field.

Contents

Supervisory Committee **ii**

Abstract **iii**

Contents **iv**

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**

2 Related Works **5**

3 Background **8**

3.1 Bitcoin blockchain 8
3.2 Bitcoin transactions 10
3.3 User transaction graph 11
3.4 Address Clustering 11
3.5 Notations 12

4 Methodology **14**

4.1 Normal form 14
4.2 Denomination flags 15
4.3 Representing Bitcoin transaction IDs and addresses efficiently 19
4.4 Minimizing the size of the normal form 19
4.5 Transaction indexing 21
4.6 Retrieving input addresses 22
4.7 UMap in CPython 25
4.7.1 Python *dict* and its memory footprint 25

4.7.2 Python *int* and the NumPy library 26
4.7.3 Handling hash collisions 27
4.7.4 Trivial UMap 29
4.7.5 Optimized UMap 30
4.8 UMap in a statically typed environment 32
4.9 Address clustering 32
4.10 Storing $UMap$ and $parentMap$ efficiently 34

5 Algorithms **36**
5.1 First Stage: Transforming raw transaction data into normal form . . 36
5.2 Second stage: Resolving *uout* references 40
5.3 Third stage: Address clustering 44
5.4 Fourth stage: Bitcoin user transaction graph 46
5.5 Fifth Stage: Exporting data to HDF5 48

6 Results **53**
6.1 Compression 54
6.2 Memory 56
6.3 Elapsed time 58
6.4 Information loss from floating point arithmetic 59

7 Conclusions **63**

Bibliography **65**

List of Tables

Table 6.1 Experiment Setup. The first column lists experiment parameters, the second column lists the results from the small experiment and the third column lists the results from the large experiment. . . 53

Table 6.2 First column lists the parameters of the elapsed time experiment, second and third columns list the results in Experiment Small and Large respectively (single thread/process), fourth column lists Experiment Large/ Experiment Small timing ratios. 58

Table 6.3 First column lists the parameters of the experiment, second column lists the ground truth data, and third column lists the observed values. 61

List of Figures

Figure 6.1 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. . . 55

Figure 6.2 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. 56

Figure 6.3 Cumulative Distribution Function of negative wealth. The X-axis represents negative wealth values and the y-axis node count. 60

Chapter 1

Introduction

The advent of cryptocurrencies has ushered in a transformative era in the finance and technology industry. Bitcoin[1], the pioneering cryptocurrency, has gained remarkable recognition and adoption in recent years, drawing the attention of both academic researchers and industry experts. Its underlying technology, the blockchain, represents a decentralized and secure ledger system that has the potential to disrupt various industries beyond the realm of digital currencies. As the interest in Bitcoin grows, so does the demand for effective methods to handle and analyze the blockchain.

The Bitcoin blockchain is a decentralized, distributed ledger that records every transaction ever conducted with Bitcoin. 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 need to extract, store, and analyze blockchain data efficiently has become a pressing issue in the interdisciplinary academic and enterprise domains.

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 is converted into a form that we call *normal form*, which adheres to a data-oriented design [2].

Data-oriented design is a design paradigm applied to algorithms and data structures to maximize the use of the CPU cache (and GPU cache when the GPU is used by a program). This involves minimizing the size of the working set (the input being processed by a program, i.e. a matrix) to fit as much data as possible into the cache, using a contiguous memory layout as much as possible, as well as designing algorithms so that data used in a loop stays in the cache until it is reused[3].

A cache line is the currency between the CPU and the main memory[4]. Reading

from memory is a significant bottleneck for modern CPUs, thus high-performance data analysis necessitates fitting as much 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 (until Jan. 1st 2023) or user transaction graph to fit into the main memory of a commodity research machine (64 GB), with $O(1)$ indexing of the underlying data for fast lookups.

We introduce an efficient, language-agnostic API that can be implemented by any Turing-complete programming language. The framework facilitates the extraction of the raw blockchain data and transforms them into our normal form with a minimal memory footprint, such that commodity research machines with no more than 32GB of memory can extract and convert the entire Bitcoin blockchain, which as of January 1st, 2023 consists of $800e6$ transactions[1].

To maintain the framework's language-agnostic theme end-to-end, output data are exported to the HDF5 file format[5], which can be processed by any mainstream programming language and data analysis software. Furthermore, even though the core design principle of the output data is high-performance data analysis, they can also 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 because of its popularity amongst researchers without a computer science background and the challenges associated with implementing the framework efficiently using this language. Python's reference implementation, CPython[2], is a dynamically typed object-oriented programming language suitable for automating non-performance-sensitive tasks, as well as efficient data analysis and machine learning via C-wrapped libraries such as NumPy[3] and PyTorch[4].

CPython can be thought of as a high-level C API that abstracts lower-level programming tasks, such as memory management and type checking, with a great selection of packages for data analysis and machine learning. The trade-off is slower performance and excessive memory consumption.

Everything in Python is an object, which boosts productivity but simultaneously means that even a short unsigned integer (integers in the range $[0, 2^{16})$) is represented

[1] `https://blockchain.com/explorer/charts/n-transactions-total`
[2] `https://github.com/python/cpython`
[3] `https://numpy.org/doc/stable/`
[4] `https://pytorch.org/docs/stable/index.html`

by a 28-byte instance of the *int* class. To put this into perspective, in a statically typed environment (i.e. Java, C, C++, Rust), a short unsigned integer is represented by 2 bytes, and the cache line of an average CPU is 64 bytes.

As a result, implementing the framework in CPython such that it can extract the entire blockchain on a commodity research machine (32-128GB of main memory) is challenging.

Our contributions can be summarized as follows.

- **Algorithmic Framework:** We introduce a comprehensive algorithmic framework 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 includes over 800 million transactions as of January 1st, 2023.

- **'*Normal Form*' Data Layout:** Our *normal form* data structure, adhering to data-oriented design principles, is accompanied by one-dimensional offset arrays that facilitate $O(1)$ indexing for instantaneous data access. This layout enables the entire Bitcoin transaction ledger or user transaction graph to be accommodated in the main memory of a standard research machine, typically equipped with 64GB of RAM.

- **Language-Agnostic API:** The framework's versatile API can be implemented in any Turing-complete programming language.

- **Minimal Memory Footprint:** Our methodology permits the processing of the entire Bitcoin blockchain, with a significant transaction count of 800 million, on commonly available research machines with 32GB to 128GB of RAM, effectively addressing the challenges of large-scale data analysis. The extraction and transformation of the blockchain into our normal form is possible on machines with no more than 16GB of memory.

- **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. This format is integral for cross-disciplinary accessibility and can be seamlessly stored in both SQL and No-SQL databases.

- **Python Proof-of-Concept Implementation:** To validate our framework, we present a Python-based proof-of-concept that overcomes the language's performance hurdles, accompanied by relevant quantitative experiments. Our implementation efficiently manages memory to handle the full blockchain dataset within the constraints of a standard research computer's memory specifications.

Chapter 2

Related Works

Blockchain extraction. The existing body of research (cf. [6, 7, 8, 9]) has predominantly offered solutions that expose high-level interfaces to query data stored in traditional Database Management Systems (DBMS). These approaches, while valuable, often tether users to specific language and DBMS environments, limiting their universality and flexibility.

BlockSci[6] is a comprehensive state-of-the-art blockchain exploration tool implemented in C++ with a CPython interface for seamless high-level querying of 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 (i.e. selecting the desired subset of attributes that the user wishes to extract) as well as migrating the data to a different DBMS.

BTCSpark[7] is built on top of Apache Spark[1] 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. The framework revolves around Apache Spark and thus poses challenges for applications outside the Spark environment.

BitSQL[8] is an SQLite and MariaDB-based blockchain querying tool. The researchers use SQLite to build the database, and MariaDB to query (read and write) the data. The data layout is similar to the one proposed by this work, but redun-

[1] `https://spark.apache.org/documentation.html`

dantly 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.

[9] 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 rates, user IP addresses), and stores them in an SQL or No-SQL database. The purpose of this framework is to facilitate individual as well as cross-chain data analytics. They also present their own open-source Scala library as an implementation of their framework. The framework extracts the raw data and stores them in a MongoDB collection (similar to an SQL table), where they are stored as Documents (set of key-value pairs, similar to an SQL table record). It is a great data analysis tool when performance is not a concern.

In contrast, our work introduces a novel approach that seeks to break down these language and data storage barriers by offering an efficient, language-agnostic API for Bitcoin transaction and user transaction graph extraction. This framework is designed to seamlessly integrate with any Turing-complete programming language, providing users with the freedom to work with their tool of choice, both for implementing and tailoring the framework to their specific needs, as well as analyzing the transaction ledger and/or user transaction graph. Moreover, the proposed data layout is not confined to a single storage format. The output data can be stored and queried either using our in-memory flat unsigned integer array format, with constant-time indexing, or an SQL/NoSQL DBMS.

In the rest of this section we describe works about clustering blockchain addresses. These are needed to create the final graph but are orthogonal to our main approach of extracting and compressing the raw Bitcoin blockchain data.

Blockchain user transaction graph. Approximating and analyzing the user transaction graph of the Bitcoin blockchain is an actively researched topic because of its utility in simulating the underlying economy as well as monitoring user activity. (cf.[10, 11, 12, 13, 14, 15]).

In [10], the authors present a graph creation algorithm that approximates the real user transaction graph using the *Common-Input-Ownership* heuristic[2], first mentioned in the Bitcoin white paper[1] and extensively covered in [16]. The paper

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

presents findings about the user graph's properties, such as the clustering coefficient, distribution of outgoing/incoming edges, and wealth distribution amongst nodes.

[11] expands on previous work related to address clustering by proposing a probabilistic clustering model (log-likelihood maximization). Their algorithm considers both on-chain (transaction properties) and off-chain information (i.e. information from Bitcoin forums).

[12] utilizes the user transaction graph to predict the market price of Bitcoin. Their approximation of the user graph is also based on the *Common-Input-Ownership* heuristic. They train and test a selection of statistical models (i.e. Linear Regression, Support Vector Machines, Neural Networks) for price prediction.

[13] expands on previous work related to address clustering. They introduce algorithms for validating and expanding the outputs of widely adopted clustering heuristics. The authors also contribute a set of algorithms for tracking a *peel chain*[3], which is an illicit Bitcoin laundering technique.

[15] introduces and evaluates a new Bitcoin address clustering heuristic that expands on the mainstream change-output[4] heuristic first proposed in [17].

[3] https://www.fraudinvestigation.net/cryptocurrency/tracing/peel-chain
[4] https://en.bitcoin.it/wiki/Change

Chapter 3

Background

3.1 Bitcoin blockchain

The Bitcoin blockchain[1] is an append-only record of transactions organized in blocks. Network participants broadcast transactions over the P2P network, which in turn are picked up and grouped into blocks by miners.

Miners are network nodes that create and broadcast transaction blocks. Every time a block is created the protocol rewards the miner with newly minted Bitcoins and any aggregated transaction fees. A node creating a block resembles a miner extracting valuable materials from the ground, hence the moniker.

Miners are incentivized to select transactions with the highest fee per megabyte. The protocol currently restricts block size to a theoretical maximum of 4MB, hence a rational agent picks a set of transactions that maximizes their earnings subject to the block size constraint.

The protocol relies on *Proof-of-Work*[18] for immutability and (eventual) network consensus. A distributed network satisfying eventual consistency[19] guarantees that if data are no longer updated (i.e. miners stop appending blocks to the blockchain), all network participants will, at some arbitrary point in the future, have the same view of the data.

Bitcoin satisfies eventual consistency by accepting the longest chain as the valid version of the blockchain; that is the chain with the greatest sum of proof-of-work. For a miner node to create a new block, the node has to hash the raw data of a block's header, the result of the hash being a random integer number, and if that number is less than or equal to the network's current *target* (an arbitrary integer number), then

the miner can prove to the network that they invested effort in the creation of this block - proof-of-work.

The block header contains, amongst other things, a reference to its parent/previous block unique id (block hash), thus expanding the chain, and a *nonce*[1], which is a random 32-bit integer used to produce a block hash that is less than or equal to the target. Proof-of-work allows a trustless network of nodes to prove that they performed a certain amount of work. It is fair for miners because the probability of finding a nonce (through trial-and-error) that makes block header data hash to a number less than or equal to a target is uniform, hence all participants have the same odds of guessing such a number.

Lemma 1. *Let A and B be two arbitrary integers that are target candidates and a hash function H that maps an arbitrary input to a random integer. If $A > B$, then it is more difficult to find a hash below B than it is for A.*

Proof.

$$A, B \in N \tag{3.1}$$

$$S1 = \{0, 1, 2, \ldots, A\} \tag{3.2}$$

$$S2 = \{0, 1, 2, \ldots B\} \tag{3.3}$$

$$H \colon X \to Y \tag{3.4}$$

$$A > B \to |S1| > |S2| \to P(H(block) \leq A) > P(H(block) \leq B) \tag{3.5}$$

□

Hence, the *smaller* the target, the more *difficult* it is for a miner to create a block, so more work is invested in the creation of a block when the target is *small*. This unambiguous definition of *work* allows network participants to reach a consensus regarding the state of the chain. When nodes are presented with the same set of candidate valid chains, PoW guarantees that they will all pick the same chain as the longest chain.

Validators are network nodes that verify transactions and blocks broadcast by other participants and relay verified valid data so that the rest of the network updates its version of the blockchain. Examples of conditions that qualify blocks as valid or invalid are their nonce and grouped transactions. For instance, if a user spends more money than they have then the corresponding transaction is invalid.

[1] `https://en.bitcoin.it/wiki/Nonce`

3.2 Bitcoin transactions

A Bitcoin transaction consists 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*) specifying the conditions under which the associated Bitcoins can be spent.

ScriptPubKey can be thought of as a *lock* because it allows only people with the right key to spend the associated Bitcoins. An unspent output is spent when referenced in a transaction input alongside the appropriate unlocking script (*scriptSig*).

Bitcoin scripts solve *proof-of-ownership* in a trustless distributed network using digital signatures[20]. When a transaction is broadcast over the network, participants can verify the ownership claim based on the validity of the provided signature.

Conventional output scripts specify a cryptographic public key (which can be thought of as a username) associated with a private key (password). A valid signature is a function of the referenced public key and the correct private key that proves knowledge of the correct private key without revealing it.

A Bitcoin script[2] can be completely arbitrary in its conditions, but conventional output scripts specify a cryptographic public key (can be thought of as a username) that is associated with a cryptographic private key (can be thought of as a password). Accordingly, an input script that references a conventional output provides a digital signature that is a cryptographic computation based on the referenced output, the public key referenced in the output, and the private key, so that it cannot be recycled and proves knowledge of the private key associated with the corresponding public key, without revealing the former.

In blockchain research, Bitcoin users/entities are an abstraction of output script addresses. For instance, *Pay-To-Public-Key* is a locking script type that requires the unlocking script to provide a digital signature for a specific public key. The public key is considered the *receiver*'s address. A *sender* is represented by an unspent output referenced in a transaction input. The referenced output contains the corresponding address.

Transactions consist of an arbitrary number of inputs and outputs. 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

[2] `https://en.bitcoin.it/wiki/Script`

remainder as a fee. Transactions with no fee (input value equals output value) are less likely to be added to the blockchain in a timely fashion because they lack the monetary incentive necessary for miners to pick them up.

If the total input value is greater than the debt that needs to be settled, the remaining Bitcoins can be reclaimed by the *sender* by appending an (or multiple) output 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*[3] 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 nominal money supply.

3.3 User transaction graph

A foundational part of blockchain research is the analysis of Bitcoin's user transaction graph. A directed graph can be mathematically defined as an ordered pair $G = (V, E)$, where V is a set of vertices and E is a set of edges (ordered pairs of vertices). The rigorous definition only allows edges between distinct vertices, though a more relaxed definition allows self-loops too - an edge with the same endpoints.

The transaction-ledger representation makes it difficult to analyze the underlying economy and transaction network. Transforming raw data into a graph enables the application of useful graph analysis methods to the blockchain.

By allowing output script addresses to represent users, the blockchain can be transformed into 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.

3.4 Address Clustering

As of 2023, there are over a billion distinct addresses in the blockchain[4]. That does not mean that $\approx 1/8$th of the planet uses Bitcoin. A single user can control multiple addresses, as is mostly the case. A Bitcoin user is an abstraction of a finite set of output script addresses and thus compresses the Bitcoin address space.

[3] `https://en.bitcoin.it/wiki/Coinbase`
[4] `https://blockchain.com/explorer/charts/n-unique-addresses`

Users are inferred via address clustering, which is an actively researched area of blockchain analysis. 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 by the same user, even though it is technically possible for each input to be signed by a separate user. CIO is a computationally cheap algorithm widely regarded as a good rule-of-thumb to infer address clusters.

3.5 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. h: Block height. This is a block's ordered position (index) in the blockchain.
2. B_h: List of blocks at height h. A block contains a list of transactions.
3. T_h: List of transactions at height h, all the transactions in blocks in B_h.
4. A_h: Address set at height h, all the addresses in transactions in T_h.
5. U_h: Sub-list of T_h containing all the transactions of T_h with at least one unspent output.
6. 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 $b.t$
7. 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*, list of integers (see Methodology)
   (e) $t.in$: List of transaction inputs (see below for the definition of inputs)

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

(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[6]

(i) $t.insum$: Sum of inputs' referenced unspent output values. Each input references an output from a transaction t' that has not been spent yet.

  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 $t.insum > t.outsum$

8. in: Input

(a) $in.i$: i^{th} input in $t.in$

(b) $in.id$: Referenced output's own transaction ID ($in.id \neq t.id$)

(c) $in.outi$: Referenced output's position in its own transaction's (t') output list ($t'.out[in.outi]$)

(d) $in.addr$: Input address

9. out: Output, $uout$: Unspent output

(a) $out.addr$: Output address

(b) $out.f$: Denomination flag

(c) $out.v$: Output value (denominated in Bitcoin or Satoshi)

10. G_h: User transaction graph at height h

(a) V_h: Node set

(b) E_h: Edge set

[6] https://en.wikipedia.org/wiki/Timestamp

Chapter 4

Methodology

We present our algorithmic framework alongside a CPython implementation. The framework is a multi-stage pipeline with each stage transforming the data appropriately and feeding them to the next as input.

4.1 Normal form

We transform raw Bitcoin transaction data into an ordered set of unsigned 4-byte integers that we call *normal form*. This set contains input/output addresses, output values, sum of input values, and timestamp. The pipeline can be easily modified to include other attributes too, such as output script type. This representation offers a structured way to capture all essential attributes of a Bitcoin transaction in a concise format.

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 in the structure are the input addresses in the order they appear in the original transaction.
3. The following m elements are the output addresses in the order they appear in the original transaction.
4. The following $2 \cdot m$ elements are $(flag, value)$ pairs, one pair for each 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.

6. $|t.norm|(n, m)$ can be calculated using the formula $n + 3m + 5$.

7. Storage requirement of a normal t, $S(t.norm)$, is four bytes times its element count, $|t.norm|(n, m)$.

Summarizing all the above we have,

$$\begin{aligned} t.norm = (&n, m, t.in[0].addr, \ldots, t.in[n-1].addr, \\ &t.out[0].addr, \ldots, t.out[m-1].addr, \\ &t.out[0].f, t.out[0].v, \ldots, \\ &t.out[m-1].f, t.out[m-1].v, \\ &t.insum.f, t.insum.v, t.ts,) \end{aligned} \tag{4.1}$$

$$|t.norm|(n, m) = n + 3m + 5 \tag{4.2}$$

$$S(t.norm) = 4 \cdot |t.norm|(n, m) \tag{4.3}$$

Where $S(.)$ is the size of a data structure/type in bytes. The purpose of denomination flags preceding each $out.v$ and $t.insum.v$ is to indicate whether values are denominated in Bitcoin or Satoshi.

4.2 Denomination flags

Raw outputs in the blockchain are denominated in a unit called Satoshi, where $1\,BTC$ (Bitcoin) = $100e6\,SAT$ (Satoshis). Satoshis are non-negative integers, thus they can be stored in an unsigned integer data type.

However, output values greater than or equal to $2^{32}\,SAT$ cannot be stored in an unsigned four-byte (32 bits) integer, (*uint32* henceforth) because the latter can represent numbers in $[0, 2^{32})$. To avoid using 8-byte unsigned integers, we convert such large Satoshi values into Bitcoin, round them to the fourth decimal, and then

multiply by $1e4$ to cast them back to integer format. In rare cases where even this encoding cannot fit in a *uint32*, we store the value's offset (difference) from $2^{32} - 1$, which is *uint32*'s maximum value.

Given the three possible denominations described above, a flag can take one of six values:

- $(0, 3)$: SAT
- $(1, 4)$: BTC
- $(2, 5)$: BTC, expressed as offset from $2^{32} - 1$

The reason each denomination can take two values instead of one is that it also encodes an output's binary *spent* state. This facilitates the construction of the unspent transaction output set ($UTXO$), which contains Bitcoin's underlying currency, the set of transaction outputs that have not been spent yet[1].

If a denomination flag is less than three, the corresponding output value is not spent. When an output is spent, its denomination flag is incremented by three. This mechanism ensures that Bitcoin transaction values are stored both efficiently and flexibly.

We contribute the two following algorithms. The first one implements the Satoshi to flag encoding, and the second one implements the inverse (flag to Satoshi).

[1] `https://en.wikipedia.org/wiki/Unspent_transaction_output`

Algorithm 1 Converting Satoshi into a flag denomination

1: **FUNCTION** satoshiToFlag
2: **Input:** value (output value in Satoshi)
3: **Output:** flag, value
4: $flag = 0$
5: **if** $value < 2^{32}$ **then**
6: **return** $flag, value$
7: $flag = 1$
8: $value/ = 100e6$
9: //round value to fourth decimal
10: $value = round(value, 4)$
11: $value\cdot = 10e4$
12: **if** $value \geq 2^{32}$ **then**
13: $value- = (2^{32} - 1)$
14: $flag = 2$
15: **return** $flag, value$

Algorithm 2 Converting a flag-denominated output value into Satoshi

1: **FUNCTION** toSatoshi
2: **Input:** flag, value
3: **Output:** value (in Satoshi)
4: **if** $flag \in \{0, 3\}$ **then**
5: **return** $value$
6: **else if** $flag \in \{1, 4\}$ **then**
7: **return** $value \cdot 1e4$
8: **else**
9: **return** $(value + (2^{32} - 1)) \cdot 1e4$

Consider the following toy transaction t,

- $|t.in| = |t.out| = 3$
- $t.out[0].v : 50,000,099,876,540$ SAT
- $t.out[1].v : 4,312,344,000$ SAT
- $t.out[2].v : 50,000,000$ SAT
- $t.insum.v : 50,004,512,216,540$ SAT

We convert the raw Satoshi values as follows,

1. $satoshiToFlag(t.out[0].v) \rightarrow (2, 705042704)$
2. $satoshiToFlag(t.out[1].v) \rightarrow (1, 431234)$
3. $satoshiToFlag(t.out[2].v) \rightarrow (0, 50e6)$
4. $satoshiToFlag(t.insum.v) \rightarrow (2, 705483927)$

$t.norm$ is as follows,

$$[3, 3, t.in[0].addr, t.in[1].addr, t.in[2].addr, \tag{4.4}$$

$$t.out[0].addr, t.out[1].addr, t.out[2].addr, \tag{4.5}$$

$$2, 705042704, 1, 431234, 0, 50e6, 2, 705483927, \tag{4.6}$$

$$t.ts] \tag{4.7}$$

We can calculate $t.fee$ as follows,

1. $outsumSAT = sum(\forall i \in [0, |t.out|) \; toSatoshi(t.out[i].f, t.out[i].v)$
2. $insumSAT = toSatoshi(t.insum.f, t.insum.v)$
3. fee.f, fee.v $= satoshiToFlag(insumSAT - outsumSAT)$

Where fee.f $= 0$ and $fee.v = 0.4987$ BTC.

The real fee is 0.5 Bitcoins, which is very close to the value $t.insum.v - t.outsum.v$. The 0.0013 rounding error between the true value and our approximation is a small price to pay compared to using long integers, which would double the storage requirements for our integer representation - *no free lunch*.

Rounding errors can cause $t.outsum.v > t.insum.v$, and as a result $t.$fee < 0, which means that either the real fee is zero or very close to zero. We note that $t.outsum.v$ is not stored in the normal form but can be computed by summing the values of all the outputs in the transaction

When such a case is encountered during the analysis of the data, the transaction fee can be considered to be zero. In *Results* we show that rounding errors have negligible impact on the quality of the output data.

4.3 Representing Bitcoin transaction IDs and addresses efficiently

Raw Bitcoin transaction IDs (*txid*) are 64-character long HEX strings. The hexadecimal system represents numbers using a base of sixteen, so each HEX character occupies four bits/half a byte ($2^4 = 16$). Bitcoin addresses are *base58Check* encoded alphanumeric strings of 58 characters[21][2].

txid ($t.id$) can either be represented as a HEX string of 64 characters, which is 32 bytes (i.e. *uint32* array, 8 hex chars/integer). A Bitcoin address can be represented as a 58-byte array of ASCII characters.

Both encodings are cache inefficient; a single Bitcoin address takes up an entire cache line, while a single transaction ID occupies half a cache line. Furthermore, our normal form uses an unsigned integer representation, therefore addresses must be converted into integers to adhere to the the normal form's representation.

Our solution uses the *MurmurHash*[3] non-cryptographic hash function to convert addresses and transaction IDs into 4-byte unsigned integers. *MurmurHash*'s underlying algorithm is computationally efficient with a relatively low hash collision rate.

A 4-byte unsigned integer consists of 32 bits, therefore the theoretical probability of two arbitrary inputs being mapped to the same value is $1/2^{32} \approx 0$, assuming that the hash function (MurmurHash in this case) is perfectly random. The observed probability (hash collision rate) is between 0.5 and 1%.

4.4 Minimizing the size of the normal form

We can rigorously prove that our normal form using 4-byte unsigned integers requires less space than using 8-byte unsigned integers, even though the latter eliminates the need for denomination flags, and thus the element count would be smaller by $m + 1$ integers (m output flags, 1 input sum flag).

Let $|t.norm|'(n, m)$ denote the element count of a transaction with n inputs and m outputs represented in normal form using 8-byte unsigned integers,

$$|t.norm|'(n, m) = |t.norm|(n, m) - (m + 1) = n + 2 \cdot m + 4 \tag{4.8}$$

[2] https://en.bitcoin.it/wiki/Base58Check_encoding
[3] https://github.com/aappleby/smhasher

$$S(|t.norm|'(n,m)) = 8 \cdot |t.norm|'(n,m) \tag{4.9}$$

For $n, m \geq 1$ we have,

$$S(|t.norm|(n,m)) < S(|t.norm|'(n,m)) \Leftrightarrow n + m > -3, \forall n, m \tag{4.10}$$

Thus, regardless of the size of n or m, the 4-byte unsigned integer normal form takes less space than the 8-byte one does, even though the former needs $m+1$ more elements than the latter.

We further support our design choice by comparing the average size of the normal form of both encodings. Plugging the empirical average number of transaction inputs and outputs, which happens to be 3 for both as of January 2023[4], into $S(|t.norm|)$ and $S(|t.norm|')$ yields 68 and 104 bytes respectively.

Taking into account that the size of the average CPU cache line is 64 bytes, on average, our encoding occupies ≈ 1.063 cache lines per transaction, while the 8-byte encoding needs ≈ 1.63. Given $(n = m) \rightarrow (S(|t.norm|(n,m))/S(|t.norm|'(n,m)) \approx 0.65$, we can conclude that our encoding needs $\approx 2/3$ of the space required by the 8-byte one. By appending timestamps to the end of the normal form (bytes $64 - 67$, for $n = m = 3$), an average transaction's input/output address sets, output/input sum values, and denomination flags can all fit into one cache line.

To provide further context, if we think of a transaction in normal form as a C *struct* of 4-byte unsigned integers, and the transaction array as an array of said *structs*, no padding is necessary within a transaction or between transactions. Every *struct* element has the same size, 4-bytes, so the compiler does not need to add padding between *struct* elements to make it or the array cache-aligned.

Cache lines are atomic, thus our 1.063 cache lines/transactions is the equivalent of reading two cache lines, which is the same as the 8-byte encoding. As a result, both encodings on average read two cache lines per transaction. However, the size of the working set is 35% smaller, which means that more transactions can reside in the cache using our encoding than the 8-byte one. As a practical example, consider a 64KB L1 data cache; using our encoding, on average, 941 transactions can fit in the cache, while only 615 8-byte encoded transactions can fit in the same cache.

Furthermore, if the transaction timestamp is not an important transaction attribute for analysis, then the working set can be further compressed by storing timestamps in a separate smaller array (i.e. tsArr) instead of the transaction array. This

[4] `https://bitcoinvisuals.com/chain-input-count-tx`

is possible because the timestamp is a block feature, meaning that all transactions that belong to the same block share the same timestamp. In this case, instead of appending timestamps to the end of transactions, an auxiliary array of size $|B_h|$ can be used. Given a transaction's block height (h), the desired transaction timestamp is stored in tsArr[h].

As a practical example, consider that, as of January 1st, 2023, $|T_h| \approx 800e6$ and $|B_h| \approx 770e3$, hence $|B_h|$ is a fraction of $|T_h|$. Each timestamp is a 4-byte unsigned integer, thus storing this attribute for each transaction requires 3.2 GB for $|T_h| = 800e6$, while storing it on a per-block basis requires just 3 MB for $|B_h| = 770e3$. As a result, grouping timestamps by block frees 3.2 GB of memory/storage resources, and each transaction can still be mapped to its timestamp in $\approx O(1)$ using our auxiliary indexing arrays presented later in the next section, but most importantly the average transaction in normal form occupies exactly 64 bytes, which is one cache line.

4.5 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.

$$\forall i \in [1, |T_h|) : \text{tArr[toff[i]:toff[i+1])} \tag{4.11}$$

$$= T_h[i].norm \tag{4.12}$$

This means that 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,

$$\forall h \in [1, |B_h|) : \text{tArr[toff[boff[h]] : toff[boff[h+1]]) =} \tag{4.13}$$

$$= B_h[h].t \tag{4.14}$$

We can also index transactions by block height and transaction block position $(t.h, t.b)$,

$$a = \text{toff[boff[t.h]+t.b]} \tag{4.15}$$

$$b = \text{toff[boff[t.h]+t.b+1]} \tag{4.16}$$

$$tArr[a : b) = \tag{4.17}$$

$$= B_h[t.h].t[t.b].norm \tag{4.18}$$

These two auxiliary arrays allow $O(1)$ mapping from raw transaction data to our normal form and vice versa. Transactions are stored contiguously and in the same order they appear in the blockchain, which is the equivalent of ascendingly sorting transactions based on their block index and their intra-block index.

It is worth noting that these offset arrays contain either 4 or 8-byte unsigned integer numbers. Depending on the size of the blockchain sample that needs to be retrieved, an 8-byte representation might be necessary for the offset arrays. If the size of the desired blockchain sample necessitates 8-byte offsets, this can be avoided by storing the data across multiple arrays. If this is not desired, then the cost of 8-byte offset arrays will result in a $\approx 5\%$ increase of the output's size (transaction data and offset data).

In our large experiment, which covers all transaction data from the first block until the last block mined in 2022, our 4-byte offset arrays account for less than 5% of the total output size. Thus doubling the size of the offset arrays due to an 8-byte integer representation does not double the size of the output.

4.6 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,

$$\text{tArr} = \tag{4.19}$$

$$= [B_h[0].t[0].norm, \ldots, B_h[N-1].t[M-1].norm] \tag{4.20}$$

$$= [T_h[0].norm, \ldots, T_h[(N \cdot M) - 1].norm] \tag{4.21}$$

Thus, tArr is a one-dimensional array that stores blockchain transactions in nor-

mal form, in the same order they appear in the blockchain.

In our reference CPython implementation, we do not store tArr in a single array because of memory limitations. We store a subset in secondary storage on $5e6$ transaction intervals. The result is an ordered set of files, each adhering to the data layout described above. Depending on the machine's memory capacity, the data can be stored either in a single file or across multiple files. If the latter is necessary, once the pipeline finishes its execution the files can be seamlessly merged into one.

In the intermediate representation, between stages one and two, inputs are represented as a triplet of integers compressed into one integer. We will explain the encoding shortly, but for now, assume that each input is represented as an integer triplet (a list of three integers).

For every $t \in T_h$ we have,

$$\forall i \in [0, |t.inp|) : (t^i.h, t^i.b, t.in[i].outi) \wedge (t^i \neq t) \tag{4.22}$$

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 $in.v$.

Extracting the first two attributes efficiently is non-trivial because an input (in) references the output it spends by the latter's $t'.id$ and position in $t'.out$. We could use BitcoinCore[5] to retrieve the output based on the tuple ($in.id$,$in.outi$) but the performance hit on the pipeline would be unbearable.

Blockchain data in BitcoinCore are stored in secondary storage, in their raw format. As we will show later in this work, the complete randomness of output references results in sequential lookups of data not stored *close* to each other. Thus, CPU data prefetching[22], which relies on a predictable sequence of memory reads, cannot be leveraged, and as a result, the majority of the data are CPU cache misses. When data do not reside in the cache or prefetch buffer, the CPU has to retrieve them from memory, which is orders of magnitudes slower than the cache/prefetch buffer, and the worst case scenario is a major page fault[6], in which case data reside in secondary storage - which itself is orders of magnitude slower than a memory read.

Using BitcoinCore to retrieve each input's output reference is the equivalent of sequential non-contiguous memory address reads from secondary storage. That would

[5]`https://bitcoin.org/en/bitcoin-core/`

[6]`https://en.m.wikipedia.org/wiki/Page_fault`

be catastrophic for the pipeline's performance. Furthermore, the absence of any additional info renders linear search the only possible option to find t' in the blockchain. This means that for each in, the pipeline would have to sequentially traverse all $t \in [0, t'.h)$. The search space becomes forbiddingly large very fast, and thus $O(n)$ search is not a viable option for this problem.

The ordered traversal of the blockchain holds two important pieces of information not stored in raw transaction data; $t.h$ and $t.b$. Ordered storage of tArr enables efficient transaction indexing. Given an arbitrary transaction id as input, the corresponding transaction can be retrieved in $O(1)$ via,

$$TMap_h(id) \rightarrow [t.h, t.b] \tag{4.23}$$

$$id = t.id \tag{4.24}$$

And thus,

$$t = B_h[t.h].t[t.b] \tag{4.25}$$

Where $TMap_h$ is a key-value data structure that maps an arbitrary transaction ID, whose transaction height is $\leq h$, to a tuple that contains the transaction's block height and position in the block.

In our case, we are only interested in transactions with unspent outputs. Once an *out* is *spent* by an *inp*, it cannot be referenced by another valid transaction. As a result, once $t.uout \in \varnothing$, the transaction no longer needs to be indexed by the map.

To keep track of a transaction's $|t.uout|$ we modify the map as follows,

$$UMap_h(id) \rightarrow [t.h, t.b, |t.uout|] \tag{4.26}$$

When a transaction is added to $UMap$, $|t.uout|$=$|t.out|$. When a $in \in t'.in$ references $t.uout$ sometime in the future ($t'.ts > t.ts$), $|t.uout|$ 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. We present the empirical gains from this optimization in *Results*. Thus, $UMap_h$ indexes all of the unspent transactions in the blockchain at height h.

4.7 UMap in CPython

4.7.1 Python *dict* and its memory footprint

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

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[9]. For large payloads, the resizing factor for a Python dictionary is two[10], which we can use to approximate an upper bound.

Let k and v be two arbitrary Python objects representing a key-value pair. Let n be the number of key-value pairs in the dictionary, and let n be large enough such that the dynamic array contains 4-byte unsigned integers (i). Let the size of the hash (h) be 8 bytes. The following formula calculates a dictionary's payload (P) size,

$$P(n, k, v, i, h) \tag{4.27}$$

$$S(P) = n \cdot (S(k) + S(v) + 2 \cdot S(ptr) + S(h) + S(i)) \tag{4.28}$$

$$= n \cdot (S(k) + S(v) + 2 \cdot 8 + 8 + S(i)) \tag{4.29}$$

$$= n \cdot (S(k) + S(v) + S(i) + 24) \tag{4.30}$$

i in our case is a 4-byte unsigned integer,

$$S(P(n, k, v, uint32, h)) = n \cdot (S(k) + S(v) + 28) \tag{4.31}$$

Thus, we estimate a Python dictionary's memory footprint to fall within this range,

$$S(P) \leq S(dict) \leq 2 \cdot S(P) \tag{4.32}$$

[7] `https://github.com/python/cpython/blob/main/Objects/dictobject.c`
[8] `https://mail.python.org/pipermail/python-dev/2012-December/123028.html`
[9] `https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf`
[10] `https://fengsp.github.io/blog/2017/3/python-dictionary/`

4.7.2 Python *int* and the NumPy library

CPython uses *bignum* arithmetic to represent integers[11], which allows an *int* object to represent values of arbitrary length. Integers are represented as a weighted sum of powers of a base. In CPython, the base is 2^{30} (b) and each weight is an unsigned 4-byte integer. Weights can take any unsigned value up to $2^{30} - 1$ because CPython uses only 30 out of the 32 bits for each integer.

Weights are stored in an array in little-endian order - starting from the least significant value. For instance, consider the number $i = 234254646549834273498$. In *bignum* arithmetic, i is represented by the following weighted sum,

$$i = 462328538 \cdot (b)^0 + 197050268 \cdot (b)^1 + 203 \cdot (b)^2 \tag{4.33}$$

Internally, the underlying C *struct* that implements the *int* object contains a *uint32* array (*digits*), that in i's case contains the following weights,

$$digits = [462328538, 197050268, 203] \tag{4.34}$$

If we omit the *digits* array, the underlying C struct contains 24 bytes of data[12], thus a small integer that requires only one weight to be represented takes up 28 bytes (24+4). As a result, the minimum size of a CPython *int* is 28 bytes.

Considering the above, 84 bytes are needed to store $t.h, t.b$, and j, all three of which are relatively small 4-byte unsigned integers, plus the size of the container, which itself is an object (i.e. list).

In CPython, raw data types (i.e. *int*, *float*) can be stored in their raw format using the *NumPy* library. NumPy is a comprehensive C-wrapped library/API that facilitates fast and efficient vector and matrix operations by leveraging modern CPU parallel computation features, such as Single Instruction Multiple Data (SIMD). As a result, performing matrix operations on NumPy data structures is orders of magnitude faster than performing the same operations on *list* objects.

However, traversing a NumPy array is orders of magnitude slower than traversing a conventional *list*, because every raw data type element in a NumPy array has to be converted into a CPython object. Our main transaction array is not a mathematical vector, it is a one-dimensional array of *structs* (in C terms), traversed in every stage

[11] `https://levelup.gitconnected.com/how-python-represents-integers-using-bignum-f8f0574d0d6b`
[12] `https://github.com/python/cpython/blob/3.9/Include/longintrepr.h`

following its creation, thus a NumPy representation would severely slow down the pipeline.

4.7.3 Handling hash collisions

We previously described how we use *MurmurHash* to convert raw transaction IDs and Bitcoin addresses into 4-byte unsigned integers. Unambiguous resolution of hash collisions usually leads to garbage output data, which in turn renders any data analysis applied to the data garbage.

Even though the hashed transaction ID is not part of our normal form, it represents the keyset of $UMap$, which is crucial for recurrent use of the pipeline as well as constructing the UTXO set (see *Storing $UMap$ and $parentMap$ efficiently*).

The ordered traversal of the blockchain and *MurmurHash*'s low hash collision rate allow us to store address transaction IDs efficiently. Since transactions are traversed in a sequential chronological order, their raw ID (which is unique) has not been referenced by any previous transaction inputs yet. Thus, 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 has reserved 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.

An alternative to using $t.id$ as a key for t's entry in $UMap$ in the event of a hash collision is using multiple hash functions, or applying *MurmurHash* to several variations of the $t.id$ until an available hash is found.

Considering a 1% hash collision rate, the empirical probability of two Bitcoin addresses sharing two hashes (double hash collision) is $0.01^2 = 0.01\%$, or 1 in every $10e3$. Given that our dataset consists of $800e6$ transactions, we should expect $\approx 80e3$ instances of two distinct transaction IDs sharing two distinct hashes.

This introduces ambiguity to the resolution of references made by inputs (spending an output of an older transaction). To replace raw input data with the $(t'.h, t'.b, in.outi)$ triplet, we need to retrieve the first two elements via $UMap$.

Let's assume that t''s *MurmurHash* and second hash both exist in $UMap$'s keyset (double hash collision). Unless $in.outi$ (the referenced output slot) is greater than the size of the unspent output set of one of the two $UMap$ entries that might represent

t', there is no unambiguous way of resolving t''s correct entry in $UMap$. Picking a $UMap$ entry in this case is the equivalent of basing a decision on the outcome of a fair coin flip.

As a result, employing multiple hash functions to resolve hash collisions for $UMap$ introduces ambiguity and is computationally expensive. The latter is valid because, for every transaction, we need to check the set membership of all of its hashes in $UMap.keyset$ to determine the occurrence of a hash collision.

Storing the raw transaction IDs in the event of a hash collision is much more efficient and unambiguous. For any arbitrary transaction t' referenced by some input in another transaction t, we check the membership of $t'.id$ in $UMap$. If it exists, t' is represented by $t'.id$ in $UMap$, and if it doesn't, it is represented by the *MurmurHash* of $t'.id$.

Considering a 1% hash collision rate, the average number of map lookups we have to perform per input to retrieve t''s indexes from $UMap$ is 1.99. This is a small price to pay compared to the cascading effect that incorrect transaction references have on the output data.

Resolving hash collisions between two user addresses unambiguously requires knowledge of both raw addresses. Unlike transaction IDs, user addresses are included in a transaction's normal form, therefore they must be in *uint32* format. Mapping raw addresses to their *uint32* hashes requires a key-value data structure where addresses (58 bytes) are keys and hashes (4 bytes) are values. Considering that as of January 2023 $|A_h| \approx 1e9$[13], this map requires at least 62GB of free memory, which is undesirable and defeats the core design principle of our pipeline and normal form, which is minimizing our working set.

Even though the probability of a hash collision is small, ambiguity in the resolution of transaction ID hash collisions is catastrophic because it results in garbage outputs. Several transactions spending X number of Bitcoins will inevitably reference outputs whose sum of Bitcoins is significantly less than X, and this cascades to the entire network of transactions. When the pipeline's output is analyzed, several addresses will have a negative net worth/account balance, which means that data integrity is completely compromised.

User address hash collisions have less severe side effects on the integrity of the output data. An address hash collision can be thought of as a false positive address cluster. A false positive address cluster leads to distinct user account balances being

[13]`https://blockchain.com/explorer/charts/n-unique-addresses`

incorrectly merged into one account balance, which while inaccurate, does not produce a garbage output.

4.7.4 Trivial UMap

We first present a trivial implementation of $UMap$.

- key: Hashed *t.id* or raw t.id (64 char long HEX string), stored as *int* or *str* respectively.
- value: $(t.h, t.b, |t.uout|)$ stored as a *tuple(int,int,int)* object instance.

We store the raw *t.id* only in the event of a hash collision, which empirically accounts for less than 1% of total hashes. *MurmurHash* maps the raw *t.id* to a 4-byte unsigned integer, which is stored in a 28 or 32-byte *int* object in CPython (i.e. integers close to 2^{32} are stored in a 32-byte *int* instance). Because the key's object type varies, $S(key)$ is stochastic, so we estimate the memory footprint based on $E(S(key))$.

- $P(hash\, collision) \approx 1\%$
- $S(t.id) = S(str[64]) = 132$ bytes
- $S(MurmurHash(t.id)) = S(int) = 32$ bytes
- $E(S(k)) = 0.01 \cdot S(str) + 0.99 \cdot S(int) = 33$ bytes

For $n = |T_h| = 800e6$ (end of 2022), we get:

1. $S(k) = E(S(k)) = 33$ bytes
2. $S(v) = S(tuple) + 3 \cdot S(int) + 2 \cdot S(ptr)$[14]
3. $S(tuple) = 48$ bytes[15]
4. $S(int) = 28$ bytes, $\forall i \in [0, 1e6)$
5. $S(v) = 48 + 3 \cdot 28 + 2 \cdot 8 = 148$ bytes

[14] The PyTupleObject C struct contains one object pointer by default, hence $2 \cdot S(ptr)$ instead of $3 \cdot S(ptr)$

[15] `https://github.com/python/cpython/blob/54ba556c6c7d8fd5504dc142c2e773890c55a774/Include/cpython/tupleobject.h#L9`

6. $S(P) = 800e6 \cdot (33 + 148 + 28) \approx 167$ GB

7. $S(UMap_h) \in [167, 334]$ GB

A dictionary value is 148 bytes which is not ideal. In a statically typed language, like C++, the three unsigned integers can be packed into a 12-byte *struct*, which is less than a tenth of 148 bytes. We can leverage the fact that a finite range bounds $t.h, t.b$, and $|t.out|/|t.in|$ to store all three in one 36-byte *int* instance, instead of three 28-byte *int* instances stored in a container object (i.e. *tuple*, list).

4.7.5 Optimized UMap

Considering the current size restrictions of the Bitcoin protocol and miners' economic incentives, it is infrequent for either $|b.t|$ and $|t.out|$ to exceed $10e4$, and virtually impossible to exceed $1e6$. The average number of transactions per block for the past six years has hovered around $2e3$, briefly reaching a peak of $\approx 3.4e3$ before reverting to the mean[16]. The record stands at transactions with $20e3$ inputs and $13e3$ outputs[17]. As of January 1st, 2023, $|B_h| \approx 770e3$, hence $b.h$ can be represented by a seven-digit integer to accommodate block heights $\leq 1e7$.

The following map transforms a $t.h, t.b$, and j ($|t.out|$, $in.i$, $in.outi$, $out.i$) triplet tr into a 22-digit unsigned integer,

$$EncodeT(tr) \rightarrow ((1e7 + t.h) \cdot 1e7 + t.b) \cdot 1e7 + j \tag{4.35}$$

Depending on the context, j can represent the following:

1. If tr describes an in, then $j \rightarrow in.i$ or $in.outi$

2. If tr describes an out, then $j \rightarrow out.i$

3. If t is meant to keep track of $|t.uout|$, $j = |t.uout|$

Let $DecodeT(int)$ be the inverse of $EncodeT(t)$,

$$DecodeT(int) \rightarrow (t.h, t.b, j),\ int \in N \tag{4.36}$$

Where,

$$t.h = (int)\ div\ (1e14) - 1e7 \tag{4.37}$$

[16] https://blockchain.com/explorer/charts/n-transactions-per-block

[17] https://coinmetrics.io/batching/

$$t.b = ((int)\ mod\ (1e14))\ div\ (1e7) \tag{4.38}$$

$$j = (int)\ mod\ (1e6) \tag{4.39}$$

Any unsigned integer in this range can be stored and represented by a 36-byte *int* instance. The maximum value in *EncodeT*'s range is $i = 19999999099999990999999$, which can be interpreted as block height 9999999, the transaction index in the block is 999999 and j is also 999999. i's *digits* array contains the following weights,

$$digits = [984550335, 776742595, 1734] \tag{4.40}$$

And thus, i equals the following weighted sum.

$$i = 984550335 \cdot (b)^0 + 776742595 \cdot (b)^1 + 1734 \cdot (b)^2 \tag{4.41}$$

Where $b = 2^{30}$.

Since i is represented by three 4-byte unsigned integers, the underlying C *struct* is $24 + 3 \cdot 4 = 36$ bytes. This takes a quarter of the resources required by the naive implementation's dictionary value.

Encode and *Decode* allows us to compress *tuple(int,int,int)* into a single *int* instance, resulting in significant memory savings. As before:

1. $E(S(k)) = 33$ bytes
2. $S(u) = S(int) = 36$ bytes
3. $S(P) = 800e6 \cdot (33 + 36 + 28) \approx 78$ GB
4. $S(UMap_h) \in [78, 156]$ GB

Our optimized map consumes $\approx 50\%$ less memory resources than the trivial implementation does. *Results* show that, in practice, our pruning optimization trims n to a fraction of $800e6$, thus $S(UMap_h) \lll 78$ GB.

It is worth considering that both *EncodeT* and *DecodeT* involve computationally expensive operations (*mod*,*div*), especially given the overhead associated with CPython's *bignum* arithmetic. Thus, even though this encoding reduces $UMap$'s memory footprint by half, the overhead associated with extracting the three integers from the single integer erases any performance gains from packing more data per cache line.

Traversing and extracting the triplet is approximately 80% slower than the tuple representation. In a statically typed language, like C++, the cost of decompressing a similar encoding (i.e. compressing a *struct* of three 2-byte unsigned integers in the range [1,9] into one 2-byte unsigned integer), would be approximately equal to the cost of reading more cache lines; traversing the map would take approximately the same time in both cases, but the memory footprint of the compressed map would be a fraction of the uncompressed one.

However, the 80% performance hit of *Decode* is hidden by the main latency in the pipeline which is reading data from secondary storage (raw blockchain data) in the first stage and the resolution of output references in the second stage. This means that the pipeline does not incur a 2X slowdown from *Decode* because the cost of reading data from the dictionary is negligible compared to the unavoidable cost of reading the raw blockchain data from storage.

As a result, as long as map lookups are not the main latency in the pipeline, halving $UMap$'s memory footprint makes it possible to extract the entire unclustered blockchain on a machine with no more than 32GB of memory (see *Results*) using vanilla Python. In 2023, the overwhelming majority of researchers have access to a machine with 32GB of memory.

4.8 UMap in a statically typed environment

In a statically typed environment (i.e. C, C++, Rust), the encoding that we just presented is not necessary. The $t.h, t.b$, and j triplet can be stored in a *struct* (or array) that contains three unsigned 4-byte integers.

One side-effect is that the intermediate representation corresponds to an incomplete normal form. Each input address is represented by three unsigned integers, and thus each transaction contains $2 \cdot |t.in|$ more elements than it would have in normal form. Consequently, in a statically typed implementation of the pipeline, transactions are in normal form at the end of stage two.

4.9 Address clustering

This stage is necessary for the creation of the user graph, as well as the clustered compressed blockchain. By this stage, every t is in normal form ($t.norm$).

We use the Common-Input-Ownership (CIO) heuristic to cluster addresses. This heuristic assumes that all *in.addr* $\in$ *t.in* belong to the same owner (cluster), hence when addresses from different clusters are encountered in a *t.inp*, they are merged into one cluster (transitivity).

Formally, let I_t be a finite that contains t's list of input addresses. Let $C(a) \rightarrow c$ map some address a to some cluster c, where both a and c are integers. The two following properties then hold,

$$\forall a \in I_t : C(a) \rightarrow c \tag{4.42}$$

And $\forall (t, t') \rightarrow (t \neq t')$:

$$(I_t \cap I_t' \neq \varnothing) \rightarrow (\forall a \in (I_t \cup I_t') : C(a) \rightarrow c) \tag{4.43}$$

CIO can be efficiently implemented using the *Weighted Quick Union-Find* [23, 24] disjoint-set data structure, with path compression. Disjoint-set data structures efficiently manage disjoint sets while facilitating operations to determine set membership and union of sets. [14] implements CIO using a variation of *Union-Find*.

In the original *Quick Union Find* each set element is represented as a node in a tree structure, where each tree represents a set. To determine set membership ($Find$), it follows parent pointers from an element until it reaches the root, which is the set's representative. $Union$ involves attaching the root of one tree to the root of another ($O(1)$), thereby merging two sets. Joining the two sets arbitrarily can lead to extremely deep trees. In the worst case, $O(N)$ memory accesses are required to to reach the root ($N =$ number of nodes in a tree). Linear complexity scales poorly for large inputs ($1e9$ addresses in our case).

The weighted version, (WQUF henceforth) with path compression introduces two clever optimizations.

Weighting: By keeping track of the size of each tree (number of nodes), the root of the smaller tree is attached to the root of the bigger tree. The resulting tree structure remains balanced, or at the very least, nearly balanced. As a result, trees grow horizontally instead of vertically, which leads to $O(logN)$ bounded memory accesses in $Find$.

Path Compression: By adding a second loop in $Find$, the path from the element to the root can be compressed by making each node on the path directly

point to the root. Path flattening renders $Find \approx O(1)$.

Both in theory and in practice, WQUF is orders of magnitude faster than quick union. Regarding path compression, we did not observe its theoretical gains over WQUF. Given that the vanilla algorithm guarantees nearly balanced and relatively shallow trees, the cost of traversing the path for a second time was not amortized in our case. Our implementation only flattens the input node of $Find$ at no extra cost.

4.10 Storing $UMap$ and $parentMap$ efficiently

$UMap$ serves as a checkpoint that facilitates efficient recurrent use of the pipeline and construction of the UTXO set.

Given a past extraction of heights $\in [0, h]$ the pipeline can pick up from where it left off to extract block heights $\in (h, h+k]$, $k > 0$. The map can be converted into three one-dimensional arrays so that its size in secondary storage equals the size of its raw payload (in raw data type terms).

Map keys are either *uint32* or 64-char long strings, and values are three *uint32* numbers (in our CPython implementation, this requires decompressing *dict* values using *Decode*). String-integer pairs can be stored in raw byte (keys) and *uint32* (values) arrays and integer-integer pairs can be stored in a single *uint32* array. In the integer-integer case, every dictionary entry (key-value pair) consists of four integers, one for the key and three for the value, so every four consecutive integers in the array (starting from index 0) correspond to a single map/dict entry. As a result, the map can be reconstructed on-demand in one $O(n)$ array pass.

The process of storing $parentMap$ is a little bit more involved because of the size of $|A_h|$. As of January 2023, $|A_h| \approx 1.08e9$, so the map contains one billion key-value pairs of four-byte unsigned integers. Storing $parentMap$ in a single array, as we did with $UMap$, requires 8.7 GB of storage $(4 \cdot 2 \cdot \ A_h)$.

In *Results* we make the case that the CIO heuristic compresses A_h to approximately half of its size, hence the new compressed address set contains about $|A_h|/2$ addresses. This means that $parentMap$ has approximately $|A_h|/2$ keys that point to themselves (set root elements). We can leverage this fact by storing root/cluster addresses in one array ($roots$), cluster members in a separate array ($children$), and an offset array (off) to index each root element's cluster members (slice in the *children* array).

We can calculate the total number of elements in the updated structure as follows:

- $|roots| \approx |A_h|/2$
- $|\text{off}| \approx |A_h|/2$
- $|children| \approx |A_h|/2$
- Total elements $\approx 3 \cdot (|A_h|/2)$

The one-array structure contains $2 \cdot |A_h|$ elements, whereas the new structure that groups addresses by root elements (cluster IDs) contains $\approx 3 \cdot (|A_h|/2)$ elements, which is 25% less than the one-array structure.

Chapter 5

Algorithms

Algorithm 3 Pipeline entry point.

1: **FUNCTION** Main
2: **Input:** clustered (boolean), unclustered (boolean), graph (boolean), first block height (starth), last block height (endh), path (output directory)
3: **Output:** $\varnothing$
4: $parentMap = \{\}$
5: $UMap = ToNormalForm(path, starth, endh)$
6: $InFileUOUT(path)$
7: **if** *clustered* or *graph* **then**
8: $\quad parentMap = CIO(path)$
9: $\quad$**for** $addr \in parentMap.keys$ **do**
10: $\quad\quad Find(addr)$
11: $\quad$**if** *graph* **then**
12: $\quad\quad CreateGraph(path, parentMap)$
13: $\quad\quad ExportGraph(path, parentMap)$
14: $ExportBlockchain(path, unclustered, clustered)$
15: $ExportMaps(path, UMap, parentMap)$

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 into *tArr* before moving on to the next transaction in the blockchain.

The procedure consists of two main loops; the outer loop traverses blocks, and the inner loop traverses a block's transaction set ($b.t$). $bacc$ is a block transaction offset accumulator that is incremented by $|b.t|$ before entering the inner loop and is appended to boff. For instance, the genesis block (b) contains transactions with $t.i \in [\text{boff}[0], \text{boff}[1]) = [0, |b.t|)$, while the second block (b') contains transactions with $t.i \in [\text{boff}[1] : \text{boff}[2]) = [|b.t|, |b.t| + |b'.t|)$.

A raw transaction (t) input references an unspent transaction (t') output by the output's raw id ($t'.id$) and its position in the transaction's output set ($out.i$). One of the goals in the first stage is to represent transaction inputs as encoded triplets $Encode((t'.h, t'.b, in.outi))$.

For each transaction input (in), $in.id$ (the equivalent of $t'.id$) is looked up in $UMap.keys$, and if it exists, it means that the raw transaction ID is stored in $UMap.keys$ instead of its murmur hash. This happens when there is a hash collision. Thus, $key = in.id = t'.id$. If $t'.id$ is not in $UMap.keys$, then the raw transaction's murmur hash is stored in $UMap.keys$. In this case, $key = Hash(in.id) = Hash(t'.id)$.

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. $UMap[key]$ is decremented by one to reflect that $t'.out[in.outi]$ is spent. This is the equivalent of decrementing $|t'.uout|$ by one.

Next, each output address in $t.out$ is murmur hashed and appended to $tempt$, followed by each output's $(out.f, out.v)$ pairs. The last three elements in the array are $t.insum.f = 0, t.insum.v = 0, t.ts$. The input sum flag/value pair is set to 0 because it is retrieved in the second stage of the pipeline.

t is now in normal form, and thus $tempt$ can be unrolled in $tArr$. $tacc$, which is a transaction offset accumulator, is incremented by $|tempt|$ and appended to toff, marking where the transaction's slice in $tArr$ ends. $t.norm$ is stored in $tArr[\text{toff}[t.i] : \text{toff}[t.i + 1])$. Before moving on to the next transaction, t is added to the unspent transaction set, represented by $UMap$.

The algorithm checks $t.id$'s murmur 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 murmur hash.

After a block's transaction set is converted into normal form, the procedure checks whether the pre-defined memory threshold is exceeded or not. If it is, $(\text{toff}, \text{boff}, tArr)$ is exported to a file in secondary storage and re-initialized. The file contains information about the height range spanned by $tArr$. The starting and ending points of the file's height range are the heights of $t.norm = tArr[0 : \text{toff}[1])$ and $t'.norm = tArr[\text{toff}[|tArr|] : |tArr|)$ respectively (first and last transaction in the array).

$UMap$ is pruned such that transactions with empty unspent output sets are removed from the data structure ($t.uout \in \varnothing$). If $(UMap[key]) mod(1e5) == 0$, then $t.uout \in \varnothing$. Appending $|t.uout|$ to the end of the integer ($UMap[key]$) allows decrementing it and comparing it against 0 without having to extract the triplet via $DecodeT$.

Algorithm 4 Traversing raw blockchain data to convert transactions into normal form.

```
1: FUNCTION ToNormalForm
2: Input: path, starth, endh
3: Output: unspent transactions map (UMap)
4: using MurmurHash as Hash
5: UMap, tArr, toff, boff, tacc, bacc = {}[ ], [0], [0], 0, 0
6: for b ∈ blockchain[starth : endh] do
7:     bacc+ = |b.t|
8:     Append(bacc, boff)
9:     for t ∈ block.t do
10:        tempt = [ ]
11:        Append(|t.in|, tempt)
12:        Append(|t.out|, tempt)
13:        if t.b > 0 then
14:            for in ∈ t.in do
15:                key = in.id
16:                if (key ∉ UMap.keys) then
17:                    key = Hash(in.id)
18:                outT = DecodeT(UMap[key])
19:                inT = (outT[0], outT[1], in.outi)
20:                Append(EncodeT(inT), tempt)
21:                UMap[key]− = 1
22:        else
23:            Append(CoinbaseAddr, tempt)
24:        for out ∈ t.out do
25:            Append(Hash(out.addr), tempt)
26:        for out ∈ t.out do
27:            Append(out.f, tempt)
28:            Append(out.v, tempt)
29:        for x ∈ (0, 0, t.ts) do
30:            Append(x, tempt)
31:        Append(tempt, tArr)
32:        tacc+ = |tempt|
33:        Append(tacc, toff)
34:        key = Hash(t.id)
35:        if (key ∈ UMap.keys) then
36:            key = t.id
37:        UMap[key] = EncodeT((t.h, t.b, |t.out|))
```

```
1:  //Line 37 of FUNCTION ToNormalForm
2:  //Nested inside the main loop
3:  if Limited Memory then
4:      file = toff, boff, tArr
5:      Store(file, path)
6:      tArr, toff, boff = [ ], [0], [0]
7:      UMap' = {}
8:      for k ∈ UMap.keys do
9:          x = UMap[k]
10:         if (x)mod(1e5) > 0 then
11:             UMap'[k] = x
12:     UMap = UMap'
13: //Outside the main loop
14: return UMap
```

5.2 Second stage: Resolving *uout* references

The ordered set 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. At the start of stage two, each transaction in $tArr$ is in normal form, which is an ordered set of the following $n + 3m + 5$ elements:

1. $|t.in|$

2. $|t.out|$

3. $\forall i \in [0, |t.in|) : EncodeT((t^i.h, t^i.b, t.in[i].outi))$

4. $\forall out \in t.out : out.addr$

5. $\forall out \in t.out : out.f, out.v$

6. $t.insum.f = 0$

7. $t.insum.v = 0$

8. $t.ts$

In a statically typed environment, transactions are in incomplete normal form. Each ordered set contains the following $3(n + m) + 5$ elements:

1. $|t.in|$
2. $|t.out|$
3. $\forall i \in [0, |t.in|) : (t^i.h, t^i.b, t.in[i].outi)$
4. $\forall out \in t.out : out.addr$
5. $\forall out \in t.out : out.f, out.v$
6. $t.insum.f = 0$
7. $t.insum.v = 0$
8. $t.ts$

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 $EncodeT((t.h, t.b, j))$. For a fileID $\in$ *lookupMap.keys* we have the following entry in $lookupMap$,

$$\text{fileID} : [in, out, \ldots, in', out'] \tag{5.1}$$

The ordered set of files and their transactions are traversed sequentially. 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 located in $tArr$. The referenced output (out) is retrieved from $tArr$ using the toff and boff offset arrays, and the corresponding *uout* reference in $t.in$ is replaced by $out.addr$.

In the first stage, every transaction input sum (flag, value) pair is initialized to zero so that every input sum equals 0 Satoshi at the start of stage two. The referenced output value ($out.v$) is converted into Satoshi and added to $t.insum.v$. $out.f$ is incremented by three to indicate that the output is spent. Once all inputs $\in t.inp$ are traversed, $t.insum.v$ is converted into the appropriate denomination, and the denomination flag is updated accordingly ($t.insum.f$).

If $t'.h$ is not in the file's block height range, then the input and the referenced output are encoded via $EncodeT$, the $(inInt, outInt)$ pair is appended to $lookupMap[t.h]$,

and the corresponding *uout* reference in $t.in$ is replaced by NULL. $lookupMap[t.h]$ contains all the inputs in $t.in$ with out-of-file *uout* references.

Once all transactions in the file are traversed, $lookupMap$ is grouped by file ID. Every entry in the updated $lookupMap$ contains unspent output references located in the same file. For every key (file ID) in $lookupMap.keys$, the appropriate file is loaded via file$'$ = files$[key]$. The sequence of $(inInt, outInt)$ pairs $\in lookupMap[key]$ is traversed, such that, each NULL input in file is replaced by the appropriate output address, the input sum is incremented and converted into the appropriate denomination, and outputs in file$'$ are incremented by three, marking them as spent.

Algorithm 5 In-file *uout* reference retrieval.

1: **FUNCTION** InFileUOUT
2: **Input:** path
3: **Output:** $\varnothing$
4: $lookupMap = \{\}$
5: **using** $lookupMap$ **as** map
6: **using** $path$ **as** files
7: **for** file $\in$ files **do**
8:     $\text{toff}, \text{boff}, tArr = \text{file}$
9:     **for** $t \in$ tArr **do**
10:         **for** $j \in [0, t.in)$ **do**
11:             $inT = DecodeT(t.in[j])$
12:             **if** $(inT[0] \in \text{file.heights})$ **then**
13:                 $a = \text{boff}[t.h] + t.b$
14:                 $t^j = tArr[\text{toff}[a] : \text{toff}[a+1])$
15:                 $out = t^j.out[inT[2]]$
16:                 $t.in[j].addr = out.addr$
17:                 $t.insum.v += toSatoshi(out.\text{f}, out.v)$
18:                 $t^j.out[inT[2]].\text{f} += 3$
19:             **else**
20:                 $inInt = EncodeT((t.h, t.b, j))$
21:                 $outInt = EncodeT((t^j.h, t^j.b, out.i))$
22:                 $Append(inInt, map[t.h])$
23:                 $Append(outInt, map[t.h])$
24:                 $t.in[j].addr = NULL$
25:         $\text{f}, v = satoshiToFlag(t.insum.v)$
26:         $t.insum.\text{f}, t.insum.v = \text{f}, v$
27:     $map = \text{OutOfFileUOUT}(path, \text{file}, map)$

Algorithm 6 Out-of-file *uout* reference retrieval.

1: **FUNCTION** OutOfFileUOUT
2: **Input:** path, file (toff, boff, tArr), map (lookupMap)
3: **Output:** $\varnothing$
4: $\text{toff}, \text{boff}, tArr = \text{file}$
5: Group map.keys by file ID
6: **for** $id \in map.keys$ **do**
7: $\quad \text{file}' = Load(path/id)$
8: $\quad \text{toff}', \text{boff}', tArr' = \text{file}'$
9: $\quad$ **for** $(in, out) \in map[id]$ **do**
10: $\quad\quad inT = DecodeT(in)$
11: $\quad\quad outT = DecodeT(out)$
12: $\quad\quad a = \text{boff}[inT[0]] + inT[1]$
13: $\quad\quad t = tArr[\text{toff}[a] : \text{toff}[a+1])$
14: $\quad\quad a = \text{boff}'[outT[0]] + outT[1]$
15: $\quad\quad t^j = tArr'[\text{toff}'[a] : \text{toff}'[a+1])$
16: $\quad\quad out = t^j.out[outT[2]]$
17: $\quad\quad t.in[inT[2]].addr = out.addr$
18: $\quad\quad t.insum.v = toSatoshi(t.insum.\text{f}, t.insum.v)$
19: $\quad\quad t.insum.v+ = toSatoshi(out.\text{f}, out.v)$
20: $\quad\quad t^j.out[outT[2]].\text{f}+ = 3$
21: $\quad\quad \text{f}, v = satoshiToFlag(t.insum.v)$
22: $\quad\quad t.insum.\text{f}, t.insum.v = \text{f}, v$
23: $\quad \text{file}'[2] = tArr'$
24: $\quad Store(\text{file}', path)$
25: $\text{file}[2] = tArr$
26: $Store(\text{file}, path)$
27: **return** $\{\}$

5.3 Third stage: Address clustering

This stage applies the CIO heuristic to the address set (A_h), which is implemented based on the Weighted Quick Union Find algorithm. $WQUF$ 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. Tree sizes are stored in a key-value data structure (*dict* in CPython), where a key represents a root address and the value the tree size.

The procedure traverses a transaction's (t) input space and applies $Find$ to every address until a non-empty set value is returned. If there is an input $\in t.in$ such that $Find(in.addr) \notin \varnothing$ the loop breaks, and every orphan input before in is attached to $in.addr$'s root address (cluster ID). $root$ is temporarily set to $Find(in.addr)$. $Find$ is then applied to every input (in') following in, followed by $Union(root, Find(in'.addr))$ to merge the two sets and update $root$ accordingly.

If all input addresses are orphans, we arbitrarily set $root$ equal to the address of $t.in[0]$ and make all input addresses point to it. This is the equivalent of creating a set of size $|t.in|$ by merging $|t.in|$ sets of size one.

$Find$ relies on a key-value pair data structure ($parentMap$) to map an address ($addr$) to its cluster ID/address. The procedure iteratively calls $addr = parentMap[addr]$ until $parentMap[addr] == addr$. A key that satisfies $parentMap[key] == key$ represents a set root (a cluster ID).

Algorithm 7 Applying the CIO heuristic to every $t.in \in T_h$.

```
1: FUNCTION CIO
2: Input: path
3: Output: {address : cluster address} key-value pairs (parentMap)
4: parentMap = {}
5: using parentMap as parent
6: using path as files
7: for file[2] ∈ files as tArr do
8:    for ∀t ∈ tArr do
9:       root, stop = ∅, 0
10:      for in ∈ t.in do
11:         root = Find(in.addr)
12:         stop = in.i
13:         if (root ∉ ∅): break
14:      if (root ∈ ∅) then
15:         root = t.in[0].addr
16:         ∀in ∈ t.in : parent[in.addr] = root
17:         treeSize[root] = |t.in|
18:         continue
19:      ∀i ∈ [0, stop) : parent[t.inp[i].addr] = root
20:      treeSize[root]+ = stop
21:      for  i ∈ [stop + 1, |t.in|) do
22:         addr = t.inp[i].addr
23:         root = Union(root, Find(addr))
24: return  parent
```

Algorithm 8 *Find* operation

```
1:  FUNCTION Find
2:  Input: node (address)
3:  Output: root (cluster address) or ∅
4:  father = parent[node]
5:  if (father ∈ ∅) then
6:      return ∅
7:  root = father
8:  while parent[root] ≠ root do
9:      root = parent[root]
10: if (father ≠ root) then
11:     parent[node] = root
12: return root
```

Algorithm 9 *Union* operation

```
1:  FUNCTION Union
2:  Input: rootA, rootB
3:  Output: rootA
4:  if (rootA == rootB) then
5:      return root_1
6:  treeAsize = treeSize[rootA]
7:  treeBsize = treeSize[rootB]
8:  if (treeAsize < treeBsize) then
9:      Swap(rootA, rootB)
10:     Swap(treeAsize, treeBsize)
11: treeBsize+ = (treeBsize == 0)
12: parent[rootB] = rootA
13: treeSize[rootA]+ = treeBsize
14: return rootA
```

5.4 Fourth stage: Bitcoin 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 is connected to each individual output cluster ($\forall out \in t.out : parentMap[out.addr]$). This process leads to the creation of $|t.out|$ number of edges.

A dummy $Coinbase$ node (user) can be added to the graph to represent transfers of newly minted Bitcoins and transaction fees. For each t, if $t.fee > 0$, 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,

$$edge = (parentMap[out.addr], out.\mathrm{f}, out.v, t.ts) \tag{5.2}$$

Edges have a fixed-size representation. Starting from the first list element, every four consecutive integers make up a single outgoing edge.

A quick optimization is flattening $parentMap$ by calling $Find$ on $\forall k \in parentMap.keys$. The cost of this is amortized during the creation of the graph because subsequent $Find$ operations cost $O(1)$. $parentMap$ is flattened after CIO finishes its execution.

Algorithm 10 User transaction graph $\hat{G}$ creation.

1: **FUNCTION** CreateGraph
2: **Input:** path, parentMap
3: **Output:** $\varnothing$
4: **using** $parentMap$ **as** $root$
5: **using** $path$ **as** files
6: $graph = \{\}$
7: **for** file[2] $\in$ files **as** $tArr$: $\forall t \in tArr$ **do**
8: $tEdges = [\,]$
9: $cluster = root[t.inp[0].addr]$
10: $outsum = 0$
11: **for** $out \in t.out$ **do**
12: $edge = (root[out.addr], out.\mathrm{f}, out.v, t.ts)$
13: $outsum+ = toSatoshi(out.\mathrm{f}, out.v)$
14: $Append(edge, tEdges)$
15: fees $= toSatoshi(t.insum.\mathrm{f}, t.insum.v) - outSum$
16: **if** (fees > 0) **then**
17: $\mathrm{f}, v = satoshiToFlag(\mathrm{fees})$
18: $edge = (CoinbaseAddr, \mathrm{f}, v, t.ts)$
19: $Append(edge, tEdges)$
20: **if** $(cluster \in graph.keys)$ **then**
21: $\forall e \in tEdges : Append(e, graph[cluster])$
22: **else**
23: $graph[cluster] = tEdges$
24: **if** $Limited\ Mempry$ **then**
25: $Store(graph, path)$
26: $graph = \{\}$

5.5 Fifth Stage: Exporting data to HDF5

The last stage exports the (toff,boff,tArr) and graph files, UMap, and parentMap into HDF5 format. Like the HDF5 structure for transactions, the user graph HDF5 output contains three arrays; *nodes*, off, and *edges*. The *nodes* array contains cluster addresses and *edges* is grouped by cluster address based on the order of addresses in

$nodes$.

For an arbitrary $i \in [0, |nodes|)$, $nodes[i]$'s outgoing edges can be retrieved via $edges[\text{off}[i] : \text{off}[i+1])$. Moreover, every transaction in the clustered version of the blockchain has one input, therefore $|t.in|$ is omitted because it is always equal to one. Therefore, a transaction in the clustered blockchain consists of $3 \cdot m + 5$ elements.

Algorithm 11 Exporting the clustered/unclustered normal transaction set to HDF5.

1: **FUNCTION** ExportBlockchain
2: **Input:** path, unclustered (boolean), clustered (boolean)
3: **Output:** $\varnothing$
4: **using** $parentMap$ **as** $root$
5: **using** $path$ **as** files
6: **if** $unclustered$ **then**
7:    **for** file $\in$ files **do**
8:       $out\text{HDF5} = \text{HDF5}()$
9:       $\text{toff}, \text{boff}, tArr = file$
10:       **for** $\forall arr \in (\text{toff}, \text{boff}, tArr)$ **do**
11:          $Append(arr, out\text{HDF5}.dataset)$
12:       $Store(out\text{HDF5}, path)$
13: **if** $clustered$ **then**
14:    **for** file $\in$ files **do**
15:       $\text{toff}, \text{boff}, tArr = file$
16:       $\text{toff}', tArr' = [0], [\,]$
17:       $acc = 0$
18:       **for** $\forall t \in tArr$ **do**
19:          $tempt = [\,]$
20:          $Append(|t.out|, tempt)$
21:          $clusterAddr = root[t.in[0].addr]$
22:          $Append(clusterAddr, tempt)$
23:          **for** $\forall out \in t.out$ **do**
24:             $Append(out.addr, tempt)$
25:          **for** $\forall out \in t.out$ **do**
26:             $Append(out.f, tempt)$
27:             $Append(out.v, tempt)$
28:          $Append(t.insum.f, tempt)$
29:          $Append(t.insum.v, tempt)$
30:          $Append(t.ts, tempt)$
31:          $acc+ = |tempt|$
32:          $Append(acc, \text{toff}')$
33:       $out\text{HDF5} = \text{HDF5}()$
34:       **for** $\forall arr \in (\text{toff}', \text{boff}, tArr')$ **do**
35:          $Append(arr, out\text{HDF5}.dataset)$
36:       $Store(out\text{HDF5}, path)$

Algorithm 12 Exporting the unspent transaction set ($UMap$) and the address to cluster address map (parentMap) to HDF5.

1: **FUNCTION** ExportMaps
2: **Input:** path, UMap, parentMap
3: **Output:** $\varnothing$
4: $strintK, strintV, intintKV = [\,], [\,], [\,]$
5: **for** $\forall k \in UMap.keys$ **do**
6:    **if** $type(k) == int$ **then**
7:       $keys = intintKV$
8:       $values = intintKV$
9:    **else**
10:       $keys = strintK$
11:       $values = strintV$
12:    $Append(k, keys)$
13:    $\forall v \in DecodeT(UMap[k]) : Append(v, values)$
14: outHDF5 = HDF5()
15: **for** $\forall arr \in (strintK, strintV, intintKV)$ **do**
16:    $Append(arr, out\text{HDF5}.dataset)$
17: $Store(out\text{HDF5}, path)$
18: **if** $parentMap.keys \notin \varnothing$ **then**
19:    $acc = 0$
20:    $clusterMap = \{\}$
21:    $\text{off}, roots, children = [0], [\,], [\,]$
22:    **for** $\forall k \in parentMap.keys$ **do**
23:       $root = parentMap[k]$
24:       **if** $k == root$ **then**
25:          **continue**
26:       **if** $root \in clusterMap.keys$ **then**
27:          $Append(k, clusterMap[root])$
28:       **else**
29:          $clusterMap[root] = [k]$
30:    **for** $k \in clusterMap.keys$ **do**
31:       $Append(k, roots)$
32:       $\forall addr \in clusterMap[k] : Append(addr, children)$
33:       $acc+ = |clusterMap[k]|$
34:       $Append(acc, \text{off})$
35:    outHDF5 = HDF5()
36:    **for** $arr \in (\text{off}, roots, children)$ **do**
37:       $Append(arr, out\text{HDF5}.dataset)$
38:    $Store(out\text{HDF5}, path)$

Algorithm 13 Exporting the user graph to HDF5.

1: **FUNCTION** ExportGraph
2: **Input:** path, parentMap
3: **Output:** $\varnothing$
4: **using** *path* **as** files
5: **for** file $\in$ files **as** *graph* **do**
6: $acc = 0$
7: $\text{off}, nodes, edges = [0], [\,], [\,]$
8: **for** $node \in graph.keys$ **do**
9: $Append(node, nodes)$
10: $\forall x \in graph[node] : Append(x, edges)$
11: $acc+ = |graph[node]|$
12: $Append(acc, \text{off})$
13: $out\text{HDF5} = \text{HDF5}()$
14: **for** $arr \in (\text{off}, nodes, edges)$ **do**
15: $Append(arr, out\text{HDF5}.dataset)$
16: $Store(out\text{HDF5})$

Chapter 6

Results

We present and evaluate our CPython implementation's performance concerning execution time, memory efficiency, achieved compression rate, and information loss.

Experiments	**Small**	**Large**
Start date (DD/MM/YY)	03/01/09	03/01/09
End date	09/05/14	31/12/22
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 6.1: Experiment Setup. The first column lists experiment parameters, the second column lists the results from the small experiment and the third column lists the results from the large experiment.

6.1 Compression

We define compression rate (CR) as the percentage data size reduction relative to the uncompressed data.

$$CR(a, b) = (1 - (S(a)/S(b))) \cdot 100 \tag{6.1}$$

Where a is a compressed version of b.

Experiment Large covers transactions from the genesis block to the last block mined in 2022, when $|T_h| \approx 800e6$ and $|A_h| \approx 1e9$, and $h = H$. Given the empirical average $|t.in| = |t.out| = 3$ for unclustered data (u), $|t.in| = 1$ and $|t.out| = 3$ for clustered (c), we can estimate the size of our normal form for both cases.

1. $S(u)$: $|T_H| \cdot S(t.norm) = 792e6 * 68 \approx 54$ GB
2. $S(c)$: $|T_H| \cdot S(t.norm) = 792e6 * 56 \approx 44.4$ GB

At $h = H$, $n = |UMap_{h=H}| \approx 90e6$ (see *Memory*). For $w = P(\text{hash collision}) \approx 0.01$ we get:

1. $S(umaparr) = n \cdot (w \cdot (64 + 3 \cdot S(uint32)) + (1 - w) \cdot (S(uint32) + 3 \cdot S(uint32))$
2. $S(umaparr) = 90e6 \cdot (0.01 \cdot 72 + 0.99 \cdot 12) \approx 1.13$ GB

At $h = H$, the number of clusters (set roots) is $n \approx 490e6$. The size of $parentMap$'s HDF5 structure is calculated as follows:

1. $S(pmaparr) = 2 \cdot n \cdot S(uint32) + (|A_{h=H}| - n) \cdot S(uint32)$
2. $S(pmaparr) = (|A_{h=H}| + n) \cdot S(uint32) \approx 1.62e9 \cdot 4 = 6.3$ GB

At $|T_{h=H}| \approx 800e6$, the auxiliary off array costs an extra 3.2 GB. In total we have:

1. $S(u') = S(u) + S(umaparr) + S(\text{off}) = 58.3$ GB
2. $S(c') = S(c) + S(umaparr) + S(pmaparr) + S(\text{off}) = 55$ GB

The observed $S(u)$, $S(c)$, $S(umaparr)$, and $S(pmaparr)$ are 53.8, 44.5, 1.2 GB, and 6.3GB respectively, which is consistent with our estimates. With $S(Blockchain)$ ≈ 446 GB on January 1st 2023, our normal form yields $CR(u', Blockchain) \approx 88\%$. Our normal form and the necessary key-value maps compress the Bitcoin blockchain to $\approx$ an eighth of its size.

Estimating $S(\hat{G_H})$ is harder because we do not know $|\hat{V}_H|$ apriori (the size of the vertices' set). In experiment A, $|\hat{V}_H| \approx 16e6$, which corresponds to 44% of $|A_H|$. In B, $|\hat{V}_H| \approx 490e6$, which is also a little less than half of $|A_H|$. We can safely estimate that applying the CIO heuristic to T_h produces an address set A'_H with $|A'_h| \approx |A_h|/2$, and $CR(A'_h, A_h) \approx 50\%$.

The observed $S(\hat{G}_H)$ in experiment B is $\approx$ 43 GB, with 4 GB evenly divided between *nodes* and off arrays, and 39 GB for *edges*. Because $UMap$ and $parentMap$ are also part of the output, the total size of the output is $\approx$ 47.3 GB. Since the *nodes* array already contains the root addresses, the *root* array that is part of $parentMap$'s HDF5 structure is omitted from the output.

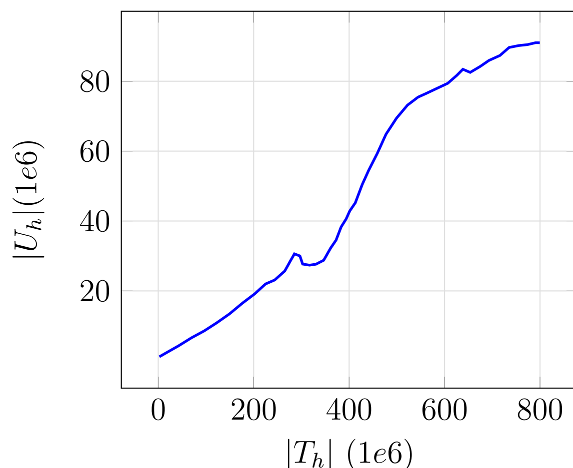

Figure 6.1: X and Y-axis represent the sizes of the transaction set ($|T_h|$) and unspent transaction set ($|U_h|$) respectively, as h grows to H.

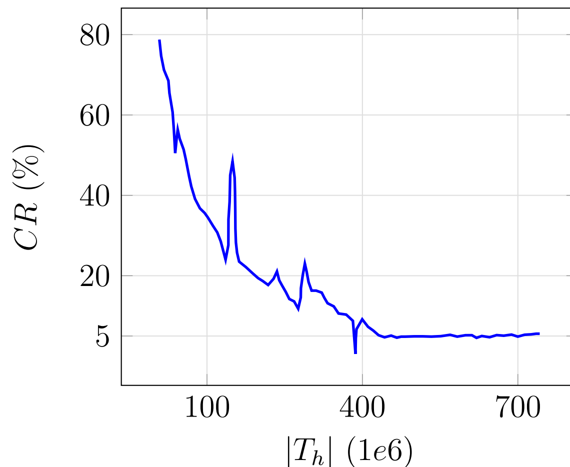

Figure 6.2: The graph shows the observed compression rate from pruning $UMap$ ($CR(UMap', UMap)$) as a function of the size of the transaction set $|T_h|$, as h grows to H.

6.2 Memory

$UMap$ and $parentMap$ are the data structures with the largest memory footprint in the API. A reasonable solution would be to replace *dict* with a key-value pair DB, in which case a sample of the key-value pairs would reside in the cache (DRAM in this case) and the rest in storage. We tried two state-of-the-art key-value storage solutions, *RocksDB* and *Redis*, and in both cases, elapsed time increased at least three-fold.

Because the underlying data are unspent output or address references, which are inherently random, caching data effectively and efficiently, such that frequently accessed keys are stored in memory and the rest in secondary storage, is challenging. In both instances, approximately half of the queries are cache misses, and retrieving data from storage is orders of magnitude slower than main memory. This necessitates the exclusive use of in-memory storage.

dict is CPython's fastest key-value pair data structure in terms of look-up and insertion time, hence it plays a crucial role in our implementation.

The decision to store intermediate data ([toff,boff,tArr]) in *pickle* files has a similar reasoning. We tried two RDBMs, *MySQL* and *SQLite*. Our implementation packed as many *INSERT* statements per database transaction as possible for efficient writes. Once again, elapsed time increased at least ≈ three-fold.

Figure 6.1 shows $|U_h|$ as a function of $|T_h|$ in the large experiment. The graph shows that $|U_h| \approx |T_h|/10$ throughout the experiment. This means that at an arbitrary h, approximately 10% of the transactions in T_h have $|t.uout| \geq 1$.

We previously estimated $S(UMap_H) \in [78, 156]$ GB by ignoring the effects pruning might have on n. For $n = |U_H| \approx 90e6$, $S(P(90e6, int, int))$ yields 8.7GB, thus our updated estimate is $S(UMap_H) \in [8.7, 17.4]$ GB.

The observed $S(UMap_H)$ is 13.1 GB, ≈50% greater than our lower bound, and ≈ to the median value of our estimated size range. As of October 2023, $|T_{h=H}| \approx 900e6$, thus our pruning optimization makes it possible for a machine with no more than 32GB to extract 90% of the unclustered transaction data via a CPython implementation of the API. Dictionary pruning is not an in-place process, as it first copies the desired key-value pairs to a new dictionary (D'), and then deletes the old one (D), thus it requires at least $2 \cdot S(D)$ of memory.

Figure 6.2 shows $CR(UMap', UMap)$ in the large experiment with respect to $|T_h|$. The map is pruned every time the extracted block data are stored in secondary storage and removed from the cache. Data is dumped in storage on $5e6$ transaction intervals, thus 160 such measurements are made.

The graph shows that 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 $40e6 \cdot (1.05)^{80} \approx 200e6$. That would have been more than double the observed $|UMap.keys_H| \approx 90e6$, yielding a cumulative $CR(UMap', UMap) \approx 55\%$.

$parentMap.keys$ 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 6.2: First column lists the parameters of the elapsed time experiment, second and third columns list the results in Experiment Small and Large respectively (single thread/process), fourth column lists Experiment Large/ Experiment Small timing ratios.

Extracting and converting raw transaction data into our normal form is computationally cheap. The dominant cost is retrieving blocks from secondary storage in the first stage, followed by retrieving out-of-file references in the next stage, and lastly querying the two maps in main memory.

Reading blocks from secondary storage is a cost that can not be avoided. We parse raw blockchain data using a CPython blockchain library we found on GitHub[1], because of its flexible API and the fact that it is faster at reading the data than BitcoinCore is. Moreover, we optimized the retrieval of out-of-file *uout* references by indexing transactions via $UMap$, such that we avoid file scans, and by grouping data by file ID, such that we load each file once.

Table 6.2 lists the timings for each stage. Experiment Large's working set is ≈ 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. A main priority for our future work is further investigating the potential bottlenecks caused by this API.

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

The remarkably poor performance of stage two can be attributed to the fact that its worst case, in terms of complexity, turns out to be its average/expected complexity due to the inherent 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 the overwhelming majority of our block files. As a result, most files had to access all previous files to retrieve input data, which is the equivalent of performing a file scan each time. Thus, our group-by optimization failed to improve the second stage's performance due to the inherent nature of the data (at least at no extra cost).

A quick solution to this problem would be to keep transaction data in memory instead of breaking them up across multiple files. In this case, a CPython implementation would require at least 512GB of memory, which most commodity machines don't have. However, if such a machine is at the researcher's disposal, then that solution would be optimal. Our indexed data layout allows $O(1)$ lookups, and even though most reads will still be cache misses for the reasons described above, DRAM is still orders of magnitude faster than reading from secondary storage (we refer to the overhead of loading pickled files from storage). We estimate that a C++ or Java implementation can keep all data in memory on machines with 64-128 GB of memory (depending on the size of the desired output) because all of the associated data structures would be raw data types, and thus occupy a fraction of the memory that the corresponding CPython objects do.

6.4 Information loss from floating point arithmetic

In the normal form, Satoshi values $\geq 2^{32}$ are mapped to 4-byte unsigned integers based on the encoding described in *Normal Form.* Rounding (already) rounded numbers introduces cumulative approximation error, meaning that the observed error can be greater than the individual approximation error ($1e-4$ in our case, because we round to the fourth digit).

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 respectively at h. $Supply_h$ represents the Bitcoin supply in the blockchain at h, and $\hat{Supply_h}$ 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 property states that a node cannot spend more Bitcoins than it owns. The following properties are deduced from the first one; the total wealth must be equal to the Bitcoins in circulation, and Bitcoins in circulation equals the sum of all Coinbase $t.out$ values minus the sum of every t.fee in the blockchain. All three constraints are relaxed, therefore a node's wealth can be *slightly* negative.

Regarding the third property, a Coinbase output (out) is funded by newly minted coins and aggregated transaction fees (t.fee) in its block. CB's incoming edges represent fees paid by individual transactions. Therefore, subtracting the total value of incoming edges from the total value of outgoing edges equals Bitcoins in circulation. CB is the only node in $\hat{G}$ whose wealth is allowed to be negative because $\hat{E}$ ($\hat{G}$'s edges) does not contain the self-loops necessary to represent the creation of Bitcoins.

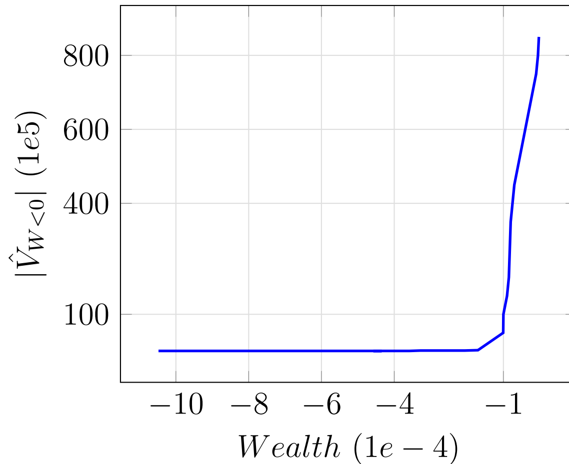

Figure 6.3: Cumulative Distribution Function of negative wealth. The X-axis represents negative wealth values and the y-axis node count.

Table 6.3 lists the necessary data to prove that our $\hat{G}_H$ satisfies all three constraints. All measurements are made at the end of experiment B, at height $h = H =$

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 6.3: First column lists the parameters of the experiment, second column lists the ground truth data, and third column lists the observed values.

769, 842. The raw blockchain data are the ground truth dataset.

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 actual 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 ($v \in \hat{V}_H : \hat{Wealth}_H(v) < 0$). With $|\hat{V}_H| \approx 490e6$, and $|\hat{V}_{H_{W<0}}|/|\hat{V}_H| \approx 14\%$, this means that $\approx 69e6$ nodes/vertices in the graph have a negative net worth. While $|\hat{V}_{H_{W<0}}|$ might seem concerning at first, it does not mean $\hat{G}_H$ is an invalid economy. This is because the empirical expected value of negative wealth is so close to 0 that it qualifies as a rounding error.

Our claim is backed by Figure 6.3, which shows the cumulative distribution function of negative user wealth at $h = H$. 99% of negative $Wealth$ values fall in the $[1e-4, 0)$ range, where $1e-4$ also happens to be our encoding's individual approximation error. 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 $(17/Supply_H) \approx 8.83e-57$, which means that the observed figure is very close to zero. $\hat{G}_H$ satisfies the last constraint, and can thus be classified as a *valid economy*.

Chapter 7

Conclusions

The undeniable impact of Bitcoin and blockchain technology on the global financial and technological landscape has spurred a pressing need for a comprehensive extraction and compression framework. As Bitcoin gains prominence in both academic and enterprise domains, the urgency to harness its vast data efficiently becomes increasingly evident.

This research has responded to this imperative by introducing an abstract framework that transcends the dependencies of any specific tech stack, effectively transforming raw data into a streamlined, normalized format tailored for rigorous research and high-performance data analysis.

A notable achievement of our framework is its ability to compress raw data to a fraction of its original size, reducing it to an eighth of its bulk while maintaining constant-time indexing. This paves the way for loading and querying the entire blockchain in the main memory of standard academic and enterprise machines, eliminating, for most tasks, the need for a resource-intensive DBMS.

Our data layout capitalizes on the CPU cache, boosting performance by accommodating an average Bitcoin transaction within a single cache line. To put this into perspective, on a machine equipped with a 384KiB L1 data cache (such as our commodity machine B), our framework allows an average of 5648 transactions to reside within the L1 cache, roughly equivalent to two and a half blockchain transaction blocks. Moreover, the versatility of our data layout seamlessly integrates with conventional SQL or NoSQL database management systems, offering a choice to suit researchers' preferences.

Furthermore, considering the diverse academic backgrounds of Bitcoin researchers, many of whom possess proficiency in Python, our work takes a significant stride for-

ward. Our reference implementation demonstrates that the entire blockchain and its accompanying user transaction graph can be extracted in six days with no more than 128GB of memory. This accessibility ensures that researchers, irrespective of their technical expertise and access to high-end machines, can engage in comprehensive Bitcoin data analysis.

Moreover, this does not prevent more computer science-savvy researchers from implementing the framework using a lower-level language (i.e. C++, Rust, Java). Based on our empirical observations from analyzing the user transaction graph using C++ and Python (where the main cost is traversing a large array), we estimate that an efficient C++ implementation of our framework should be able to extract the entire normalized blockchain and user transaction graph in less than twelve hours. This estimate is predicated upon the observed 14X+ speedup when analyzing the graph using C++ instead of CPython.

In closing, we hope our contribution facilitates the research and high-performance data analysis of the Bitcoin blockchain, enabling a broader spectrum of individuals to delve into its intricacies. We envision a future where research and innovation in the Bitcoin space will flourish, partly thanks to the accessibility and efficiency our framework offers.

Bibliography

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Cryptography Mailing list at https://metzdowd.com*, 2009, accessed: 2023-11-09. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] J. D. Bayliss, *The Data-Oriented Design Process for Game Development.* IEEE Computer Society, 2022, vol. 55, no. 05, pp. 31–38.

[3] M. Kowarschik and C. Weiß, *An overview of cache optimization techniques and cache-aware numerical algorithms.* Springer, 2003, pp. 213–232.

[4] U. Drepper, "What every programmer should know about memory," accessed: 2023-11-09. [Online]. Available: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[5] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 workshop on array databases.* Association for Computing Machinery, 2011, pp. 36–47.

[6] H. Kalodner, M. Möser, K. Lee, S. Goldfeder, M. Plattner, A. Chator, and A. Narayanan, "{BlockSci}: Design and applications of a blockchain analysis platform," in *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, 2020, pp. 2721–2738.

[7] J. Rubin, "Btcspark: Scalable analysis of the bitcoin blockchain using spark," https://rubin.io/public/pdfs/s897report.pdf, 2015, accessed: 2023-11-09.

[8] H. Mun and Y. Lee, "Bitsql: A sql-based bitcoin analysis system," in *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).* IEEE, 2022, pp. 1–8.

[9] M. Bartoletti, S. Lande, L. Pompianu, and A. Bracciali, “A general framework for blockchain analytics,” in *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers.* Association for Computing Machinery, 2017, pp. 1–6.

[10] D. Di Francesco Maesa, A. Marino, and L. Ricci, “Data-driven analysis of bitcoin properties: exploiting the users graph,” *International Journal of Data Science and Analytics*, vol. 6, pp. 63–80, 2018.

[11] D. Ermilov, M. Panov, and Y. Yanovich, “Automatic bitcoin address clustering,” in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA).* IEEE, 2017, pp. 461–466.

[12] A. Greaves and B. Au, “Using the bitcoin transaction graph to predict the price of bitcoin,” 2015, accessed: 2023-11-09. [Online]. Available: https://api.semanticscholar.org/CorpusID:18038866

[13] G. Kappos, H. Yousaf, R. Stütz, S. Rollet, B. Haslhofer, and S. Meiklejohn, “How to peel a million: Validating and expanding bitcoin clusters,” in *31st USENIX Security Symposium (USENIX Security 22).* USENIX Association, 2022, pp. 2207–2223.

[14] D. Ron and A. 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.* Springer, 2013, pp. 6–24.

[15] Y. Zhang, J. Wang, and J. Luo, “Heuristic-based address clustering in bitcoin,” *IEEE Access*, vol. 8, pp. 210 582–210 591, 2020.

[16] F. Reid and M. Harrigan, *An analysis of anonymity in the bitcoin system.* Springer, 2013.

[17] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, “A fistful of bitcoins: characterizing payments among men with no names,” in *Proceedings of the 2013 conference on Internet measurement conference.* Association for Computing Machinery, 2013, pp. 127–140.

[18] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Advances in Cryptology — CRYPTO' 92*, E. F. Brickell, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–147.

[19] S. Burckhardt, "Principles of eventual consistency," in *Foundations and Trends® in Programming Languages*, vol. 1, no. 1-2. Now Publishers, 2014, pp. 1–150.

[20] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," vol. 1. Springer, 2001, pp. 36–63.

[21] A. M. Antonopoulos, *Mastering Bitcoin*, 2nd ed. O'Reilly Media, Inc., 2017, pp. 55–88.

[22] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, 1990.

[23] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 215–225, 1975.

[24] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley Professional, 2011, pp. 216–233.