Masaryk University
Faculty of Informatics

Bitcoin Blockchain Analysis

Bachelor's Thesis

Peter Petkanič

Brno, Spring 2018

Masaryk University
Faculty of Informatics

Bitcoin Blockchain Analysis

Bachelor's Thesis

Peter Petkanič

Brno, Spring 2018

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Peter Petkanič

Advisor: RNDr. Martin Stehlík, Ph.D.

Acknowledgements

First of all, I would like to thank my advisor RNDr. Martin Stehlík, PhD and his colleague Mgr. Jaroslav Šeděnka for active and continuous support in every problem that has occurred while working on my thesis. Also, I am very fond of CESNET - MetaCentrum for providing enough computing power and virtual space and thus helped me complete this work.

Abstract

The primary goal of this thesis was to cluster Bitcoin addresses in blockchain into parts representing real-life Bitcoin wallets. This project focuses on database selection and optimal implementation of blockchain parsing and clustering. We go through database and implementation comparison and hardware usage as well as analysis of the results.

Keywords

blockchain, bitcoin, analysis, clustering, addresses, heuristics, multi-input, shadow, change, consumer, database, neo4j, java, multithreading, golang, graph database

Contents

Introduction

With the current monetary system, there are ways to make fraudulent transactions such as money laundering or black-market trading. The main problem for fraudsters is to find a way to hide such exchanges from the law which can be difficult as there are middlemen – banks. With the arrival of cryptocurrencies, there were more doors opened for such people as the main idea was to remove the middlemen and make the whole system owned by all the people. To make a transaction, there is no need to provide any kind of personal information like identity document required by banks. This may seem like heaven for false people, but are they truly under the thick hood of anonymity?

To store the transactions, cryptocurrencies use a database called blockchain representing ledger. The whole database is publicly held and distributed by everyone using peer-to-peer communication. By analysing this database, we can come up with heuristics that will lead to new connections between transactions that may uncover frauds. The result of analysis differs throughout various virtual currencies. This thesis covers the first and most popular one – bitcoin [1].

Such analyses were done on Bitcoin blockchain many times with different approaches and results. One of the first publicly available ones is WalletExplorer.com, which was created by Aleš Janda [2]. This implementation groups known addresses that are frequently used, which includes exchanges, gambling and black-market sites. Moreover, this website is partially grouping addresses into wallets, which means that one known address is enough to identify all other addresses of its owner to a certain degree. Since then, more companies such as Elliptic Enterprises Ltd. [3] or Chainalysis, Inc. [4] provide commercial solutions for crypto businesses. Institutions and people that can make use of the deanonymized blockchain usually work in the field of law enforcing, whether they are giants like Europol or smaller companies that provide cyber defence for the public. This data is also useful for an ordinary person that wants to know more about somebody's financial background.

The main assignment was to create analysis on grouped addresses similar to WalletExplorer.com, but using more information when creating address groups. This resulted in higher probabilities of real

wallets holding the same addresses as grouped addresses in our result. Blockchain consists transactional records for all of the addresses. By observing Figure 1, think of vault icon as banks taking care of your money and addresses representing your bank accounts. In Bitcoin protocol, we are the banks. Everyone takes responsibility for their assets and is their owner. Our vault is Bitcoin wallet (wallet explained in section 2.1), which grants us access to our funds on the blockchain. Wallet contains addresses that are holding our funds, but only we know which addresses are in the wallet. By analysing bitcoin Blockchain, we were able to partially identify which addresses come from the same wallet.

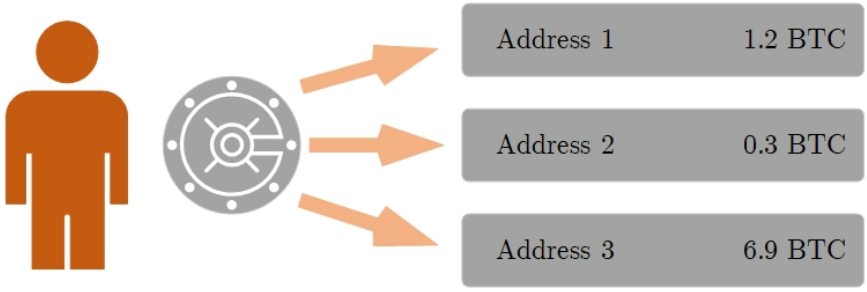

Figure 1: Bitcoin "account" visualization

In the first chapter, we describe what Bitcoin blockchain is and how it works. This chapter also explains how the blocks, transactions and addresses are represented in the blockchain. The second chapter contains a theory on Bitcoin wallets and address clustering. We describe every heuristic that was used in address grouping in detail. In the third chapter, we use the knowledge gained from previous chapters for database selection and creation of the database model, as well as a blockchain import into the database. We also describe implementation and application of heuristics explained in the second chapter. This includes approaches with multiple computing threads and difficulties that we encountered. The last chapter consists of the analysis including data and implementations analysis.

1 Bitcoin blockchain

As many technical terms bound with Bitcoin blockchain are used in this thesis, the current chapter explains the important facts and terms while discussing the technical background and relevant literature.

1.1 Overview of Bitcoin blockchain

Bitcoin blockchain is the ledger for the cryptocurrency bitcoin. It is holding all bitcoin transactions that have been made since the inception of the Bitcoin protocol in the immutable and verifiable way. Blockchain itself is formed by blocks of data containing transactions chained by their creation time. Every new block contains transactions which were confirmed after addition of the last block. By observing Figure 1.1, we can see that every block has an identifier of its predecessor - previous block hash. This means that the modification of the block 350000 would require alternation of all its descendants (block 350001 and 350002). This is important in terms of immutability, because it takes time to compute new block (more on block creation in 1.3.1).

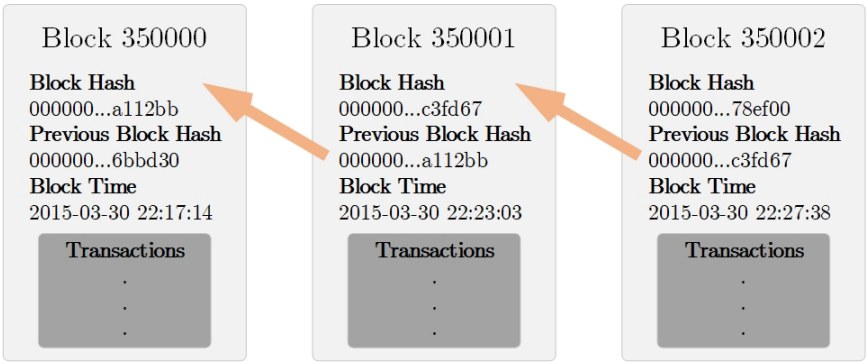

Figure 1.1: Representation of blocks inside the Bitcoin blockchain

The main idea behind the Bitcoin blockchain lies in its independence. The Bitcoin blockchain is not only decentralised but also permissionless database, which means that everyone has access to all the

data. No access control layer is required because, cryptocurrencies use proof of work (explained in section 1.3.1) [5] as a security measure.

1.2 Technical details of Bitcoin blockchain

Current chapter section provides information about blockchain in the real world. We will use this information moving forward in the thesis.

1.2.1 Block

The blockchain consists of blocks, which are integral nodes chained together. Every block holds a list of transactions that became valid when it was added to the blockchain. Figure 1.2 describes data included in the blocks relevant for this thesis.

- **Block Height** is a value that represents number of blocks between first and current block
- **Block Hash** is a hash of the current block header
- **Previous Block Hash** stores a hash of the previous block header
- **Block Time** stores GMT time and the date when the block was added to the blockchain
- **Transactions** contains a list of transactions included in the current block

Figure 1.2: Information saved in a Bitcoin block

1.2.2 Address

The address is an identifier for funds destination similar to bank account number. A single person can have hundreds to thousands of addresses since their creation is idiomatic and free.

There currently are three formats of addresses in use in the bitcoin blockchain. The only difference between them is in the transaction scripts (more on transaction scripts in section 2.1.1) they are used in [5]. Bitcoin does not require exchanges only between same address formats. Only relevant difference for us are the the first few characters:

- **Pay to Pubkey Hash (P2PKH)** starts with **1**
- **Pay to Script Hash (P2SH)** starts with **3**
- **Bech32** starts with **bc1**

Two more special formats will be included too:

- **Invalid** address format represents invalid (unreadable) address
- **Coinbase** address format represents input in first transaction where miner receives block reward

1.2.3 Transaction

Transferring a bitcoin to another address is called transaction. A transaction is confirmed when added to the blockchain, which means that the data we observe are static and irreversible since they are a part of the block. Every block begins with a *coinbase transaction*, which is the reward for the miner that was the first one to find the satisfying nonce (explained in subsection 1.3.1). Figure 1.3 describes the data included in transactions relevant for this thesis.

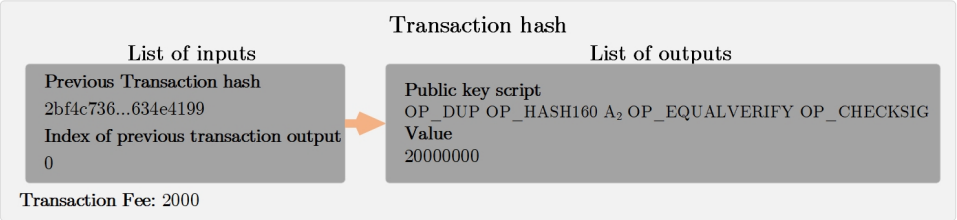

Figure 1.3: Information saved in transaction

- **Transaction hash** is a hash of the current transaction
- **Transaction fee** is the amount of bitcoin in satoshi[1] received by each miner as a reward, which generally depends on the transaction size - more inputs or outputs mean higher fees
- Each **Transaction input** consists of
  - **Previous transaction hash** - a hash of a previous transaction, of which output will be spent
  - **Index of previous transaction output** - zero-based index of a previous transaction output that will be spent
- Each **Transaction output** consists of
  - **Public key script** - script containing the address and the language that defines the conditions for spending this output
  - **Value** - the amount of bitcoin in satoshi that this output has

The important part about Bitcoin is that in order to send bitcoin, the sender has to have access to transaction outputs with enough funds. Bitcoin spends all the funds in the output and usually creates a new address with our change. For example in Figure 1.4 we are the owners of addresses A_2 and A_5.

1. Smallest unit in Bitcoin 1 Sat. = 0.00000001฿

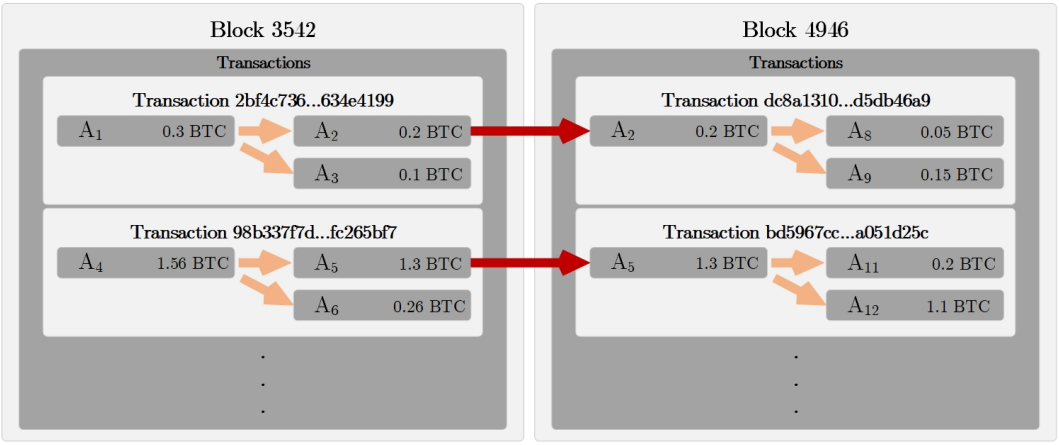

Figure 1.4: Transactions between blocks

In transaction with hash *2bf4c736...634e4199* we received 0.2฿ in block *3542* (see Figure 1.4). We decided to send 0.15฿ in block 4946. Input of the transaction *dc8a1310...d5db46a9* contains:

- **Previous transaction hash**, which is *2bf4c736...634e4199*
- **Index of previous transaction output**, which is 0 because output that we wish to spend is the first in the transaction that the output is a part of

Since we did not want to spend the whole output (all of 0.2฿), we received a new address A_8 with our change 0.05฿.

1.3 Creation of block and transaction

In contrast to traditional databases where a single entity holds the master copy of the data, Bitcoin protocol uses a flood protocol [5] that utilises peer-to-peer network to broadcast all of the changes to everyone. This eliminates the need for middlemen or other trustworthy entity. By observing Figure 1.5, we can see that some of the nodes in the network have full copy of the blockchain and the **memory pool**. A memory pool (mempool) holds a list of unconfirmed transactions waiting to be included in the blockchain. This information, generally

stored in RAM, is useful for miners (its purpose is described below in section 1.3.1). The node is configured to remove old transactions that are in mempool for a long time and have lower transaction fee than the threshold set by miner.

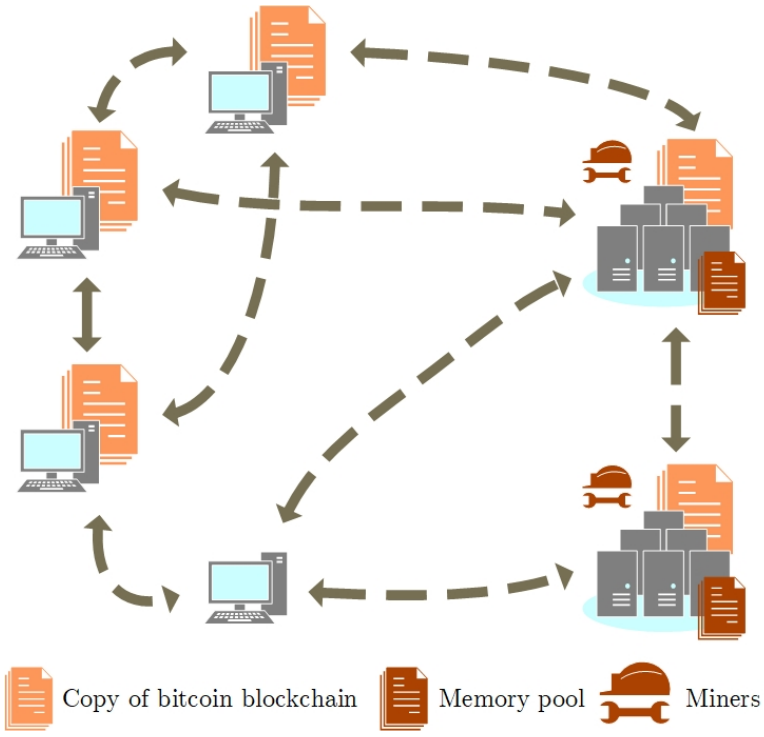

Figure 1.5: Simplified visualization of nodes in bitcoin P2P network

Wallet software uses stored cryptographic keys to create a transaction 2.1. After specifying the recipient address and transaction fee for miners, the wallet constructs a transaction and forwards this information to the closest nodes (computers). Before the transaction is handed for confirmation (mining), it has to be validated according to the consensus rules [5][6]. After successful validation, the transaction is further propagated to the closest nodes and is added to the mempool. Miners are typically selecting the most profitable transactions with the highest fee from their mempool. To add a block to the public blockchain, the miner has to solve a resource-intensive and difficult mathematical problem called proof of work. When the problem is

resolved the miner propagates its discovery of a new block, which is then added to the blockchain.

1.3.1 Proof of work

Bitcoin protocol requires a significant amount of work to create a new block, so its alternation will require even more work, which makes the blockchain secure, although still vulnerable to attacks [7][8]. Every block holds the hash of the header of the previous block, which makes it hard for untrustworthy parties to attack the network, because to modify a block, all the past blocks have to be modified. The Bitcoin network is setting the difficulty of the cryptographical assignment so that it takes two weeks to mine 2,016 blocks on average.

The task of miners rests in altering the number inside the block header called nonce to find the hash of block header lower than or equal to the target variable set by Bitcoin network. After a successful discovery of the required hash, the miner can add his Bitcoin address to the coinbase transaction (which contains the block reward) and propagate block to the network. Block reward is a sum of its transaction fees and base reward defined by Bitcoin protocol.

2 Address clustering

In this chapter we describe how wallet clients work and how we can use this knowledge to our advantage. This section also gives an overview of address clustering methodology.

2.1 Wallet

Bitcoin protocol uses public/private key pairs based on *Elliptic Curve Digital Signature Algorithm* (ECDSA) combined with asymmetrical cryptography to sign the transaction. These pairs are stored in file (or database) called wallet.

2.1.1 Transaction creation

In Figure 1.3, we stated that transaction output contains *Public key script* defining conditions for spending this output. The primary requirement for transaction creation is proving the ownership of this output. The *Public key script* is a Bitcoin protocol script consisting of the operations that have to be done to test sender's ownership of the output [9]. If verification passed, user is able to use the output as an input. As mentioned previously, Bitcoin protocol spends all of the funds in our transaction output. This means that we often have to create ECDSA key pair that we will send our change to (example in Figure 2.2).

To receive bitcoin, user has to share one of his hashed public ECDSA key pair values included in the wallet. This hash is called *address* (more on addresses in section 1.2.2).

2.1.2 Wallet clients

Bitcoin wallet clients are multi-platform applications often used to create a transaction and to manage ECDSA key pairs. The wallet software provides interface similar to internet banking that allows the creation of ECDSA pair and transaction. It also enhances the security of your wallet.

Sending a bitcoin via bitcoin client can look like this:

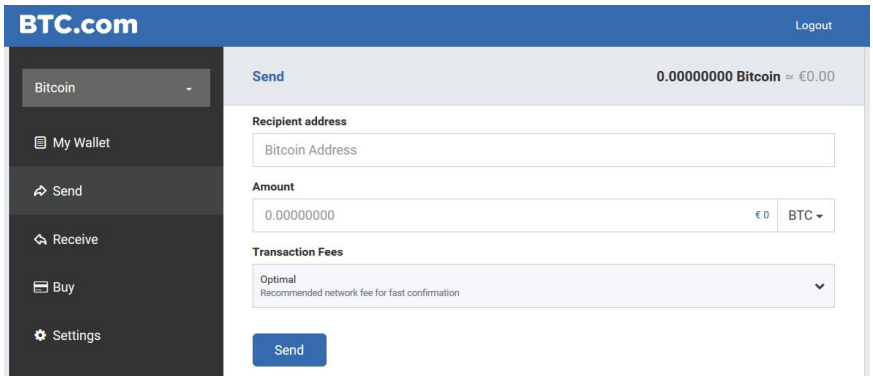

Figure 2.1: Browser based wallet client BTC.com [10]

Only address of the recipient, amount to be transferred and fee for the miner are required. Afterwards, wallet client chooses addresses with sufficient funds, constructs transaction and propagates it to the network.

2.1.3 Wallet client implementations

Before analysing bitcoin blockchain, we had to find out how wallet clients work because they are responsible for transaction creation. Specifically that they choose which *addresses* appear in the transaction.

Firstly, we had to identify which wallet clients were most frequently used throughout the history. Because there is no data about used wallet client stored in blockchain, we decided to follow these steps:

- collecting names of wallets widely used in reviews, recommendations and Google Search results
- finding out amount of ratings and downloads of each wallet client available on Google Play and App Store
- learning amount of search hits for wallet clients in several bitcoin communities - Bitcoin Reddit, bitcointalk.org, forum.bitcoin.com, bitco.in
- using information gained from two previous analyses in Google Trends

Results are shown in the table 2.1. We have used these found names followed by "*wallet*" to distinguish between companies and wallets (some companies, such as blockchain.info, provide both explorer and wallet client).

Table 2.1: Results of Wallet usage analysis

Search Term	GTrends	GPlay (Dl.)	AStore	Reddit	Forums
Coinbase	27.17%	141974 (5M)	1923	175476	104812
Ledger	26.36%	0	0	37827	113544
Blockchain.info	17.12%	33606 (1M)	395	14620	231911
Electrum	9.78%	1055 (0.1M)	0	36439	778580
Coinomi	8.42%	12653 (0.5M)	0	1792	9478
Luno	3.26%	5331 (1M)	0	448	2970
Breadwallet	2.45%	1994 (0.1M)	21	8685	2510
Freewallet	2.45%	2621 (0.1M)	0	213	2206
Mycelium	2.17%	4909 (0.5M)	5	21210	398323
Bitcoin.com	0.82%	18178 (1M)	0	1715	22646

GTrends column represents percentage share of all collected search amounts for each specific term used in the last year. Zero in *GPlay* and *AStore* means that no wallet exists for this platform (or if it does, it is not widely used).

By merging results from this method, we discovered that most frequently used clients are Coinbase, Ledger, Blockchain.info, Electrum, Coinomi, Luno, Breadwallet, Freewallet, Mycelium and Bitcoin.com. This analysis is not completely accurate, but it is sufficient for this bachelor thesis.

By studying source codes and documentation, we found out that all of these wallets use the same algorithm to choose transaction inputs. This implementation is well defined in Electrum[1]. Its goal is to make byte size of every transaction as small as possible to reduce fees while spending outputs. Electrum is also currently the only wallet allowing

1. `https://github.com/spesmilo/electrum/blob/80675121ce4882aa59ab98e662a2c3e8239602c6/lib/coinchooser.py#L212`

its users to choose algorithm which increases their privacy at the cost of larger transaction size.

The algorithm will collect addresses from the oldest one, till addresses with enough funds are collected. After unneeded addresses (their removal will not affect the sufficiency of funds) starting with the smallest value are removed, thus decreasing transaction fee. In Figure 2.2 algorithm will first collect all addresses *0c9ae7...4561f8* and *049bb7...b638f8*. Afterwards, only *0c9ae7...4561f8* is removed, since its unnecessary address with the smallest value.

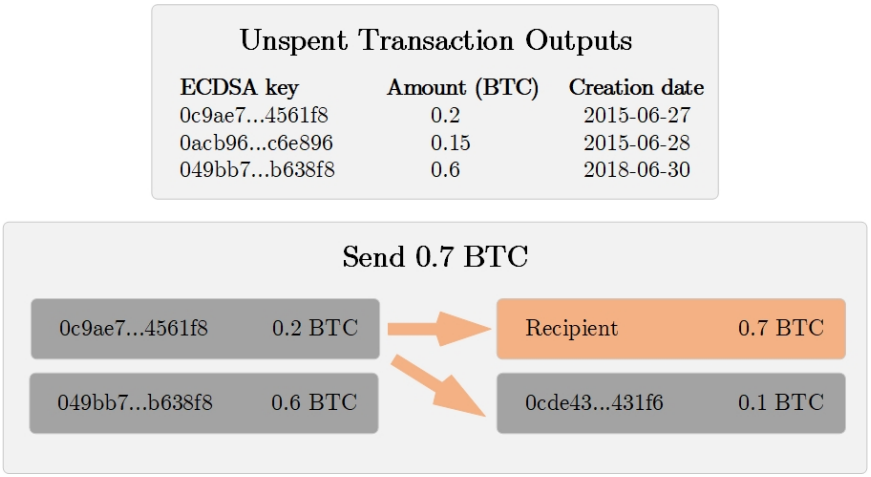

Figure 2.2: Spending outputs using wallet client.

2.2 Heuristics

With the knowledge gained from previous chapters, we can see some relations in the blockchain. Using them, we can partially group addresses into their corresponding wallets. Grouping them might give us the ability to analyse the financial background of a person or a company [11]. We are assuming that vast majority of bitcoin community uses consumer wallets unable or less likely to negate our predicament.

2.2.1 Multi-Input heuristic

Knowing how the algorithm for input address selection works, we can assume that all input addresses are associated with the same wallet [11]. In Figure 2.3, we are sending 0.5฿ . Our wallet client has chosen two addresses as an input for this transaction, which means all of the inputs come from the same wallet.

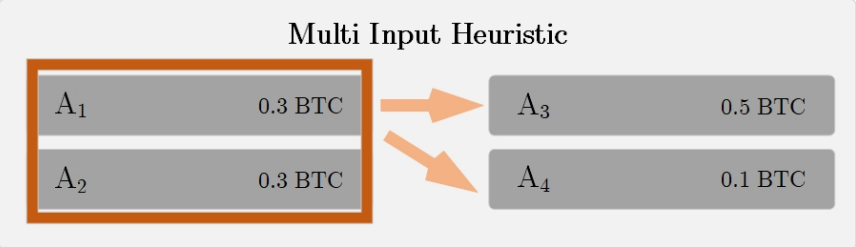

Figure 2.3: Multi-Input heuristic example. Addresses A_1 and A_2 are often from the same wallet.

The multi-input heuristic $h_I(A)$, where A is address, searches for all transactions with more than one inputs, where one of them is A and returns all inputs of this transactions.

This heuristic can be defeated by CoinJoin[12], which is a method of joining input addresses from multiple wallets. In Figure 2.3, it is possible that owner of A_1 is not the same one as owner of A_2, but none of the wallets from wallet usage analysis (in subchapter 2.1.3) implement this.

The success of this heuristic is dependent on the number of transactions with multiple inputs.

2.2.2 Shadow heuristic

When sending less bitcoin than output contains, most consumer wallet clients generate a new address and send the change to it [11]. In Figure 2.4, address A_2 has appeared for the first time in this transaction. Address A_3 has been used in blockchain before as well as A_1 (since it is input address). Addresses A_1 and A_2 come from the same wallet.

Shadow heuristic algorithm $h_S(A)$ searches for transactions that receive from A and have two or more outputs. After that, it returns an

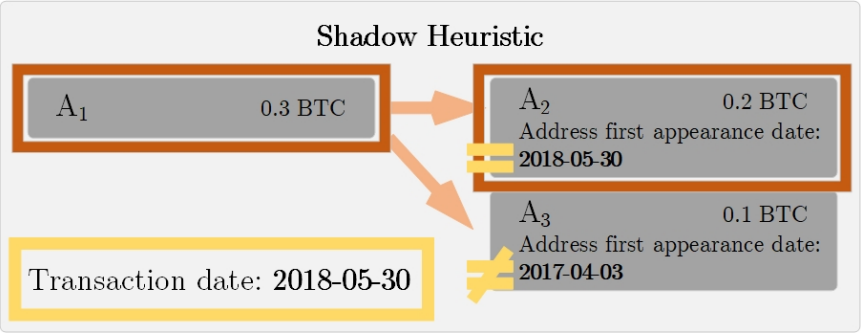

Figure 2.4: Shadow heuristic example. Addresses A_1 and A_2 are often from the same wallet.

output address that has appeared for the first time in the history of the blockchain, and there are no such other outputs in current transaction.

This heuristic can be defeated if the user specifies change address as already used address or receiver's address has not been used yet.

The success of this heuristic is dependent on the wallet software of the sender generating a new change address for each transaction.

2.2.3 Address format heuristic

Wallets usually stick to a single address format if the user does not choose otherwise. To identify change address of the sender, we can compare address formats used in inputs and outputs. In Figure 2.5, input A_1 has *P2PKH* format. There is only one output - A_2 - with same address format as input. Therefore, we assume that A_1 and A_2 come from the same wallet.

Address Format heuristic algorithm $h_F(A)$ searches for transactions that receive A and have two or more outputs. After that, it returns an output address with same format as one of input addresses, if there are no such other output addresses.

This heuristic can be defeated if the user starts using address format he has never used before.

The success of this heuristic is dependent on whether consumer wallet client allows changing address format or funds are transferred to another wallet.

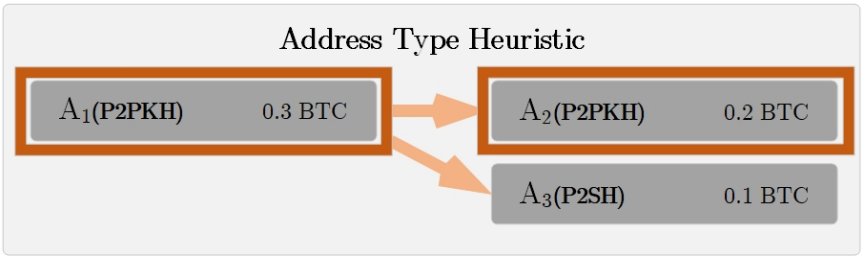

Figure 2.5: Address Format heuristic example. Addresses A_1 and A_2 are often from the same wallet.

2.2.4 Change heuristic

Change heuristic aims to answer same question as previous two heuristics, while observing the values of addresses in the transaction. It relies on wallet client's input selection algorithm not using unneeded inputs. Generally speaking, this only increases transaction size, thus also increasing transaction fee. Valid transaction's sum of inputs must be equal to or higher than the sum of outputs [11].

Change heuristic algorithm $h_C(A)$ returns list of output addresses that passed change heuristic described in Algorithm 1. The algorithm uses two functions:

- *SumValues(addressArray)* returns sum of values from addresses
- *MinValue(addressArray)* returns lowest value from addresses

If we run this algorithm on transaction in Figure 2.6, it returns only one address - A_5. There are, however, some cases when algorithm would return more than one address. If that happens, we can combine results with other heuristics (more in subsection 2.2.5).

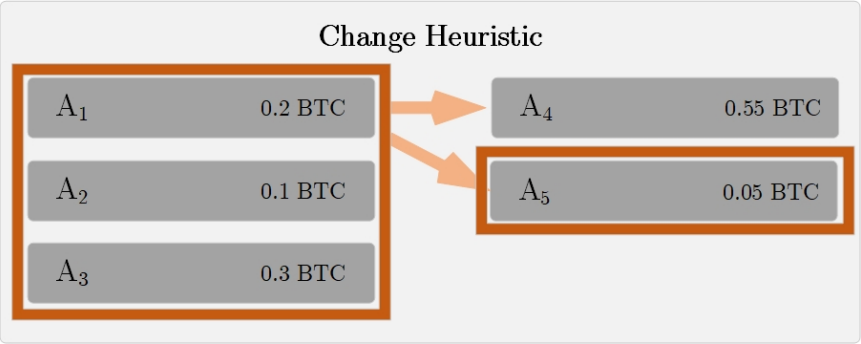

Figure 2.6: Change heuristic example. Addresses A_1, A_2, A_3 and A_5 are often from the same wallet.

Algorithm 1 ChangeHeuristic($txIns$, $txOuts$)

Input: $txIns$ is an array of transaction inputs,
$txOuts$ is an array of transaction outputs
Output: list of transaction outputs, that passed change heuristics

```
changeArray ← New(List)
if len(txIns) < 2 or len(txOuts) < 2 then return changeArray
end if

sumTxIns ← SumValues(txIns)
minTxIn ← MinValue(txIns)
sumTxOuts ← SumValues(txOuts)

for i ← 1 to len(txOuts) do
    if sumTxOuts − value(txOut[i]) > sumTxIns − minTxIn then
        changeArray.Add(txOut[i])
    end if
end for
return changeArray
```

This heuristic can be defeated by wallet client spending unnecessary outputs. Generally, when any of the inputs selected by selection

algorithm (see section 2.1.3) are removed, the sum of remaining ones should not be as high as the sum of output values. If this assumption is incorrect, the algorithm is unable to find change address.

The success of this heuristic is dependent on wallet client not spending unnecessary outputs.

2.2.5 Combinations of heuristics

These heuristics can be combined to achieve more accurate results. Possible outcomes are:

- **Change heuristic returning one address** - Output address in transaction has passed Change and in addition this address may also pass Shadow and Address Format heuristic, thus increasing the credibility.
- **Change heuristic returning more than one address** - Shadow and/or Address Format heuristics need to determine which address to include, otherwise no address will pass heuristic.
- **Change heuristic returning more than one address and Shadow and/or Address Format returning address not included in Change heuristic's result** - In this case, we select Shadow over Address Format heuristic.

3 Implementation of address clustering

Using information from previous chapters, we were able to partially cluster addresses in Bitcoin blockchain. This chapter describes how clustering was implemented and what difficulties we met.

3.1 Database selection

As we already know, blockchain is not a suitable database for data analysis. This section explains our choice of database.

3.1.1 Relational database

Our first choice was a relational database. Implementation of importing Bitcoin blockchain into the SQL database already exists and it is called Bitcoin Abe[1]. This gives us an opportunity to use various database engines (i.e. SQLite, PostgreSQL, InnoDB). On top of that, it also creates Bitcoin explorer similar to Blockchain.info[2] or Block Explorer[3]. Even though Abe does not cluster or analyze blockchain itself, it presents good overview of blockchain in relational database.

By trying out this implementation, we found out that:

- the initial import and parsing of blockchain takes approximately two weeks, which includes 490000 blocks (approx. 130GB of raw data) - we were able to reduce this to 52 hours by parsing data beforehand and using bulk insert
- querying transaction with more than 100 outputs and inputs combined takes more than 200ms

Afterwards, we applied address clustering heuristics on first fourth of blockchain to test the performance of the analysis. This added complexity and many relations, leading to unsatisfying results. For example, querying temporal change of heuristics usage took more than 10 minutes. This was caused mostly by multi-table relationships.

1. `https://github.com/bitcoin-abe/bitcoin-abe`
2. `http://blockchain.info`
3. `http://blockexplorer.com`

If our use case was Bitcoin explorer, this type of database would be sufficient. However, running even the simplest analysis took significant amount of time, so we decided to move one step further and try graph databases.

3.1.2 Graph database

Graph database uses graph structures with nodes, edges and properties to store and retrieve data. Nodes represent the entities (or rows in relational database) and edges represent the relations between these entities. A key concept of this system is the vertex graph, which directly relates stored data. In most cases this allows complex data to be retrieved with just one operation.

There are public implementations of Bitcoin blockchain stored in graph database. Bitcoin to Neo4j[4] parses each block and imports its results to the database block by block. Using this approach, it would take more than two months just to import the data to the graph database. We decided to parse Bitcoin blockchain data early and import them in bulk, which resulted in significant time improvement. We decided to use **Neo4j graph database engine** because of my previous experience with it during my studies.

3.2 Database model

Since correct database model is fundamental, we did a research on Bitcoin blockchain in graph databases using previous works as well as Neo4j documentation [13]. Many iterations were necessary for finding out which model was the best for address cluster analysis. One fourth (with respect to size) of blockchain was used and each iteration required:

- recreation of the database using new model consisting of blockchain parsing and database import
- implementation of address clustering algorithms

4. `https://github.com/in3rsha/bitcoin-to-neo4j`

- observation of the analysis results - returned data and overall performance

The final iteration had all the information we needed and performed well (more in chapter 4).

Database model consists of five main node types - **block**, **transaction**, **address**, **heuristic**, **cluster**.

3.2.1 Block node

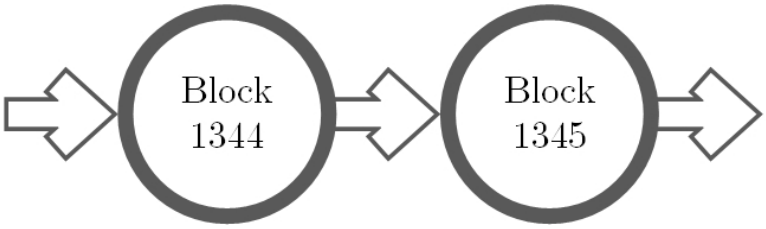

Figure 3.1: Block node representation in graph database

All block nodes are connected by oriented edges pointing to their successor - see Figure 3.1. Block node contains (technical details explained in section 1.2.1):

- **block height** stored as integer
- **block time** stored in Unix time format as long integer

3.2.2 Transaction node

Block nodes have oriented edges pointing to all transactions contained in represented blocks (visualized in Figure 3.2). Transaction node contains:

- **transaction hash** stored as string
- **transaction fee** stored as long integer

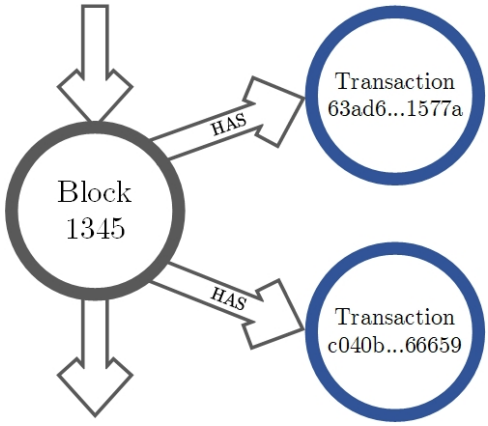

Figure 3.2: Transaction node representation in graph database

3.2.3 Address node

As pictured in Figure 3.3, there are two types of connections between each address and transaction nodes. Address nodes with oriented edge of type *SENT* represent transaction inputs. On the other hand, address nodes with oriented edge of type *RECEIVED* represent transaction outputs. Both edges store **value** representing amount being sent or received. Combinations of address and transaction nodes as well as *SENT* and *RECEIVED* edges create representation of Bitcoin transaction in graph database.

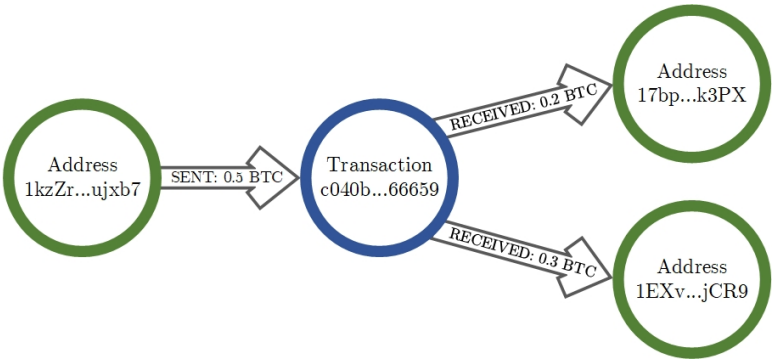

Figure 3.3: Address node representation in graph database

Address nodes have one of subtypes representing address type (format) - **P2PKH**, **P2SH**, **BECH32**, **COINBASE**, **INVALID**. Address node contains:

- **address** stored as string
- **first appearance time** of this address in whole blockchain stored in Unix time format as long integer

Both *SEND* and *RECEIVED* edge types contain amount sent or received in *satoshi* stored as long integer.

3.2.4 Heuristic and cluster node

It was important for us to preserve information on application of heuristics. Therefore, we came up with model in Figure 3.4. Green nodes represent address nodes and blue transactions. Orange nodes represent applied address clustering heuristics. For example, heuristic of type multi-input was applied in transaction T_1 and grouped two addresses A_1 and A_6 (both are inputs to the same transaction).

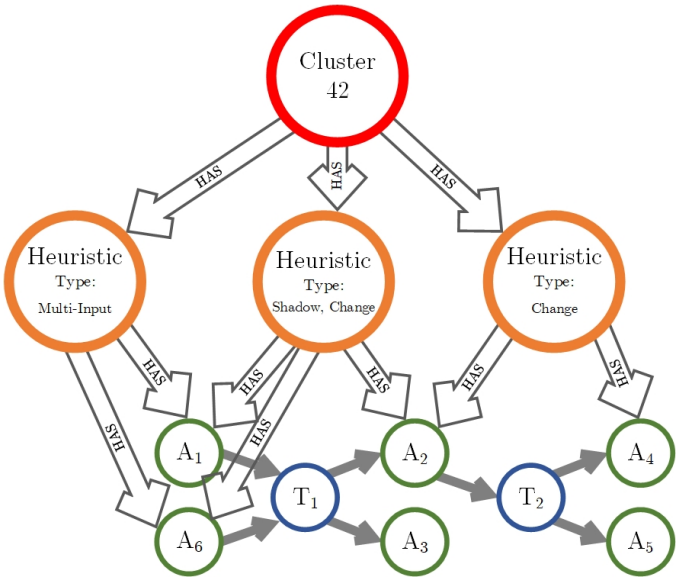

Figure 3.4: Heuristic and cluster node representation in graph database

If heuristic nodes share (have connected edge with) same address and their addresses are connected with other heuristic nodes, all these heuristics can be grouped together by cluster node. Cluster node represents possible real-life Bitcoin wallet.

Depending on the heuristics applied, heuristic nodes must have one or more of these subtypes: **shadow**, **change**, **wallet** (address format), **multi-input**. Every node also contains:

- **address** stored as string
- **appearance time** of this heuristic stored in Unix time format as long integer

3.3 Import options

In this section, we describe the possibilities importing data to Neo4j. After reading Neo4j Developer [13] and Operations Manual [14], we found out that there are three main ways of bulk importing the data to the database. This subsection describes all of them.

3.3.1 Cypher's `LOAD CSV`

Neo4j has query language called **Cypher** similar to SQL. Not only does it allow basic *CRUD (create, retrieve, update, delete)* operations, it also comes with procedure `LOAD CSV`. This allows user to import *comma-separated values*, while also specifying what should be done with the data. `LOAD CSV` can be executed on running database, but the main drawback is that Cypher uses web interface and web API calls, making loading big files slow.

3.3.2 Java API

Since Neo4j is written in Java, official support for their API driver for Java is also provided. This driver enables direct communication with the database, which makes everything faster than Cypher at cost of time spent developing. Bulk import can be implemented using `BatchInserters` class. We use Java API for data analysis, but there is a faster way for mass data import.

3.3.3 Neo4j Import tool

Neo4j Operations Manual [14] states that Neo4j has already developed utility for bulk database import in Java API called Neo4j Import tool.

This software creates database from large datasets stored in CSV files. Imported data have to be divided by nodes and relationships into separate files. Every file has its own header specifying how columns ought to be represented in database. The main advantage over other import options is simplicity and import speed, but this tool can be used only for creating new database.

3.3.4 Comparison of import options

According to the Table 3.1 we can clearly see that Neo4j Import tool is the best option for database creation. We will use Java API for database update and analysis, because it is faster than Cypher.

Table 3.1: Comparison of import options

	Speed	Preparation	DB creation	DB manipulation
`CSV LOAD`	Medium	Medium	✖	✔
Java API	High	High	✔	✔
Import tool	High	Low	✔	✖

3.4 Database preparation

To prepare data for analysis, we had to:

- parse blockchain to human readable format (JSON)
- apply heuristics on parsed data and convert results to CSV files used by Neo4j Import tool
- import data to database

3.4.1 Blockchain parsing

Parsing was necessary to further use raw blocks downloaded from Bitcoin network. According to database mode, we decided which information stored in blocks is needed and created software for converting raw blocks into structured *JSON (JavaScript Object Notation)* files. Every file had set structure:

Figure 3.5: Block structure in JSON file

Raw Bitcoin blocks are partitioned into chunks, where each contains unordered blocks in raw format. Because of that, parser has to store location of each block in file in the store.

As mentioned earlier, in raw block, transaction inputs do not save the address of the output being spent, rather reference to previous transaction and one of its outputs. As a result, this required storing transaction outputs in store, so parser could find address of an input and store.

On the top of that shadow heuristic needs to know when has address appeared for the first time in the blockchain, parser had to store address first appearance time as well.

Location of corresponding blocks on file system, outputs of transactions and address appearance times are accessed often, when parsing blockchain. Seeing that, repositories for this information are indexed key-value stores saved on the disk.

The indexing engine requires use of disks with fast random lookups like solid state drives. If spinning disks were used, this operation would take nearly fifty times longer. Parser parser can update JSON blocks with new blocks, since it preserves important data stored in key-value stores.

```
{
    "BlockHeight": "481823",
    "Hash": "507e6585...f0708e2d",
    "LockTime": 1503539571,
    "TxIns": [
        {
            "Address": "37TBito65...pm7WjGGQ",
            "Value": 257003
        }
    ],
    "TxOuts": [
        {
            "Address": "18hSSv1W...bjH6isnq",
            "AppearanceTime": 1503539571,
            "Value": 230000
        },
        {
            "Address": "3Nk7ynFs...zN1ca9ca",
            "AppearanceTime": 1503539571,
            "Value": 25098
        }
```


Figure 3.6: Transaction structure in JSON file

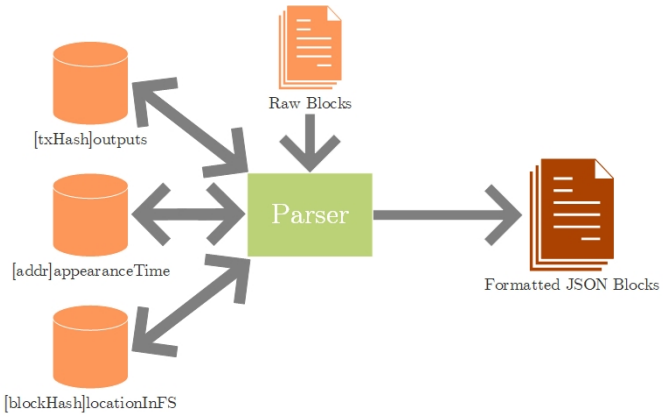

Figure 3.7: Data flow in the parsing software

3.4.2 Implementation of clustering

With structured blocks in *JSON* files, clustering software prepares data for import to *CSV* files. Program also cluster blockchain using defined address clustering heuristics. As stated before, Neo4j Import tool requires separate files for each type of node as well as relationship. For example to create block node and edges between them (as described in 3.2.1), software had to create *CSV* with specific stucture.

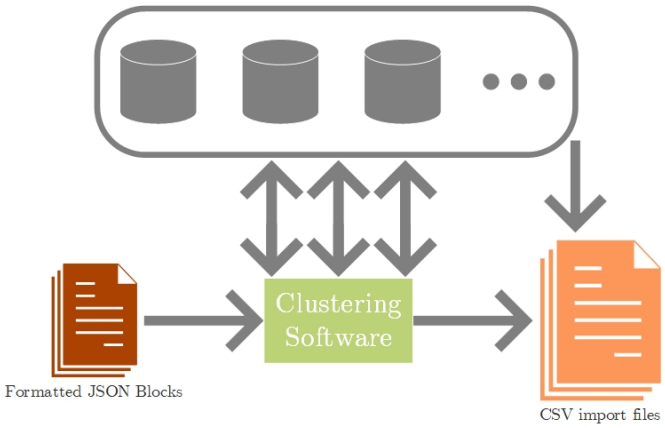

Figure 3.8: Data flow in the clustering software with blockchain split into three parts

Clustering blockchain requires merging of clusters when heuristic algorithm determines, that addresses with different clusters are in relationship. This is done in-memory by storing cluster of each address in hash map. Currently map uses `string` (address itself) as a key. This might lead to higher map lookup and insert times as well as higher memory usage than map with `integer` as key (more on improvements in 4.2). As shown in Figure 3.8 we are splitting splitting blockchain to equal parts (grey cylinders) and running clustering on each separately. The amount of parts is equal to number of processor's threads. After clustering is done, maps from each part are merged together. This approach has sped up clustering process 20 times.

Example *CSV* files:

- **Block node**

```
:ID(BlockHeight-ID),blockHeight:long,timestamp:long
456saq4d,369569,1439391754
36fdsa6s,369570,1439393756
78rwe9sd,369571,1439394214
```

- **Block to block relationships**

```
:START_ID(BlockHeight-ID),:END_ID(BlockHeight-ID)
456saq4d,36fdsa6s
36fdsa6s,78rwe9sd
```

Headers were defined according to the Neo4j Operations Manual [14] and are defining property names, types and edge orientation. `:ID` defines unique identifier of this node used when creating relationships. Edges will be oriented from `:START_ID` to `:END_ID` column defined in *CSV* file. If this data was imported into the database, the store would contain three blocks nodes connected by edges according to their creation time. Also, block node would have properties `blockHeight` (representing block height) and `timestamp` (representing block creation time). To sum up, 11 *CSV* files were needed to represent each node and relationship type as described in database model (five nodes, six relationship types). Headers are described in Appendix B.

3.4.3 Importing to database

With the results from clustering software in form of import *CSV* files, we are prepared for database import. Using manual from `help` argument of Neo4j Import tool, we were able to specify correct arguments and *CSV* files for import.

3.5 Database update

Application is designed to allow easy database update with new blocks included into the blockchain. This was however not fully implemented in the actual version of software, because it was more important to concentrate on proper address clustering itself. Only parser is able to

update *JSON* files with new blocks. It will however be accomplished this way.

Since parser is able to update *JSON* store with new blocks two elements need to be implemented:

- web API that will cluster block and returns the *JSON* result in form of clustered block
- software, which will consume result from this API and add new block to the database (using Neo4j Java API)

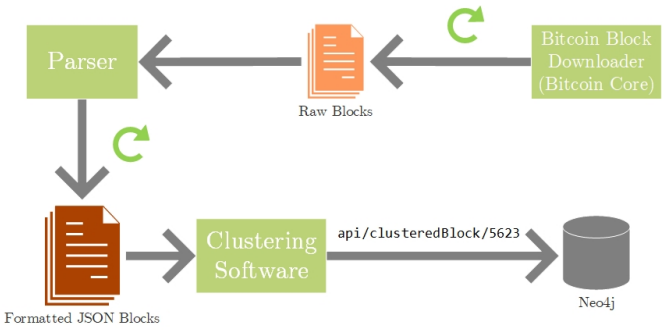

Figure 3.9: Data flow of database u[date] clustering database

4 Analysis

This chapter analyses performance of data parsing, clustering and importing as well as analyses clustered blockchain itself. Analysis includes first 508000 blocks of Bitcoin blockchain.

4.1 Data preparation

We used Bitcoin Core[1] to download raw blockchain blocks. Utilizing the example configuration file in Bitcoin Core GitHub repository [2] we have reduced the downloaded size to minimum. This can be done by appending `txindex=0` to the `bitcoin.conf` file which removed unneeded indexing of transactions.

As seen in Table 4.1, getting 508000 raw Bitcoin blocks from Bitcoin P2P network has low resource usage and resulted in 167GB of raw blocks.

Table 4.1: Resources used while downloading raw Bitcoin blocks

Computer used	CPU 6@2.8GHz, 300GB RAM, 1Gb/s conn.
RAM avg./peak	564MB/1362MB
Avg. CPU usage	31%
Time spent	5 hours and 21 minutes
Disk type	HDD 9000rpm
Size on disk	167GB

To parse raw blocks, we used our parser described in previous chapter. Different machine was used because we needed solid state drive for indexes. Table 4.3 states that 155GB, almost the same size as raw, human-readable *JSON* blocks were generated.

1. `https://bitcoin.org/en/bitcoin-core/`
2. `https://github.com/bitcoin/bitcoin/blob/master/contrib/debian/examples/bitcoin.conf`

Table 4.2: Resources used while parsing raw Bitcoin blocks to *JSON* files

Computer used	CPU 6@2.8GHz, 16GB RAM, 100Mb/s conn.
RAM avg./peak	4089MB/7863MB
Avg. CPU usage	71%
Time spent	23 hours and 49 minutes (w/o SSD 150+ hours)
Disk type	SSD and HDD 9000rpm
Size on disk	*JSON* 155GB (HDD), indexes 180GB (SSD)

Using our clustering software, we were able to apply defined heuristics on Bitcoin blockchain and export results to *CSV* files ready for database import.

Table 4.3: Resources used while clustering blockchain and preparing clustered data for database import

Computer used	CPU 6@2.8GHz, 300GB RAM, 1Gb/s conn.
RAM peak	103682MB
Avg. CPU usage	98%
Time spent	41 hours and 21 minutes
Disk type	SSD and HDD 9000rpm
Size on disk	indexes 110GB (SSD), *CSV* files 278GB (HDD)

Importing *CSV* files to the database and indexing properties of nodes was the fastest part of data preparation. Table 4.4 describes used resources.

Table 4.4: Resources used while importing clustered Bitcoin blocks

Computer used	CPU 6@2.8GHz, 300GB RAM, 1Gb/s conn.
RAM avg./peak	9760MB/20469MB
Avg. CPU usage	82%
Time spent	5 hours and 0 minutes (with indexing)
Disk type	HDD 9000rpm
Size on disk	302GB

Whole process took 75 hours and 31 minutes. If we do not include acquisition of raw blocks, the time will be reduced to 70 hours and 10 minutes. Comparing this approach to Bitcoin to Neo4j or Bitcoin Abe we can clearly see, that our implementation is faster. Moreover, we are also clustering blockchain. As mentioned previously, this is caused by other two implementations being more efficient towards memory usage. This can be achieved by direct database updates with every addition to Bitcoin blockchain.

4.2 Data analysis

This section describes how was analysis executed as well as its results. Analysis is split into three main parts:

- **Counts** - number of nodes with specified type in the database
- **Changes throughout time** - historical changes of used heuristics
- **Comparison to real-life wallet** - examine how well heuristics worked

Counts

All of the counts were queried using *Cypher's* procedure `MATCH (n) RETURN DISTINCT COUNT(LABELS(n)), LABELS(n);` using web interface

Clusters and Heuristics

Multi-input heuristic was applied 72075053 times in whole blockchain. By observing Table 4.5 we can see, that:

- if change heuristic passed, all other heuristics passed as well
- if shadow heuristic passed, at least one more heuristic passed too
- most successful heuristic was address format heuristic

Table 4.5: Heuristic counts

	Shadow	Format	Change	Combined	Any
Shadow	0	59494814	0	27043936	86538750
Format	59494814	14953278	0	27043936	101492028
Change	0	0	0	27043936	27043936

The important part is, that 65850738 cluster nodes were created out of 369479212 usable addresses. According to heuristics this means, that on average a single wallet contains keys to 5.6 addresses.

Addresses

Table 4.6: Address counts by address format

Address format	Count
P2PKH	329507787
P2SH	39950531
Bech32	20894
Invalid	4868504
Total	334376291

Heuristics usage throughout time

Using *Neo4j Java API* we were able to visualize usage of heuristics every two years. In Figure 4.1 we can observe how implementation of Bitcoin clients changed. For example till year 2012 there is very low usage of multi-input heuristic caused by change address reuse. If no new address was generated, user cannot spend more than one address at once.

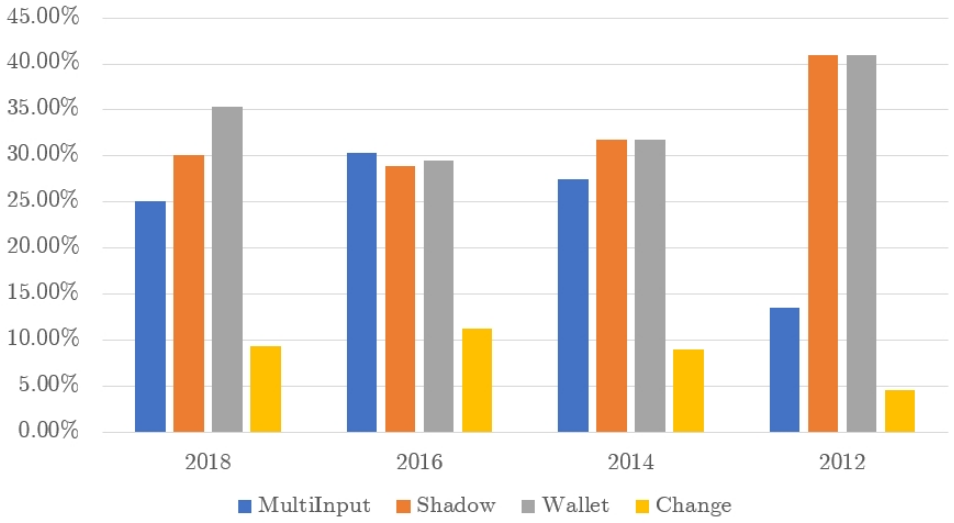

Figure 4.1: Heuristics usage in years 2012, 2014, 2016 and 2018

All results took less than 10 seconds to query.

Comparison to real-life wallet

To test how successful are the heuristics, we compared real-life wallet to the cluster created that includes one the wallet's addresses. Comparison was made with my wallet, which is using *BTC.com* Bitcoin client. Wallet included 12 transactions and 10 addresses. Address clustering was unable to fully distinguish addresses of this wallet, because *BTC.com* client automatically merges old unspent addresses. This can be resolved by adding new heuristic (more on improvements in Summary).

Summary

Since this is my first big project with big data, I have met with various difficulties. At the beginning I tried bulk importing blockchain into the graph database and running address clustering using multiple threads on live database (implemented with Java API). Multi-threaded computing made everything faster, but clustering was still very slow because Neo4j uses single thread to write. After meeting with my mentor, we decided to run address clustering on lower level - directly when parsing blockchain. Java does not allow using primitive types in maps, which resulted in out of memory errors. I decided to switch to *GoLang* because it is often compared to C, when it comes to its speed and memory usage. Moreover, it has convenient syntax, when programming concurrent applications.

Software can be improved in many ways. When parsing blockchain, addresses can be given unique identifier, that can replace strings. This can reduce memory usage and speed up indexing in maps as well as in key-value stores. After profiling clustering software, I realized, that reading *JSON* files takes significant about of time. This is caused by *JSON* files being scattered stored directly file system. Application has to open and close every *JSON* block. The solution to this can be simple database for all the files. As we have seen in section 4.2, when comparing real-life wallet to our results we were not particularly successful.

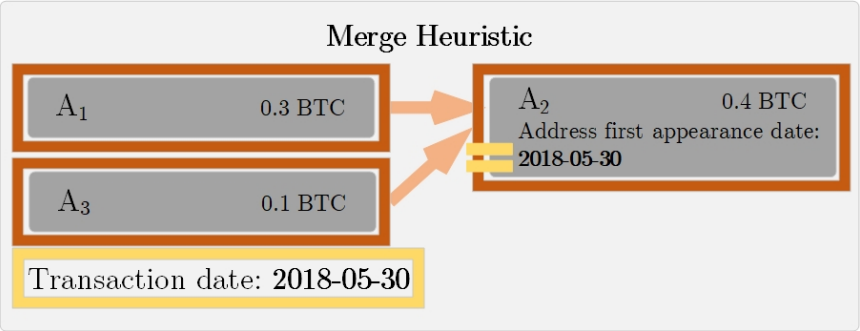

Figure 4.2: Possible representation of merge heuristic - all addresses often belong to the same wallet

Solution to that is more robust heuristics. In our case, we can create new heuristic, that will detect such merges. By observing Figure 4.2 we can see, that there is only one output address and it appeared for the first time in current transaction. Because senders have very rarely exact amount scattered in their addresses and there is only one output, we can assume, that all addresses belong to the same wallet. Lastly, I would like to implement this project for use with distributed file system i.e. Hadoop, Apache Spark and compare results.

This project will be further developed on my GitHub repository[3].

3. `https://github.com/cabrenn`

Bibliography

1. NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. Bitcoin Project, 2008 [visited on 2018-01-22]. Available from: https://bitcoin.org/bitcoin.pdf.
2. JANDA, Aleš. *WalletExplorer.com: smart Bitcoin block explorer*. 2013. Available also from: https://www.walletexplorer.com/.
3. *Elliptic*. 2013. Available also from: https://www.elliptic.co/.
4. *Chainalysis – Blockchain analysis*. 2014. Available also from: https://www.chainalysis.com/.
5. *Developer Guide - Bitcoin* [online]. Bitcoin Project [visited on 2018-01-22]. Available from: https://bitcoin.org/en/developer-guide.
6. *Protocol rules* [online]. Bitcoin Wiki [visited on 2018-01-22]. Available from: https://en.bitcoin.it/wiki/Protocol_rules.
7. *Developer Guide - Bitcoin* [online]. Bitcoin Project [visited on 2018-01-22]. Available from: https://bitcoin.org/en/developer-guide#term-51-attack.
8. *Irreversible Transactions* [online]. Bitcoin Wiki [visited on 2018-01-22]. Available from: https://en.bitcoin.it/wiki/Irreversible_Transactions.
9. *Developer Guide - Bitcoin* [online]. Bitcoin Project [visited on 2018-01-22]. Available from: https://bitcoin.org/en/developer-reference#opcodes.
10. *BTC.com Wallet* [online]. BTC.com, 2017 [visited on 2018-01-22]. Available from: https://wallet.btc.com/.
11. NICK, Jonas David. *Data-Driven De-Anonymization in Bitcoin* [online]. Switzerland: Oath Tech Network, 2015 [visited on 2018-01-22]. Available from: https://hbr.org/2017/01/the-truth-about-blockchain.
12. *CoinJoin* [online]. Bitcoin Wiki, 2011 [visited on 2018-01-22]. Available from: https://en.bitcoin.it/wiki/CoinJoin.
13. NEO4J, Inc. *The Neo4j Developer Manual v3.4* [online]. 2018 [visited on 2018-01-22]. Available from: https://neo4j.com/docs/developer-manual/current/.

14. NEO4J, Inc. *The Neo4j Operations Manual v3.4* [online]. 2018 [visited on 2018-01-22]. Available from: https://neo4j.com/docs/operations-manual/current/.

15. *Genesis block* [online]. Bitcoin Wiki [visited on 2018-01-22]. Available from: https://en.bitcoin.it/wiki/Genesis_block.

16. *from:"Satoshi Nakamoto"* [online]. The Mail Archive [visited on 2018-01-22]. Available from: https://www.mail-archive.com/search?l=cryptography@metzdowd.com&q=from:%22Satoshi+Nakamoto%22.

17. DAI, Wei. *b-money* [online]. US, 1998 [visited on 2018-01-22]. Available from: https://hbr.org/2017/01/the-truth-about-blockchain.

18. SZABO, Nick. *Bit gold* [online]. US, 2008 [visited on 2018-01-22]. Available from: http://unenumerated.blogspot.com/2005/12/bit-gold.html.

19. FINNEY, Hal. *Reusable Proofs of Work* [online]. US, 2007 [visited on 2018-01-22]. Available from: https://web.archive.org/web/20071222072154/http://rpow.net/.

20. *Vulnerability Summary for CVE-2010-5139* [online]. National Vulnerability Database, 2012 [visited on 2018-01-22]. Available from: https://nvd.nist.gov/vuln/detail/CVE-2010-5139.

21. COHEN, Brian. *Users Bitcoins Seized by DEA* [online]. US: Let's Talk Bitcoin, 2013 [visited on 2018-01-22]. Available from: http://letstalkbitcoin.com/users-bitcoins-seized-by-dea.

22. HILL, Kashmir. *The FBI's Plan For The Millions Worth Of Bitcoins Seized From Silk Road* [online]. US: Forbes, 2013 [visited on 2018-01-22]. Available from: https://www.forbes.com/sites/kashmirhill/2013/10/04/fbi-silk-road-bitcoin-seizure/#26806c612848.

23. LEE, Dave. *MtGox files for bankruptcy* [online]. US: BBC, 2014 [visited on 2018-01-22]. Available from: http://www.bbc.com/news/technology-25233230.

24. STAFF, Reuters. *Bitcoin exchange Bitstamp suspends service after security breach* [online]. US: Reuters, 2015 [visited on 2018-01-22]. Available from: https://www.reuters.com/article/us-bitstamp-cybersecurity/bitcoin-exchange-bitstamp-suspends-service-after-security-breach-idUSKBN0KF0UH20150106.

25. RUSSELL, Jon. *First China, now South Korea has banned ICOs* [online]. US: Jonas David Nick, 2017 [visited on 2018-01-22]. Available from: `https://hbr.org/2017/01/the-truth-about-blockchain`.

A Software manual

- **Requirements** - Ubuntu/Debian, Go1.10+
- **Put** - move project to `src` folder in `GOPATH`
- **Compile** main.go - `go build -i -o main main.go`
- **Run** main - `main -help`
- **Add path arguments** accordingly
- **Parsing** - start program with set path arguments and `-parse`
- **Clustering** - start program with set path arguments and `-cluster`
- **Deduplicate** - after clustering deduplicate lines in `address_node.csv` using `sort` and `uniq`
- **Remove** - remove `address_node.csv` and rename `address_node_dedup.csv` to the deleted file
- **Add** - add headers from project file `docs headers` to *CSV* folder
- **Download** - download latest `.tar` Neo4j for linux
- **Unpack and edit** - unpack downloaded `.tar` and edit line `dbms.directories.data` in file `conf/neo4j.conf` to the path of database
- **Move** - move script from project `docs/neoimport.sh` to `neo4j/bin`
- **Execute** - start import by running `neoimport.sh` with path to *CSV* files specified as first argument

B *CSV* import files headers

- **Block nodes**
  `:ID(BlockHeight-ID),blockHeight:long,timestamp:long`
- **Block to block relationships**
  `:START_ID(BlockHeight-ID),:END_ID(BlockHeight-ID)`
- **Transaction nodes**
  `:ID(TxHash-ID),txHash,fee:long`
- **Block to transaction relationships**
  `:START_ID(BlockHeight-ID),:END_ID(TxHash-ID)`
- **Address nodes** (`LABEL` defines which address format does this address have - P2PKH, BECH32...)
  `pubHash:ID(AddrPubHash-ID),firstAppearance:long,:LABEL`
- **Input address to transaction**
  `:START_ID(AddrPubHash-ID),amount:long,:END_ID(TxHash-ID)`
- **Output address to transaction**
  `:START_ID(TxHash-ID),amount:long,:END_ID(AddrPubHash-ID)`
- **Heuristic nodes** (`LABEL` defines which heuristics passed - shadow, change...)
  `:ID(HeuristicId-ID),timestamp:long,:LABEL`
- **Heuristic to address relationships**
  `:START_ID(HeuristicId-ID),:END_ID(AddrPubHash-ID)`
- **Cluster nodes**
  `:ID(ClusterId-ID)`
- **Cluster to heuristic relationships**
  `:START_ID(ClusterId-ID),:END_ID(HeuristicId-ID)`

C Inception of distributed ledger technology

In the year 2008 anonymous group of people or a single developer known as Satoshi Nakamoto published a document [1] which described distributed ledger technology. Two months later on 3 January 2009, the first bitcoin block was mined [15] by Satoshi, who released the first version of bitcoin [16] with source code. This alpha version was able to mine bitcoin and send currency to other people [16] using their public IP address or bitcoin address. One of the first people that used this application were creators of similar projects regarding the distributed monetary system, like Wei Dai's "b-money" [17], Nick Szabo's bit gold [18] or an alternation of bit gold named Reusable Proofs of Work by Hal Finney [19].

In 2012 a bug was found in the protocol [20], that allowed to bypass transaction verification and thus let users create an unlimited amount of bitcoin. This vulnerability was spotted and fixed on the same day, and blockchain hard forked to the updated version, where artificially created money was not present. There has been found no more significant exploits since then.

The great part of bitcoin was used for black-market trading. The US Drug Enforcement Administration stated that certain amount of bitcoins was seized, as they were used to pay for illegal drugs [21]. This was the first time a government took the virtual currency. Only four months after this event in October 2013, 26,000฿ was seized by FBI during the apprehension of Ross William Ulbricht, the owner of the illegal drugstore website called Silk Road [22]. Amount of the successful theft attempts risen too. By the February 2014 around 744,000฿ was stolen from the Mt. Gox, by the anonymous hacker because private keys of the wallet were not saved securely. The exchange had to liquidate [23]. Not even a year after bankruptcy about 19,000฿ was stolen from another significant exchange Bitstamp [24]. The exchange had to shut down to investigate the exploit. After a week of improving security, the exchange was restarted, and customers have not lost any funds.

The year 2017 and 2018 brought many changes to the regulations of the virtual currencies. Japan, Russia, Estonia, Norway and many other countries are showing open arms for the cryptocurrencies. On

the other side countries like China and South Korea are not fond of this idea and are aiming to remove crypto businesses from the country [25].