FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Cryptocurrency analysis support for ClueMaker
Student:	Bc. Matěj Adamec
Supervisor:	Ing. Marek Sušický
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

The main goal is to develop an application that will allow users to analyze chosen cryptocurrencies within ClueMaker.
Analyze and describe the structure of cryptocurrency transactions and describe the ways to retrieve information about the transactions in chosen cryptocurrencies (Bitcoin and Ethereum).
Design and implement a connector for a chosen graph database, which will be used for storing the data about the transactions. Propose support tools to track the flow of cryptocurrencies, determine if the coins do not come from stolen wallets, etc. Describe possible optimizations to make working with tools faster and easier (if the wallet is empty, it is possible to remove it and connect the transactions directly, etc).

Electronically approved by Ing. Michal Valenta, Ph.D. on 12 October 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Cryptocurrency analysis support for ClueMaker

Bc. Matěj Adamec

Department of Software Engineering
Supervisor: Ing. Marek Sušický

May 5, 2022

Acknowledgements

My thanks belongs to the supervisor Ing. Marek Sušický, who provided me with valuable insights and right questions. My thanks also go to the Profinit EU s.r.o. company that provided infrastructure to develop the application. Lastly, I would like to thank to my family, friends and those who provided moral support and encouragement.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 5, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Matěj Adamec. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Adamec, Matěj. *Cryptocurrency analysis support for ClueMaker.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato diplomová práce se věnuje analýze kryptoměn, konkrétně Bitcoin a Ethereum. Zdroje dat o transakcích na daných blockchainech jsou popsány a jsou navrženy a implementovány způsoby pro extrakci dat a uložení v grafové databázi JanusGraph. Následně práce popusije implementaci konektoru pro aplikaci ClueMaker, vyvýjenou firmou Profinit EU s.r.o., který umožní prohlížení a analýzu dat.

Klíčová slova ClueMaker, kryptoměny, transakce, blockchain, bitcoin, ethereum.

Abstract

This master thesis deals with the analysis of Bitcoin and Ethereum cryptocurrencies. The multiple data sources are analyzed, and an application that extracts them and stores the acquired structures in a JanusGraph database is designed and implemented. As part of the thesis, the application called ClueMaker, from company Profinit EU s.r.o. is extended for connector, enabling communication with JanusGraph database and thus allowing the users to explore the stored nodes and edges.

Keywords ClueMaker, cryptocurrencies, transactions, blockchain, bitcoin, ethereum

Contents

List of Figures

List of Tables

Introduction

Motivation and objectives

In 2008 an anonymous writer named Satoshi Nakamoto published his white paper titled "Bitcoin: Peer-to-Peer Electronic Cash System". This thesis started the era of blockchains and cryptocurrencies.

However, the idea of anonymous e-cash protocols was introduced already in the 1980s and 1990s, but they relied on a centralized intermediary. After that, in 1998, the project b-money became the first proposal to introduce the idea of creating money through solving computational puzzles as well as decentralized consensus without details of implementation. In 2005, Hal Finney introduced a concept of "reusable proofs of work", a system which uses ideas from b-money together with Adam Back's computationally difficult Hashcash puzzles to create a concept for a cryptocurrency, but once again fell short of the ideal by relying on trusted computing as a backend. Then Satoshi Nakamoto came up with a combination of the ideas on managing ownership, consensus algorithm and implementation of a trusted computing network[23].

Since then, almost everybody could hear about this kind of technology and many companies and even states have started to consider or integrate blockchain into their businesses. The more widespread and used technology is, the more analytical tools and applications are needed.

Cryptocurrencies like Bitcoin and Ethereum are the most famous representatives of blockchain technologies. Today, millions of people use them to invest, pay for goods, or power their applications. And companies, especially in the financial sector, look for tools that would help them understand, analyze and monitor the transactions and movement of the funds in these blockchains. Such analysis can be used in fraud detection, user scoring, monitoring of criminal activity, understanding the market and finding new business opportunities.

For software companies like Profinit, it is an excellent opportunity to create new products that will help their partners with their needs or attract new

clients. Profinit develops a product called ClueMaker that offers powerful data analytics with relation visualization and is already being used by banks, journalists, and fraud investigators. And many have already shown interest in cryptocurrency-related products. Its features are ideal for the visualization of relations and flow in cryptocurrencies.

This work aims to analyze and describe the structure of Bitcoin and Ethereum cryptocurrencies and create an application that will be able to retrieve and store the blockchain data structures in the graph database. Stored data will be then possible to visualize and analyze via ClueMaker.

Chapter 1

Analysis

In this part of the thesis, I will introduce the basics of blockchain technology, Bitcoin and Ethereum, with the goal of introducing the topic and associated fields. Then I will analyze options for data retrieval and data storing. And at the end of the chapter, I will introduce the ClueMaker tool that I will, later in the thesis, extend by a connector for a chosen database.

1.1 Blockchain and cryptocurrencies

In the book by Chris Dannen the blockchain is described as follows:

In the abstract, open-source blockchain networks such as Ethereum and Bitcoin are economic systems in software, complete with account management and a native unit of exchange to pass between accounts. People call these native units of exchange coins, tokens, or cryptocurrencies, but they are no different from tokens in any other system: they are a form of money (or script) that is used only within that system.

Or as said in the book "Blockchain Technology Applications and Challenges"[1], Blockchain Technology is a distributed, decentralized, peer-to-peer network to store network transactions without any third party. This solution allows the nodes to verify and manage the network. The cryptographic hashing mechanism used in blockchain lets the data in blocks be tamper-proof and secure.

- **Ledger:**

  An open, append-only ledger is used by blockchain to record the transaction history. The data in this ledger cannot be modified, unlike the traditional databases.

- **Secure:**

  Blocks in blockchain are cryptographically linked, and that is why the data can not be tampered with, thereby assuring the security of information over the blockchain.

- **Shared:**

  The public ledger can be shared among the network users, assuring transparency among the users.

- **Distributed:**

  The blockchain is distributed among the network users, which makes it robust against attacks. By increasing the number of nodes in the network, the security of the information on the blockchain is high.

Blockchain can be permissioned or permissionless. In the permissioned blockchain, only authorized users can create and publish new blocks. The project can be open or closed sourced. The read and write access can be limited. They are not necessarily using consensus methods to avoid malicious users publishing blocks. There are no rewards for publishing new blocks.

On the other hand, permissionless blockchains allow everyone to publish new blocks. The platform is open-source, and anyone can download it. There is no restriction on reading or adding new transactions. Users need to adapt the consensus mechanism. For publishing new blocks, users are rewarded [2].

The blockchain is a combination of three technologies. These three components are Peer-to-peer networking, asymmetric cryptography and cryptographic hashing.

Peer-to-peer networking means a group of computers, such as a torrent network or blockchain network, can communicate among themselves without relying on a single central authority and, therefore, not presenting a single point of failure. There are also other advantages like improved network efficiency, privacy, and scalability. Such communication brings many issues. For example, slower speed in the context of the whole network (sending data from one node to all other nodes in the network) and related to the speed is an issue with synchronization. As the nodes in the network are scattered all over the world, the protocol needs to ensure that the majority of the nodes will operate with identical data.

The second technology is asymmetric cryptography. The way for the computers to send a message is encrypted for specific recipients such that anyone can verify the sender's authenticity, but only intended recipients can read the message contents. In Bitcoin and Ethereum, asymmetric cryptography is used to create a set of credentials for the user's account, to ensure that only the owner can transfer the coins or sign the transactions. In Bitcoin and Ethereum, the Elliptic Curve Digital Signature Algorithm (ECDSA) is used to create public keys from the private key.

The third component of blockchain is cryptographic hashing. Hashing is a cryptographic technique that simply converts data of any size into o unique fixed-size output - a fingerprint. Input can be anything from text to images, and the result is a fixed-size alphanumeric string. The hashing functions are uni-directional, meaning it's hard (almost impossible) to find input based on

output. This property is called pre-image resistant. Hashing functions are also second pre-image resistant, meaning that knowing one input for some output does not help find different inputs that would result in the same result as the first one. It is highly infeasible to find two inputs generating identical hash output. As the hash is a small and unique fingerprint, it is easy to compare large datasets and a secure way to verify that data has not been altered. In a blockchain, hashing assures the users that transmitted data have not changed. Most blockchain implementations use SHA (Secure Hash Algorithm) that generates an output of size 256 bits.

Together, these three technologies can mimic a virtual machine with a database that is decentralized and stored in the nodes of the network[1].

1.1.1 Blockchain structure

Cryptocurrencies share common structures that together create the blockchain. Those structures are block, transaction, address, and wallet. The implementation and data structures can be different across the blockchains.

Blocks are structures that represent a collection of transactions, and when linked, they create a ledger. Each block contains transactions and the previous block's hash and can't be changed without modifying all blocks published after it. That is why the blockchain is secure.

The Block header consists of 6 parts that are needed. Version, previous block hash, merkle root, timestamp, difficulty target and nonce. Version represents the version number to track software or protocol upgrades. The previous block references the hash of the previous block in the chain. Merkle root is the hash of the root of the merkle tree consisting of transactions in the block. A timestamp records the approximate creation time of the block. Difficulty, size or weight is given by the Proof of Work algorithm for the block. A nonce is the last part that is needed. Miners are trying to create the blocks by looking for a nonce value to generate a hash that satisfies specific requirements after hashing with all six parts[2].

In simple terms, a transaction tells the network that the coins' owner has authorized the transfer of some of those coins to another owner. The new owner can now spend these coins by creating another transaction that authorizes transfer to another owner, and so on, in a chain of ownership.

An address is a unique identifier that serves as a virtual location where the cryptocurrency can be sent. People can send the cryptocurrency to addresses similar to how fiat currencies can often be sent to email addresses or bank accounts. However, the Bitcoin address is not intended to be permanent but just a token for use in a single transaction. Unlike a digital wallet, a Bitcoin address cannot hold a balance[3].

1.1.2 Wallet

Wallets are vital components in blockchain and cryptocurrencies. From the name "Wallet", someone could think that it is some storage where all owned coins are stored. But it is not. Wallets are more like keychains.

Now I will describe the basics of wallets and their kinds. A blockchain wallet is a storage (structured files, small database) containing private and public key pairs. Private keys are the most important because they are used to sign new transactions, and without them, users can't operate with their assets inside the network. The keys are mainly generated randomly. However, it is possible to choose one, but it is not recommended and secure as it can be guessed more easily [4].

Public keys are generated from private keys that can be used as addresses or transformed into addresses. They are safe to be known by others. Each blockchain has its way of creating public keys, using them and creating addresses or other objects.

Keys example:

- Private Key:

  5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

- Public key:

  1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x

Important to note for my use case is that the wallets don't have identifiers. At least not public ones, and they are not recorded inside the blockchain. That means the data about wallets can be retrieved from different sources, which only assume the addresses or public keys are part of one wallet. In this thesis, we will operate with wallets, and we will assume that one user has only one wallet and all his keys are inside.

Blockchain wallets can be divided by multiple parameters. Most of the time, crypto wallets are divided into two groups. Hot and cold wallets.

Most users use hot wallets. They are wallets connected to the internet and generally less secure. The pairs of keys are stored in the cloud. On the other hand, hot wallets are more user friendly and accessible and do not require a user to own specialized equipment.

The second type of wallet is a cold wallet. Cold wallets are designed for existing in offline mode. They are mainly used to store keys to larger amounts of funds that won't be moved frequently. As they are not relying on a connection to the internet, they are mostly more secure than hot wallets.

It is possible to use both types of wallets to use their advantages. Hot and cold wallets can be divided into several more categories, for example, software, hardware, mobile, web, desktop, paper and more types of crypto wallets. Wallets connected to the network usually calculate the number of

digital assets owned by the user, provide an interface to work with the assets and easily trade them with others [6].

1.1.3 Transaction

The basic understanding of transactions is that transactions are interactions between two entities. They allow users to transmit their rights to information to another user publicly over the network. In cryptocurrencies, they serve as a mechanism to transfer funds between two users. Or simply transfer the blockchain state from one to another.

Transaction data generally include information about the transaction input, output and sender's signature. When the transaction is published, the signature is checked, and miners validate if the sender has the right to operate with the input. Transaction in a blockchain does not have to transfer coins, but it can transfer data. It can publish, process and store data in the blockchain. Blockchain technology has many use cases that transactions can implement.

Transferring rights to operate with some information using transactions brings many problems that have to be dealt with by the blockchain. The main problem is double-spending. The situation where the owner of some assets tries to spend them multiple times. In Bitcoin and Ethereum, the issue is mitigated by the mining mechanism that ensures the validity of the transactions in the block using the Proof of Work mechanism[1].

1.1.4 Address

Addresses in blockchains are mostly short strings composed of alphanumeric characters. They provide transaction points for the sender and receiver. They are created from a public key using a hashing function. Each blockchain has its parameters and requirements for the addresses and can work with addresses differently.

Address example:

- Bitcoin:

  1F1xcRt8H8Wa623KqmkEontwAAVqDSAWCV

- Ethereum:

  1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x

Not all transactions have to be made from one address to another. In Bitcoin, it is possible to create transactions for anybody who provides the correct input data to open the transaction. In Ethereum, it is possible to create transactions to another smart contract. So the addresses are created for users to be able to easily describe where the transaction needs to be sent

and with a small chance of making mistakes when entering it. And that is the reason why the addresses are mostly short.

1.2 Mining algorithms

Mining algorithms solve two issues with distributed networks. First is the issue of votes in the blockchain, as the network is distributed, and nodes need to consent to the validity of the chain and newly created blocks. The second issue is minting new coins and releasing them into the chain.

1.2.1 Proof of Work

Satoshi Nakamoto, in his white paper, introduced an algorithm called Proof-of-Work that solves both problems. The distribution of votes is solved with a one-CPU-one-vote system. This name can be inaccurate as the machine running the node can have multiple CPUs and still will be counted as one vote[11].

But the main idea is that if the majority of CPU power is controlled by honest nodes, the chain will grow, and attackers won't be able to modify it for their needs. In the white paper is shown that the probability of a slower attacker catching up diminished exponentially as subsequent blocks were added. Also, with an increasing number of miners, the chain is more secure as the attackers need to control over 50 % of running nodes. That is why is attack is called The 51% attack.

To create a new block in the blockchain, the miner needs to scan for values that, when hashed, the hash starts with a number of zero bits. In the Bitcoin network, the miners are looking for the value of nonce, which is part of the block. The mining process requires solving the inversion of a cryptographic function, and that can be achieved only by brute force. Meaning that miners try different values for the nonce attribute and hash the blocks and check the result. If the hash value is less than the target (begins with the correct number of zeros), the block is sent into the network, and other nodes verify if the newly added block is correct. Right now, miners have to make on average 2^{69} tries before a valid nonce is found[4].

The probability of the miner being successful depends on the ratio of its own computational power and the total computational power of all miners in the network. However, the mining process based on proof-of-work is energy and hardware intensive. And as the cryptocurrencies using the Proof of Work mechanism grows in popularity and the energy consumption grows, alternative consensus mechanisms are emerging.

Proof of Work is currently used in Bitcoin, Ethereum and many more cryptocurrencies and is the most common among the blockchains.

1.2.2 Proof of Stake

The Proof of Stake mechanism links block generation to the proof of ownership of a certain amount of digital assets in the blockchain. The "miners" are more validators than miners. Each user who decides to stake a certain amount (or greater) of coins will become a validator. With each new block, the validators are randomly selected to mine or validate the newly generated block, and after a specific number of validators verify the validity of the block, it is closed and added to the ledger. The probability of being chosen to validate the block is related to the amount of staked assets[4].

For example, the proposed mechanism for Ethereum will require 32 ethereum coins to be staked by a user to become a validator. And each block will require at least 128 users to approve certain parts of the block. If a validator approves adding a block with inaccurate information, they lose some of their staked assets as a penalty.

The system stands on the idea that users with a large share of assets in the network are more likely to be trustworthy. This mechanism is considered to be more energy-efficient. The most popular representatives using Proof of Stake are blockchains Solana, Cardano and Algorand[7].

1.2.3 Proof of storage

The most popular consensus mechanisms are Proof of Stake and Work. But there are also different mechanisms that try to utilize different valuable commodities than processing power and staked assets. An example of such a mechanism is Proof of Storage (also as proof of capacity or proof of space). Here the resources are not CPU cycles but the amount of actual non-volatile memory space the miner must use to compute the proof. The miners are motivated to devote hard drive capacity as those who dedicate more disk space have a proportionally higher chance to mine a new block and obtain the reward. During mining, the dedicated space is used to solve a presented challenge.

The most popular tokens are Filecoin, Arwaeve and BitTorrent-New[9].

1.2.4 Proof of Authority

The last consensus mechanism I want to mention is Proof of Authority. It is suitable for private networks where preselected real authorities or entities are allowed to control the addition of new blocks to the ledger. Only those entities hold a set of private keys that are used to sign new blocks, acting as trusted signers.

There are more mechanisms introduced to the community, like Proof of Burn or some combinations of previously mentioned, but they are not so used as already mentioned [10].

1.3 Bitcoin

Bitcoin is a collection of concepts and technologies that form the basis of a digital money ecosystem based on blockchain. Units of currency called bitcoins are used to store and transmit value among participants in the bitcoin network. One bitcoin can be divided down to eight decimal places as satoshis. One satoshi can't be divided anymore and is the smallest currency unit in bitcoin. Bitcoin users communicate with each other using the bitcoin protocol primarily via the internet, although other transport networks can also be used. The bitcoin protocol stack, available as open-source software, can be run on a wide range of computing devices, including laptops and smartphones, making the technology easily accessible and making it permissionless blockchain [4].

As mentioned in the motivation, bitcoin was first introduced in 2008 by an anonymous writer under the pseudonym Satoshi Nakamoto. The cryptocurrency did not hold any real value, but since then, the value of one coin has exceeded the value of 65,000 USD in February 2021. El Salvador in 2021 became the first country to make Bitcoin legal tender, encouraging citizens and businesses to transact in cryptocurrency. Bitcoin became an alternative to the traditional banking system, where the transactions are processed by a single authority[11].

Specific properties of bitcoin

Bitcoin, like other blockchains, shares a common structure. It contains blocks, transactions and addresses. But each blockchain comes with its specifics, and I will describe the main differences here. The data I will describe is the data accessible from the bitcoin core storing the full ledger.

Block in the bitcoin holds the common data to store information about the block, data to validate it and its transactions. Every block contains at least one transaction containing the transfer of mined coins to the miner of the block. The bigger difference between bitcoin and ethereum can be seen in transactions[4].

1.3.1 Transaction

Each transaction consists of data and one or more inputs, and one or more outputs. That said, the block contains at least one transaction containing the transfer of generated coins to the miner of the block. This transaction, called a coinbase transaction, is different from others in the block as it is always first, does not have a hash, and does not spend any other output.

The inputs and outputs are not related to any account or identity. They are amounts of coins locked by the owner, and the only user who knows the secret can unlock them and use them in the next transaction. As the transfer is done by unlocking some inputs and locking outputs, there is no wallet balance.

There are only unspent transaction outputs scattered in the blockchain. How the funds are transferred can be seen in the image 1.1.

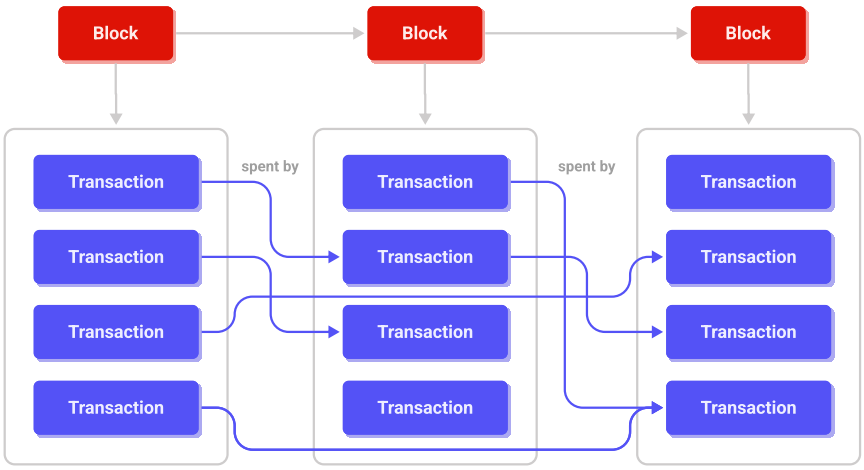

Figure 1.1: Bitcoin structure illustration

Transaction inputs, if not part of coinbase transactions, are, in simple terms, pointers to outputs with keys to unlock them. Pointers because they reference the transaction output, processed in previous transactions, by its hash and index in the list of outputs. The key consists of anything that is necessary to unlock the output locking script. It can be a public key with the signature from the private key. Or one or more signatures. Even just some specific value if the locking script is written to accept it. At the moment the transaction is stored in the blockchain, the referenced output is marked as spent and cant be used in any other transaction. There are more rules that specify when the output can be referenced by input, like a number of confirmations, the validity of the signing script etc.

An example of transaction input can be seen in the code example 1.1.

Transaction outputs are structures holding their hash that is unique, index inside the transaction, the value representing the coins and locking script. When the block containing new transactions is added to the chain, miners or users running full node add the outputs to the set of unspent outputs. A locking script is a program that, when presented with correct input, will result in TRUE or any other values that are considered as failures. The script is mostly created using the address of the user who should receive the coins. The basics of the locking script will be described in the following chapter.

```
{
   "txid": "25fa27b01c23b788a67a6f704407699239...",
   "vout": 0,
   "scriptSig": {
     "asm": "30440220462d63fc6db62da9c24312080...",
     "hex": "4730440220462d63fc6db62da9c243120..."
   },
   "sequence": 4294967295
 }
```

Code example 1.1: Shortened example of transaction input

As the inputs reference the outputs only with identifiers. They, do not contain any information about their value. This means the entire value of the output is spent. It can not be spent partially. Meaning if the user owns the output of the transaction with ten coins and wants to send five coins to another user and keep the rest, he needs to create a new transaction where the input will be that owned output containing the ten coins, and outputs will be two. One contains five coins and is locked so only the other user can open it, and the second one for which the key will save for himself.

An example of transaction output can be seen in the code snippet 1.2.

```
{
  "value": 100.00000000,
  "n": 0,
  "scriptPubKey": {
    "asm": "04bc3ee049bebf27e6e29403aeb... OP_CHECKSIG",
    "hex": "4104bc3ee049bebf27e6e29403aeb61a1c48ace...",
    "type": "pubkey"
  }
}
```

Code example 1.2: Shortened example of transaction output

1.3.2 Script language

Bitcoin uses its own script language created to be simple, compact and with cryptographic operations. It has many similarities to the programming language Forth. Just like Forth, it's stack-based, meaning it relies on the stack for passing parameters.

The language consists of constants, commands for flow control like ifs and returns, commands to control and move items on the stack, binary logic operators, arithmetic functions, cryptographic functions and reserved words. Even

with all the operators and instructions, the script language is intentionally limited and is not Turing complete. It is not meant to be a general-purpose language. It does not contain any loops or instructions for complex flow control. This ensures the language cannot be used to create an infinite loop, or other forms of "logic bomb" cant be constructed and released into the network where each transaction is validated by every full node.

All commands are listed in bitcoin documentation, but as a demonstration, I will show an example of such a script and how it works, and with it, I will introduce first of standard transaction scripts.

Common script instructions:

- OP_DUP - Duplicates the top item on the stack.
- OP_HASH160 - Hashes twice: first using SHA-256 and then a different hash function called RIPEMD-160
- OP_EQUALVERIFY - Returns true if the inputs are equal. Returns false and marks the transaction as invalid if they are unequal.
- OP_CHECKSIG - Checks that the input signature is valid using the input the public key for the hash of the current transaction.

Pay-to-PubkeyHash (P2PKH)

Now let's look at how the key from the input of the transaction can unlock the output it is referencing. In this example, I will talk about the Pay-to-PubkeyHash (P2PKH) script, which is the most common type of output script in Bitcoin. The output script looks like this:

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

The input of the transaction contains a part called scriptSig containing a string consisting of two substrings divided by space. The first is for the public key, and the second one is for the signature.

```
<signature> <pubKey>
```

When the script is executed, the locking script is pushed on the stack and then the scriptSig part of the input.

Then the script is executed. The state of the stack after each instruction can be seen on the image 1.2.

The P2PKH script is just one of many used scripts. There are other scripts commonly used in the bitcoin network.

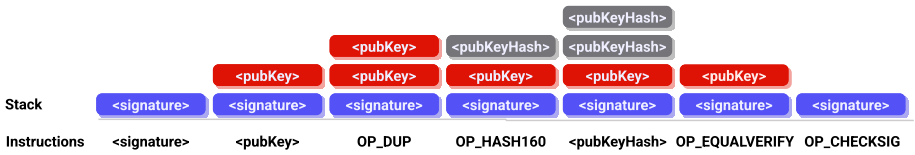

Figure 1.2: Illustration of execution of Pay-to-PubkeyHash script

Pay-To-Public-Key (P2SH)

The Pay-To-Public-Key script is a second commonly used type of script. The difference from the P2PKH is that it has a simpler form.

The locking script has the following form:

```
<Public Key> OP_CHECKSIG
```

And the unlock script is in form:

```
<Signature from Private Key>
```

The structure is similar to P2PKH but shorter without the OP_HASH160 and OP_EQUALVERIFY steps. Pay-to-Publickey is mostly used in coinbase transactions, generated by older mining software that has not been updated to use P2PKH[4].

Pay to Multi-Signature (P2MS)

The multi-signature script was created for use cases where it is wanted for multiple users to sign the transaction with their keys. The locking script creates conditions where N public keys are recorded together with a threshold that states the minimum of signatures provided to unlock the transaction. Such a function allows users to divide the responsibility for the possession of bitcoins among multiple people. Also, it lowers the chance of a single point of failure by making it substantially more difficult for the wallet to be compromised. It can also be used as a backup where the loss of one key doesn't lead to the loss of the assets[12].

For example, in 2-of-3 multi-signature transaction, the locking script will have the following form:

```
OP_2 <PublicKey1> <PublicKey2> <PublicKey3> OP_3 OP_CHECKMULTISIG
```

And the unlocking script then provides at least 2 signatures:

```
OP_0 <Signature A> <Signature B>
```

The OP_0 is inserted due to a bug in the CHECKMULTISIG instruction as it pops an extra item from the stack. It is there as a placeholder and is ignored.

For all transaction scripts that I introduced till now was in a way possible to talk about sending assets from one address to the other. But with P2MS, it is no longer possible to derive the address from the locking or unlocking script. That does not mean we can't track the flow anymore. It is just not possible to assign the address to it[13][15].

The P2MS solved a lot of issues, but it has some limitations. Due to the length of the public keys, the locking script can get big, so it is limited to up to 3 public keys. Also, not every wallet can create a multi-signature transaction and allows users to send assets only to address and not raw scripts. Also, the sender of the transaction needs to know all the public keys, and it's a burden if it is, for example, a client paying for some service or goods[14].

Pay-To-Script-Hash (P2SH)

The multi-signature feature comes with many advantages and some disadvantages, but the community came with another transaction script that replaced the P2MS. And that is P2SH. In 2012 the P2SH was introduced as a replacement for P2MS with properties that resolve its practical difficulties. The main focus is on the simplicity of using such a script. The complex locking script with multiple public keys is replaced with a single hash. Hash of the script that is then provided in the unlocking script. As nicely described in the book Master Bitcoin[4], the P2SH means "pay to a script matching this hash, a script that will be presented later when this output is spent.". This takes the burden from the sender and puts it on the receiver.

The scriptPubKey format:

```
OP_HASH160 <Redeem script hash> OP_EQUAL
```

The scriptSig format:

```
OP_0 <Signature A> <Signatur B>...<Signature X> <Redeem script>
```

The locking and unlocking script holds the redeem script in some form. Hash or the full script. The redeem script is the locking script from the P2MS transaction. It can look like the P2SH does not make the process less complex and simple. But let's look at the real value the sender of the coins must have known when creating the transaction with 1-of-2 signatures.

With P2MS:

```
OP_1 022afc20bf379bc96a2f4e9e63ffceb8652b2b6a097f63fbee6e
cec2a49a48010e03a767c7221e9f15f870f1ad9311f5ab937d79fcaee
e15bb2c722bca515581b4c0 OP_2 OP_CHECKMULTISIG
```

With P2SH:

```
OP_HASH160 748284390f9e263a4b766a75d0633c50426eb875 OP_EQUAL
```

Here it is obvious that for the sender of the money, it is much more comfortable to use P2SH. There is no need to share the entire locking script with the clients. There is only a need to share the address that is created from the script in a similar way to how addresses are created from public keys (more described in the following chapter about addresses in bitcoin)*learnscript*.

1.3.3 SegWit

SegWit or Segregated Witness is a change in transaction format. It was introduced in 2015 in soft fork under the name BIP-141 and fully deployed in 2017. The reason why I choose to mention this change in my thesis is how it changes the locking and unlocking of the transaction.

The upgrade has multiple issues it tries to solve. The main issues are protection against transaction malleability and solving a block size limitation problem that reduced the bitcoin transaction speed[17].

The transaction malleability

Each transaction has three parts. The input, output and signature that verifies that the sender is eligible to send the coins. It turns out that bitcoin's code allows digital signatures to be altered while a transaction is still waiting to get confirmed. The signature alteration can be done in such a way that if it's checked for validity, it will be still valid according to the rules of the network. But if it is hashed, the result is different[17]

This can be problematic for several reasons. First, if projects want to build second layer solutions on top of the bitcoin network, like the Lightning network, which will be discussed in the following chapter, it is important to make sure no one can alter the first layer since the one relies on the other.

Also, if users are trying to spend or accept unconfirmed funds, the alteration of transaction ids can cause issues.

Scalability

In the bitcoin network, a new block is created roughly every 10 minutes. Additionally, bitcoin's protocol limits the capacity of a block to 1MB, which limits

a block to around 2700 transactions on average (around four transactions per second). This creates a problem when a lot of people are trying to send coins simultaneously as the queue of transactions waiting to be processed gets longer and longer.

How the SegWit works

The segregated witness is a proposed change to how blocks are structured. Non-seqwit blocks, also known as legacy blocks, have a total of 1 MB of space for all of the block data. Inputs, outputs, signatures and additional scripts.

The SegWit blocks are, in fact, large 4 MB blocks that consist of a base transaction block that can have up to 1 MB in size and an extended block with a size of up to 3 MB. Segwit blocks move signatures and other data known as "the witness" outside of the base transaction block to the extended block. The witness data will still be transmitted, but it is placed inside the extended block. This allows for more transactions to fit inside the 1 MB base transaction block. The extended with an additional 3 MB includes all of the witness data that isn't mandatory in the base block[17].

Achieved goals

The new block format SegWit introduces achieves two major goals. First, it moves the digital signature outside of the base transaction block. This way, if someone changes the signature on the transaction, it won't affect the transaction id. This, in consequence, solves the transaction malleability issue. Second, it shrinks down the base transaction data. Since the witness data takes up to 65 % of the transaction size, moving it outside of the base block allows more transactions to fit inside that 1 MB block.

With the SegWit change also comes in the way how the blocks are measured. While legacy blocks are measured in size, SegWit blocks are measured in weight. This attribute is introduced in SegWit and is calculated on a per-transaction basis. The weight is calculated by the following formula 1.1

$$\text{transaction weight} = \text{Base transaction size} * 3 + \text{Full transaction size} \quad (1.1)$$

The legacy transaction weight is always equal to 4 times the transaction size. This motivates miners to prefer lighter SegWit transactions over legacy transactions since they can fit more of them inside a base transaction block. Which increases the potential miner's fee[18].

1.3.4 The Bitcoin Lightning Network

In this work, I am working with transactions as they are recorded on the blockchain. However, there are applications created on top of Bitcoin that

are trying to overcome its shortcomings and create use cases that I need to take into account. An example of such application is the Bitcoin Lightning Network.

It is a decentralized system for instant, high-volume micropayments that remove the risk of delegating custody of funds to a trusted third party.

Bitcoin is widely used; however, there are some drawbacks to its decentralized design. Processing and confirming transactions on the network can take up to one hour before it is irreversible. Micropayments, or payments less than a few cents, are inconsistently confirmed, and fees render such transactions enviable on the network today.

The Lightning Network solves these problems. It is one of the first implementations of a multi-party Smart Contract (programmable money) using bitcoin's built-in scripting. It works by locking funds into a two-party, multisignature bitcoin address. The current balance is stored as the most recent transaction signed by both parties, spending from the channel address. The transaction between storing the actual balance onto the network takes place off the network. To make a payment, both parties sign a new exit transaction spending from the channel address. All old exit transactions are invalidated by doing so.

As only the balance is current balance is recorded into the blockchain and not all transactions between the parties, it is impossible to track them. Only the sender and receiver can see all the necessary details[19].

1.3.5 Address

In the Bitcoin blockchain, the users are enabled to be pseudonymous, which means the transactions are available to the public, but their identities are not. Bitcoin addresses start with the digit 1 or 3. Like email addresses, they can be shared with other bitcoin users who can use them to send bitcoin directly to your wallet. Unlike email addresses, you can create new addresses as often as you like, all of which will direct funds to your wallet. Users can increase their privacy by using a different address for every transaction. There is practically (it exists but is too large) no limit to the number of addresses a user can create[4].

Bitcoin address is an invention that was intended to make it easier for recipients to provide senders with the information needed by the sender to construct a transaction script. Transaction data doesn't ever actually include an address. They contain public keys, script hashes and others, and the address can be, in some cases, created from available data[39].

But as of now, there are several standard address types. And I want to describe them in this section.

Legacy Addresses (P2PKH)

Legacy addresses are addresses starting with the character ”1”. It is created from the public key. To create such address, the wallet must:

- Create a hash160 of the public key.
- Prepend a prefix that can be different for testing networks, but for mainnet is equal to 0x00.
- Append checksum of the hash.
- Then encode the string with Base58.

The described process is nicely shown in the picture 1.3

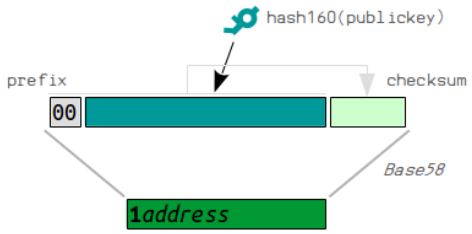

Figure 1.3: Process of creating P2PKH address from the public key[21]

This address can be easily distributed to other users and they can create transactions where the outputs will be sent to this address. To send the coins to the address, the wallet will decode the address with Base58 and will insert the hash of the public key into the lock script. Can be seen in the illustration 1.4

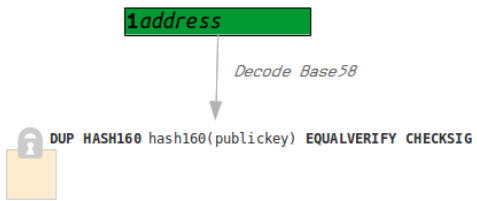

Figure 1.4: Process of creating P2PKH locking script from address[21]

Addresses starting with 1 are considered legacy and nowadays are used only by old wallets because the fees are larger than for P2SH or P2WPKH addresses.

Pay to Script Hash Addresses (P2SH)

As described in previous sections, the P2SH address is not created from the public key as P2PKH, but from the hash of the script. The process is very similar to the process of creating the legacy address but with small differences.

- Create a hash160 of the locking script
- Prepend a prefix that is for mainnet 0x05
- Append checksum of the hash
- Then encode the string with Base58

The first difference is we are hashing the script with the spending conditions and not the public key. And the second difference is that we use the prefix 0x05, which will result in the character "3" at the start of the address. Creating and using such an address can be seen in images 1.5 and 1.6.

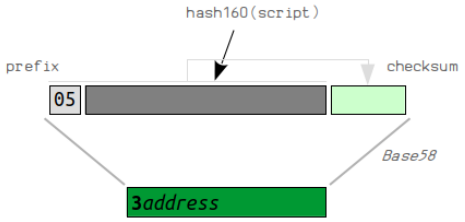

Figure 1.5: Process of creating P2SH locking script from address[21]

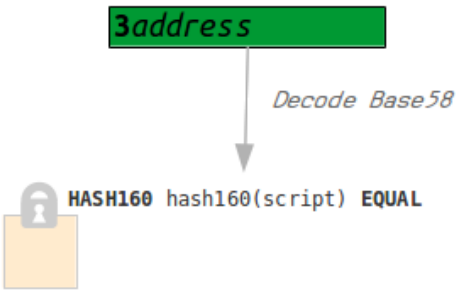

Figure 1.6: Process of creating P2SH locking script from address[21]

Pay to Script Hash Addresses are used in new transactions as they allow users to use advantages of SegWit and save roughly 26 % on transaction fees.

Native SegWit Address (P2WPKH)

A native P2WPKH address has the prefix ”bc1q” for the Bitcoin mainnet. It uses the same public key format as P2PKH transaction, with a very important difference: the public key used in P2WPKH must be compressed, that is 33 bytes in size, and starting with a 0x02 or 0x03. The P2WPKH scriptPubKey is always 22 bytes. It starts with a OP_0, followed by a canonical push of the key hash.

On the image 1.7 can be seen how the address can be created from the wallet using derivation paths. Users can derive keys in any way they want. But to help with compatibility between wallets, there is a common structure for how we derive keys for use in a hierarchical deterministic wallet.

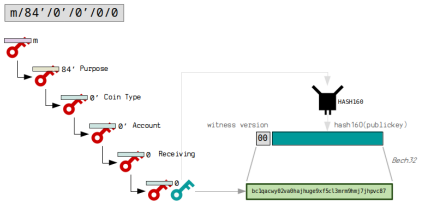

Figure 1.7: Process of creating P2WPKH address[22]

1.4 Ethereum

The term ”Ethereum” can be used to refer to three distinct things: the Ethereum protocol, the Ethereum network created by computers using the protocol, and the Ethereum project funding development of the aforementioned two. On the heels of bitcoin, Ethereum has become its own macrocosm, attracting enthusiasts and engineers from numerous industries. Many of civilization's most nagging imperfections could become the domain of blockchain's killer apps, and the Ethereum protocol (which was derived from Bitcoin and extended) is widely considered to be the network where these ”distributed” apps will spring up[1].

In contrast with Bitcoin, Ethereum offers more than transferring funds and storing small pieces of information in the blockchain. Yes, there are more applications on top of Bitcoin, but as the script language is not Turing complete and thus limited, there are limitations that applications have to accept and work with them. And some of these limitations are solved in the Ethereum.

As for the data structure, it works almost the same way that Bitcoin works. However, the difference in Ethereum is that it has a built-in Turing-complete programming language

1.4.1 Ether

Ethereum, similarly to Bitcoin, has its own currency unit. It is called ether, in the short form ETH. One ether can be divided into 10^{18} units called Wei. One Wei cannot be divided anymore. The Wei unit is important for me as I will work with the data inside the blockchain, and the value transfers are listed in Wei units. In some literature, we can find units like kilowei, megawei or microether as we are used to with kilometres and other units.

1.4.2 Accounts

Before I describe transactions and the programming languages and their use in Ethereum, I will introduce accounts. Bitcoin uses unspent transaction outputs to represent the actual state of the network. In Ethereum, the state consists of objects called accounts. Each account can be identified by 20-byte address. Transactions are the direct transfer of value and information between accounts.

The account have following parts:

- Nonce: a counter to ensure the transaction is processed only once
- Balance: account's current ether balance
- Code: account's contract code
- Storage: account's storage

These accounts can be divided into two types. Externally owned accounts and contract accounts.

Externally owned accounts (EOAs)

EExternally owned accounts are almost similar to addresses in the Bitcoin. As for the data parts, it has no code inside and empty storage. External accounts can send messages from an account by creating and signing a transaction. To create such an account is free. And transactions between externally-owned accounts can only be ETH/token transfers. Similarly to the Bitcoin address, the user has to prove he is the owner of the account. And it can be proven by presenting a signature created from the private key. From the two account types, only EOAs can initiate transactions[1].

Contract accounts

Contract accounts are something new and not present in Bitcoin. They are accounts inside the network containing smart contract code that can be executed when triggered by a message or transaction. The code inside the account can

read and write into internal storage and send other messages or create contracts in turn. Contracts can also send and receive ether, just like externally owned accounts[23].

However, when a transaction destination is a contract address, it causes that contract to run in the EVM, using the transaction, and the transaction's data, as its input. In addition to ether, transactions can contain data indicating which specific function in the contract to run and what parameters to pass to that function. In this way, transactions can call functions within contracts[5].

More about the smart contracts will be discussed in the following sections.

1.4.3 Transactions

Transactions are signed messages originated by an externally owned account, transmitted by the Ethereum network, and recorded on the Ethereum blockchain. They are the only construct that can initiate a change of state or trigger the execution of the contract in the EVM. Without a new transaction, the state would not change as Ethereum does not change autonomously.

Each transaction consists of the following parts:

- Nonce, used to prevent double execution, issued by the EOA.
- Gas price, the originator is willing to pay for one unit of gas.
- Gas limit, specifying the maximum amount of gas the originator is willing to spend for this transaction.
- Recipient, in form of account address.
- Value, the value of ether to send to the recipient.
- Data, in form of binary data payload.
- Signature, ECDSA digital signature of the originating EOA.

When using some software to display the data about the transaction, it will contain more information like the sender address, block number and more, but the software adds them to make it easy for the user.

From the execution view, the transactions are atomic, regardless of how many contracts they call. They are always executed in their entirety or not at all. A failed transaction is still recorded as having been attempted, and the ether spent on gas for the execution is deducted from the originating account, but it otherwise has no other effects on the contract or account state[5].

1.4.4 Gas

As mentioned in the previous section, the transaction contains attributes specifying gas price and gas limit.

Gas is a fuel of Ethereum. Gas is not equal to the ether, but it is a separate virtual currency. It cannot be sold like ether, but it can be converted.

Ethereum uses gas to control the number of resources that a transaction can use since it will be processed on thousands of computers worldwide. As the programming language used in Ethereum is Turing complete, the model requires some protection against logical bombs and endless loops. In Bitcoin, it is done by limitation of the language. In Ethereum is done by using gas. If anyone wants his transaction to be processed by the network, he needs to pay the miner for the computational resources. If it's a simple transfer of money, it is cheaper than executing a complex smart contract.

The gas price field in the transaction allows the transaction creator to set the price he is willing to pay in exchange for the gas. Miners prefer transactions with higher gas prices, as they will get higher rewards. Due to this fact, transactions with higher gas prices will be processed faster. During periods of high demand for space in a block, some transactions can be unconfirmed for a longer period of time. On the other hand, a transaction with a gas price set to zero can be processed when the demand is low. The price is measured in Wei per gas unit.

To avoid mentioned logical bombs and endless loops that would harm the network, the gas limit is introduced. Each operation has its own price in gas, and if the gas used to process the transaction code reaches the gas limit, the transaction ends, and execution stops. So each transaction has to stop due to the reaching the end of the code or due to depleting all available gas.

Someone could ask why Ethereum does not use the ether when it is already in the protocol. Gas is a separate currency from ether in order to protect the system from the volatility that might arise along with rapid changes in the value of ether[23].

EIP-1559

In August 2021, the London Hard Fork took place, introducing improvement in the Ethereum (under-identification EIP-1559) and changing Ethereum's fee market mechanism. This mechanism tries to solve the first-price auction and make gas fees more predictable, resulting in a more efficient transaction fee market. And due to this auction, some transactions are not included for a longer period of time.

Before the improvement, the transaction fee was calculated as follows:

```
Gas units (limit) * Gas price per unit
```

With EIP-1559, there is a discrete "base fee" for transactions to be included in the next block. For users or applications that want to prioritize their transaction, they can add a "tip," which is called a "priority fee", to pay a miner for faster inclusion. The equation then has the following form:

```
Gas units (limit) * (Base fee + Tip)
```

The base fee is always burned and should have a deflationary effect. Also, the base fee is not a set value. It is a variable value based on previous blocks. The base fee is calculated by a formula that compares the size of the previous block (the amount of gas used for all the transactions) with the target size. The base fee will increase by a maximum of 12.5 % per block if the target block size is exceeded. This exponential growth makes it economically non-viable for block size to remain high indefinitely[24].

One of the main ways EIP-1559 accomplishes these mitigations is by allowing blocks to become 200 % full, i.e. filled up to double whatever the reigning Ethereum gas limit is. This extra flexibility will grant Ethereum better capacity to support transaction demand, leading to shorter transaction wait times and clearer gas price estimations[25].

1.4.5 Smart Contracts

I described transactions, accounts, and gas. All mentioned parts are combined in the smart contracts. The term smart contract was introduced by Nick Szabo in the 1990s, who defined it as "a set of promises, specified in digital form, including protocols within which the parties perform on the other promises.". In this thesis, I will use this term to refer to immutable computer programs running deterministically in the context of EVM.

Smart contracts are usually written in high-level languages like Solidity, Vyper, Serpent or Mutan and then compiled to low-level bytecode that runs in the EVM. Once the program is compiled, it can be deployed on Ethereum using a special contract creation transaction. This transaction is special as it is sent to contract creation address 0x0.

Here is several notes for the contract:

- The address for such contract is derived from the transaction attributes.
- Unlike with EOAs, there are no keys associated with an account created for a new smart contract.
- The execution can be initiated only by the transaction or contract that is directly triggered by transaction or indirectly as part of a chain of contract calls.
- The contracts cannot run in parallel.

- The contract is atomic, meaning the changes are recorded entirely or in case of failure rolled back.
- The code inside cannot be changed after recorded in the blockchain.
- By default, it cannot be deleted, unless programmed to listen for opcode SELFDESTRUCT by creator.

1.4.6 Ethereum Virtual Machine

The core of the ethereum protocol is Ethereum Virtual Machine, shortly EVM. As the name might suggest, it is similar to Java Virtual Machine. It also interprets bytecode-compiled programming languages such as Solidity, Serpent or Mutan.

The built-in language in Ethereum allows anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions. Smart contracts, cryptographic "boxes" that contain value and only unlock it if certain conditions are met, can also be built on top of the platform, with vastly more power than that offered by Bitcoin scripting because of the added powers of Turing-completeness, value-awareness, blockchain-awareness and state[23].

I describe EVM as Turing's complete state machine, but some literature labels it with the word "quasi". Meaning the process is limited to the finite number of steps by the available given code. As such, the halting problem is "solved" (all program executions will halt), and the situation where execution might (accidentally or maliciously) run forever, thus bringing the Ethereum platform to a halt in its entirety, is avoided.

The EVM has its own instruction set consisting of arithmetic instructions, stack operations, flow control (richer than the bitcoin script language), logic, block and environmental operations. As we can see, the language contains operations with block and environment, and that is something not present in Bitcoin. It can operate with the block data as the hash, timestamp, number, difficulty or gas limit. From the environment information, it can access the address, balance, caller and more[23].

In the table 1.1 are some examples of opcodes and how much gas the execution costs.

Solidity

I introduced the EVM and its low-level opcodes, and now I want to describe the selection of the high-level languages used for programming smart contracts.

The most popular language for programming smart contracts is Solidity. It has vast community support, and a lot of developer tools like Remix and Truffle. It is an object-oriented language. Strongly influenced by some existing languages like JavaScript and C++. It is a statically typed language,

Table 1.1: examples of opcodes and their gas cost [23]

Opcode	Name	Description	Gas
0x00	STOP	Stops execution	0
0x01	ADD	Addition operation	3
0x08	ADDMOD	Modulo addition operation	8
0x31	BALANCE	Get balance of the given account	400
0x54	SLOAD	Load word from storage	200
0x56	JUMP	Alter the program counter	8
0xa0	LOG0	Append log record with no topics	375
0xf0	CREATE	Create a new account with code	32000

meaning the developer has to define the type of value in a variable so that the compiler knows what type of data to expect. This is essential when developing a deterministic application[27].

```
pragma solidity >=0.7.0 <0.9.0;

contract Promise {
    string promiseMessage;

        //set function
    function setPromise( string memory _promise) public{
        promiseMessage=_promise;
    }
    //get function
    function getPromise()public view returns(string memory){
        return promiseMessage;
    }
}
```

Code example 1.3: Solidity code example [26]

Vyper

Vyper is a contract-oriented programming language. It was specifically developed to address the security issues which were there in Solidity. It is strongly influenced by the python programming language. Unlike Solidity, Vyper does not have some object-oriented concepts like inheritance, which is popularly referred to as contract-oriented or transactional programming. The main motive was to make contracts auditable and more secure, making less error-prone contracts. It is a strongly typed language, which means it does not allow to

use of one type of data type as another. Also, a big difference in Vyper is the need to specify the length of strings. That is not needed in the Solidity[26].

```
# @version ^0.2.12

promiseMessage: public(String[100])

@external
def setPromise(_promise:String[100]):
self.promiseMessage=_promise

@view
@external
def getPromise()->String[100]:
return self.promiseMessage
```

Code example 1.4: Solidity code example[26]

Serpent

The language, as suggested by its name, is designed to be very similar to Python. It is intended to be maximally clean and simple, combining many of the efficiency benefits of a low-level language with the ease of use in programming style and at the same time adding special domain-specific features for contract programming.

On the site `https://eth.wiki/archive/serpent` is nicely shown how the serpent script 1.5 can be compiled to the bytecode but also to the sequence of opcodes 1.6 I described earlier[8].

```
def double(x):
    return(x * 2)
```

Code example 1.5: Serpent code example

```
[PUSH1, 67, DUP1, PUSH1, 11, PUSH1, 0, CODECOPY, PUSH1, 78,
JUMP, PUSH29, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, PUSH1, 0, CALLDATALOAD,
DIV, PUSH4, 238, 233, 114, 6, DUP2, EQ, ISZERO, PUSH1, 65, JUMPI,
PUSH1, 4, CALLDATALOAD, PUSH1, 64, MSTORE, PUSH1, 2, PUSH1, 64,
MLOAD, MUL, PUSH1, 96, MSTORE, PUSH1, 32, PUSH1, 96, RETURN,
JUMPDEST, POP, JUMPDEST, PUSH1, 0, RETURN]
```

Code example 1.6: Serpent code example represented by opcodes

1.4.7 Decentralized Applications

Together with Ethereum, the term DApps became more popular. Decentralized Applications, shortly DApps, are applications using smart contracts to decentralize the controlling logic and payment functions of applications. As for application development, almost all aspects of an application can be decentralized. The backend, frontend, data storage and message communications.

Each of these can be somewhat centralized or somewhat decentralized. For example, a frontend can be developed as a web app that runs on a centralized server or as a mobile app that runs on the device. The backend and storage can be on private servers and proprietary databases or a smart contract and P2P storage.

Using decentralization for application have advantages such as availability, transparency and censorship resistance. In the Ethereum ecosystem, as it stands today, there are very few truly decentralized apps. Most still rely on centralized services and servers for some part of their operation.

Here are some applications built on top of blockchain:

- Uniswap: The Uniswap Protocol is an open-source protocol for providing liquidity and trading ERC20 tokens on Ethereum. It eliminates trusted intermediaries and unnecessary forms of rent extraction, allowing for safe, accessible, and efficient exchange activity. The protocol is non-upgradable and designed to be censorship resistant[28].

- Dark Forest: Dark Forest is a universe-traversing, planet-capturing, real-time strategy game. It is open-source, and all interactions within the game are validated by the Gnosis blockchain[29].

- NFT: NFTs are tokens that we can use to represent ownership of unique items. They let users tokenise things like art, collectibles, and even real estate. They can only have one official owner at a time, and they're secured by the Ethereum or other blockchain – no one can modify the record of ownership or copy/paste a new NFT into existence[30].

1.5 Data sources

For my analysis, I need to save all the information in some storage and then perform the analysis. To do all this, I need to get the data from a source. There are several ways.

1.5.1 Commercial API

The first way is to use service API. It has been some time since the rise of cryptocurrencies, so there are sites that track changes in the bitcoin and allow visitors to observe transactions and the flow of the coins. Visitors can easily

find addresses and see where the coins come from and where they were sent afterwards. Sites like `www.blockchain.org`, `www.infura.io` or `www.coinbase.com` provide JSON RPC API that can be used by developers for their projects. The results are already enriched by connections, addresses and etc. Results are in JSON that can be easily parsed.

Using APIs has advantages like no need for running some specialized program, and we can ask only for information that we need. It is fast and easy to use. On the other hand, using APIs also has some disadvantages. Sites that provide their endpoints limit the number of requests per period of time, and if the developer wants to send more requests, he would have to pay to change the limits.

For the purposes of this thesis, I registered on `infura.io` as it was widely recommended on forums regarding the ethereum data. Each user, as part of the free tier, can send up to 100 000 requests per day.

After registering, the portal offers a dashboard where users can review their usage in simple UI, see 1.8. The service generates an HTTPS and web socket endpoint. The URL contains the project id and version.

Exaple: `https://mainnet.infura.io/v3/25394af7a2f543039c16426d2c04c314`

From the project settings, the developer can set limits on the endpoints and choose which methods should be available and which should be restricted. There are two ways how to secure the project, so only authorized applications can use the endpoint. One way is to use the project secret with the project id and send them in each request. The second way is to use JWT (JSON web token)[31].

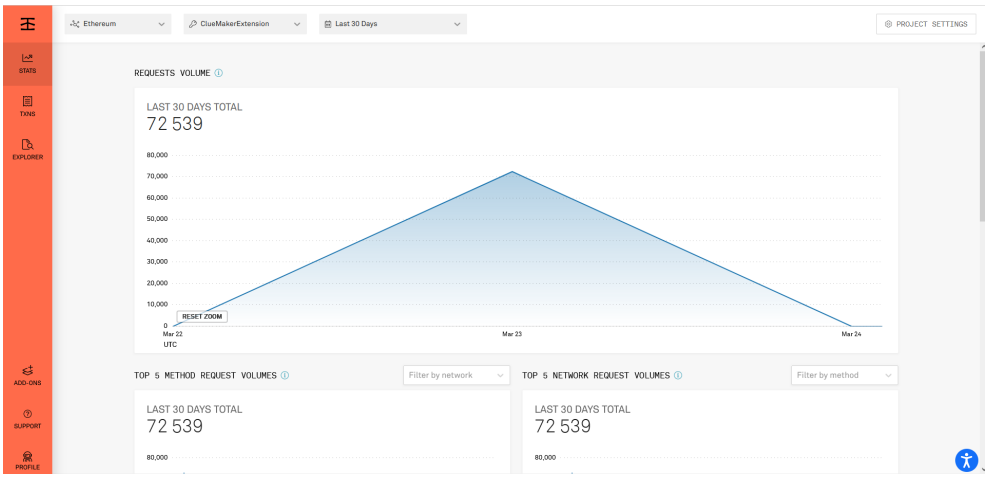

Figure 1.8: Infura.io dashboard[31]

1.5.2 Clients

As said before, the bitcoin blockchain is public, so it means that everyone can watch the chain and items saved inside. It means that users should not be dependent on sites or other third parties. That is why clients exist. There can be multiple implementations for one blockchain, but each has to follow the protocol. The client mostly functions as wallets and blockchain nodes. Users can use them to send and receive transactions but also for mining. The nodes can download the full ledger of the blockchain and take part in the verification process.

I will introduce several clients that can serve as full nodes for Bitcoin and Ethereum networks that are popular, and I will describe their advantages and disadvantages.

1.5.2.1 Bitcoin Core

Bitcoin Core is an official client that implements all aspects of the bitcoin system, including wallets, a transaction verification engine with a full copy of the entire transaction ledger (blockchain), and a full network node in the peer-to-peer bitcoin network. The code is published publicly on `github.com/bitcoin/bitcoin` under the MIT licence. It is implemented mostly in C++ and supports multiple operating systems: Linux, macOS and Windows[4].

Bitcoin core provides a graphical or command-line interface for everyone who wants to work with the bitcoin network, but it also provides JSON RPC API for being controlled remotely if configured.

It can be used for trading, managing wallet and mining.

Using the bitcoin core has disadvantages like the need for downloading the whole bitcoin ledger. The download is made by retrieving data from nodes in the bitcoin network, machines that have full node downloaded and can send us parts of it. And they are not rewarded for sending such data. The ledger is being validated by the bitcoin client, so the result data are correct. As of today, the ledger has a size of 457 GB of data.

The official minimal recommended requirements for running the bitcoin core are not very high. Apart from the disk space of around 350 GB (that I consider not enough as the blockchain has currently around 500 GB), there are also requirements for download and upload. Upload of 5 GB per day and download of 500 MB per day. Described requirements are for running the core and not mining. Mining would require more resources[32].

In this work, I will use the bitcoin core to retrieve blockchain data.

1.5.2.2 Libbitcoin

Libbitcoin is an alternative to the Bitcoin Core. It is a set of C++ libraries for building bitcoin applications. It contains parts of the blockchain divided into libraries that can be installed separately. Such libraries are:

- libbitcoin-system
- libbitcoin-blockchain
- libbitcoin-build
- libbitcoin-client
- libbitcoin-consensus
- libbitcoin-database
- libbitcoin-explorer
- libbitcoin-network
- libbitcoin-node
- libbitcoin-protocol
- libbitcoin-server

Some are dependent on others. It supports operating systems like Linux, macOS and Windows[33].

1.5.2.3 Geth

Geth is the Go language implementation that is actively developed by the Ethereum Foundation, so it is considered the "official" implementation of the Ethereum client. Typically, every Ethereum-based blockchain will have its own Geth implementation[5].

It provides a JSON RPC endpoint that can be used via HTTP, WebSocket or IPC socket. Similarly to Bitcoin Core, it can be used as a wallet, transaction manager and mining software.

The advantage is its wide support as it is the most popular Ethereum client. There is a smaller chance of encountering bugs, and if there are some, the patch will come quickly. Base on the data reported by `ethereum.org`, about 83 % of currently running nodes are using Geth, see chart 1.9.

1.5.2.4 OpenEthereum

OpenEthereum is an implementation of a full-node Ethereum client and DApp browser. It was written "from the ground up" in Rust, a systems programming language, with the aim of building a modular, secure, and scalable Ethereum client. OpenEthereum was developed by Parity Tech, a UK company, and is released under the GPLv3 free software license. It started under the name Parity but was renamed.

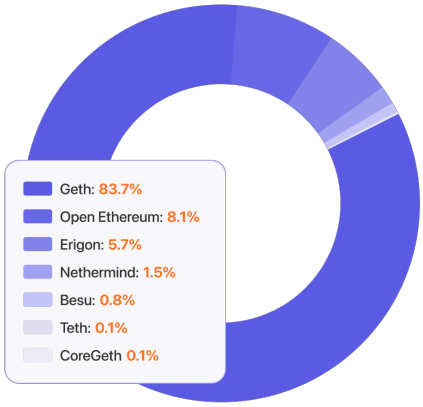

Figure 1.9: Ethereum client diversity chart[34]

Some users are reporting a smaller database footprint, but they face issues more often than Geth users.

From the chart 1.9 can be seen, that around 8 % of running nodes are using OpenEthereum[5].

1.5.3 Public websites

The data that can be retrieved from the clients like Bitcoin Core and Geth is limited to what the clients are storing inside and what they are able to send. But some data are not provided directly but can be derived from it. Or it is not stored in the blockchain ledger at all, and it can be retrieved from the users only. For such reasons, many projects have been created that focus on the data collection of some type and enable the wide public to see that information and use it in their favour.

There are several projects developed by single developers or companies that provide information about addresses, wallets, miners and frauds in the blockchain world. Those sources can be used to extract the data and enrich the already known information from the blockchain clients.

WalletExplorer

A site called WalletExplorer is a place containing Bitcoin wallets and addresses which belong to the wallet. In addition, it also contains the name of the company for some cases. Some are even sorted into categories.

The welcome page divides known wallets into categories: exchange offices, pools, services/others, hazard, old/deprecated. Each section contains a list of wallets with their names. Each wallet has its site with additional information like transactions, amounts, a list of addresses and a link to the service.

Wallet Explorer uses an algorithm based on co-spending. Addresses are merged together if they are co-spent in one transaction. Names of the services were discovered by the developer who registered to known service, made a transaction and saw which wallet bitcoins were merged with or from which wallet it was withdrawn.

The developer also notes that data are updated once per 2 days, but newer names of services are not added since he joined the team at Chain analysis and extended their database.

From the data extraction point of view, the site offers JSON API after contacting the developer. The API is limited and is made to show data on the site. After analyzing the website, it offers CSV data on URL following an easily understandable format. Using this, I am able to construct a web scraping script that accesses the data using CSV files[35].

Bitcoinabuse.com

BitcoinAbuse.com is a public database of bitcoin addresses used by scammers, hackers, and criminals. Anyone can file a report. It is a popular website where anyone can look up a bitcoin address, report a scam address, and monitor addresses reported by others.

The service provides an endpoint where reports can be downloaded in CSV format. As of now, the full report contains almost 400 000 reports. To download the full report, a developer needs to create an account, but the usage of the API is not limited by a pricing plan only to the number of requests per second[36].

OFAC

The Office of Foreign Assets Control ("OFAC") of the US Department of the Treasury administers and enforces economic and trade sanctions based on US foreign policy and national security goals against targeted foreign countries and regimes, terrorists, international narcotics traffickers, those engaged in activities related to the proliferation of weapons of mass destruction, and other threats to the national security, foreign policy or economy of the United States.

OFAC publishes lists of individuals and companies owned or controlled by or acting for or on behalf of targeted countries. It also lists individuals, groups, and entities, such as terrorists and narcotics traffickers designated under programs that are not country-specific[37].

The published list also contains information about the addresses used in several cryptocurrencies like Bitcoin, Ethereum, Lite Coin and others. For this reason, I have decided to extract data from this resource in later parts of this thesis.

1.5.4 JSON-RPC

Both blockchains, Bitcoin Core and Geth, which I work on within this thesis, provide JSON RPC endpoint, so it is suitable to describe it. JSON stands for "JavaScript Object Notation" and is a text-based data exchange format for transferring objects across the network in human-readable form[38]. And RPC is short for "remote procedure call". It is used for building distributed systems as it allows a program on one machine to call a subroutine on another machine without knowing that it is remote. It is similar to calling a function in JavaScript or Python by specifying the function name and its arguments. The JSON-RPC is a combination of the two technologies. It uses the JSON data-interchange format, which is easy for humans to read and write. It is also easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language[39].

The JSON-RPC works by sending a request to the application on the server that has implemented this protocol. The body of the request, serialized as JSON, contains typically three attributes:

- jsonrpc: a string specifying the version of the JSON-RPC protocol
- method: a string containing the name of the method to be invoked
- params: a structured value that holds the parameter values to be used during the invocation of the method
- id: an identifier for the call chosen by the client

The response is JSON object contain four attributes:

- jsonrpc: a string specifying the version of the JSON-RPC protocol
- result: value or object, if call succeded and method returns value
- error: contains object with information about error if issue occurs during the call
- id: same identifier as the value of the id attribute in the request object

As can be seen in code example 1.7 where, using http POST call on address where bitcoin core client is listening, is called method getblock with two parameters.

That is difference from the REST, where instead of method there would be name of the resource and it would be in the URL.

Disadvantage is that it is necessary for the user to know the names of the methods.

```
POST http://127.0.0.1:8332/
Authorization: Basic user pass
Content-Type: text/plain;

{"jsonrpc": "1.0", "id": "curltest", "method": "getblock",
 "params":
   ["00000000a04a30baed00999ad971f807b5e742f602e013519f89eb7248c7ddf5", 2]
}
```

Code example 1.7: Example of JSON RPC call on bitcoind client

1.5.5 IPC

Some blockchain clients allow communication with other applications via the IPC endpoint. IPC, short for Inter-Process Communication, allows threads to communicate and synchronize when they do not share memory. The IPC services allow threads to exchange messages in either asynchronous mode or in Remote Procedure Call (RPC) mode (demand/response mode).

UNIX domain sockets (also IPC sockets) enable efficient communication between processes that are running on the same host/node. UNIX domain sockets support both stream-oriented TCP and datagram-oriented UDP protocols. Unlike internet sockets in the domain where the socket is bound to a unique IP address and port number, a UNIX domain socket is bound to a file path[40].

For example, Ethereum client Geth allows communication via an IPC socket. When Geth starts and configures to use the IPC endpoint, it will create a file with the name geth.IPC. Other processes on the same computer can then use the IPC file to create bi-directional communications with Geth. The IPC is better than RPC in terms of security. If we need only applications on the machine where the Geth is running to communicate with the client, we can use IPC and mitigate the chance of being hacked. There were some cases of users being hacked using the RPC node, as the hacker sent requests to the endpoint that was opened for a short period of time[41].

1.6 ClueMaker

ClueMaker is a visual analytic tool for the visualization of relations, links and flows between subjects. ClueMaker helps enterprises like insurance companies, banks or private companies to combine data from multiple sources and find and investigate ongoing fraud, find connections between entities and detect known patterns.

ClueMaker supports a wide range of data sources like Excel Sheets, PostgreSQL, Oracle Database, MSSQL, Teradata, Apache Hive and more.

ClueMaker is being developed by the company Profinit. Profinit focuses on consulting, software services, product development and outsourcing business. The development started in 2013 and continues till now[42] [43].

The main objective of this work is to extend ClueMaker of tools that would allow users to visualize and analyze cryptocurrencies. The data have to be stored in database storage, transformed and analyzed and connected to the ClueMaker. To provide more than just visualization of the blockchains, part of the analysis is also enrichment of the chain by information that can be helpful with detecting fraud or illegal actions or entities.

The ClueMaker consists of two applications: The ClueMaker and ClueMaker Configurator.

Configurator serves for definition of the workspace that will be used in the ClueMaker app. Here are defined data sources and definitions of entities and relations. The tool also offers to include saved searches and reports.

This tool is meant for administrators who have the information about the data sources and needed access rights. Knowledge of the data structure and schema is also needed, together with knowledge of query language.

ClueMaker Application uses workspaces defined in the Configurator and connects to the data source. Using tools inside, the user can visualize entities and relations. ClueMaker can execute stored reports that will show information in a table or as a diagram.

1.7 Similar projects and solutions

In this section, I introduce products of companies that specialize in crypto markets, and their products can visualize the flows in the blockchains. I will not mention products on sites `blockchain.com/explorer` or `etherscan.io`, as they do not visualize the data and the flows in the graph.

Following products can be found on the web, but the details about the visualizers are not easy to find. The demos are not public, and someone interested in the demo has to contact the company employees and schedule the call.

Chainalysis - Reactor

As part of the analysis, I contacted the person responsible for contact with potential clients at Chainalysis, and I requested a demo. My supervisor for this thesis, who is also from Profinit, and I attended a meeting with Associate Account Executive, who showed us their products Reactor and KYT (Know Your Transaction). The tools they offer are focused on the selection of several blockchains that have the full support and other blockchains that have support only partial. The main advantage of Chainalysis is the gathering of the

resources where the company employs over 100 employees who search the web manually and gather information about wallets, addresses and frauds. The searches are also done on the dark web. Another significant source of data is the company's partners who use the products from Chainalysis and, in return, share their data about blockchain entities. The main clients of Chainalysis are governments, financial institutions, law enforcement and financial regulators.

The Reactor is developed with the focus on investigation and exploring flows in blockchains. It is very similar to what should the result of this thesis be. Users can explore data in the selected blockchain and expand the nodes in the graph. The nodes are enriched for names of services, business segments and ties to known entities. An example can be seen in the image 1.10.

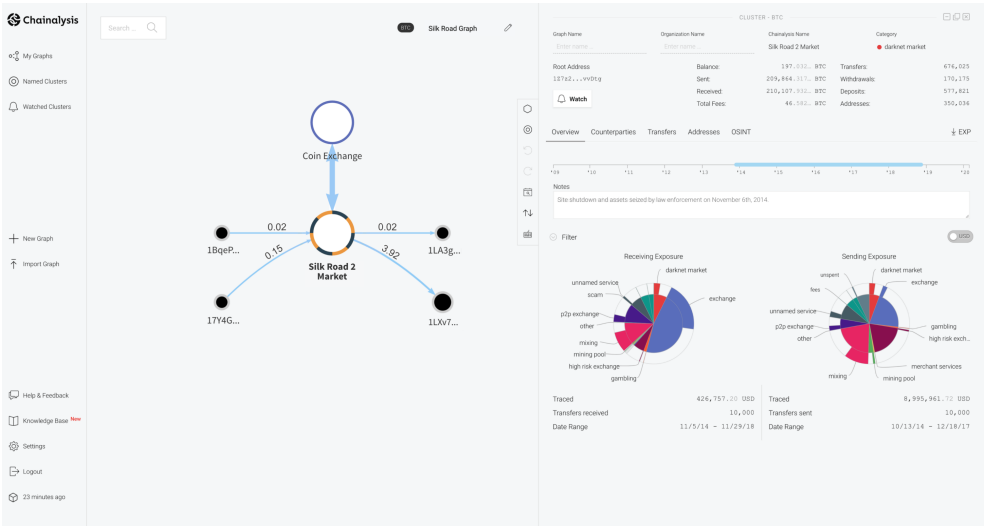

Figure 1.10: Chainalysis - Reactor[44]

Chainalysis offers four more products that I will describe shortly.

Chainalysis Business Data

Product with name **Chainalysis Business Data** offers clients aggregated and visualized data from blockchains so businesses can make decisions based on the data.

KYT

KYT, which stands for Know Your Transaction, enables users to register transactions that will be analyzed in real-time and flagged with tags and security warnings if some issues with senders or receivers are discovered. The user will receive an alert if the risk occurs.

Kryptos

Kryptos is a tool that helps users to navigate the cryptocurrency landscape and find new opportunities by in-depth metrics of data from cryptocurrencies. Users can see trends in the data and understand the behaviour of the industry.

Maket Intel

The last currently offered product is Market Intel, where users can get data and information needed for crypto investments. Daily metrics, activity and real-world applications can help clients to decide and monitor the market.

Chainalysis offers multiple types of licences. Each differs in product and limits. From the call, we learned that the cost of one licence/user/year for Reactor costs in tens of thousands USD[45]. This shows that the analysis of blockchain offers a lot of opportunities.

Coinfirm - AML Platform Vizualizer

Coinfirm was founded in 2016. From the start, it focused on financial services connected to the crypto industry. Offering blockchain analytics and regulatory technology solutions. The company specializes in blockchain AML ('Anti-Money laundering') services and fraud investigations and offers broad blockchain coverage. Coinfirm's solutions are used by many clients, ranging from crypto exchanges such as Binance, and protocols like XRP, to financial institutions and governments[46].

The ALM Platform contains a visualizer that is similar to what I want to achieve in the ClueMaker. See image 1.11.

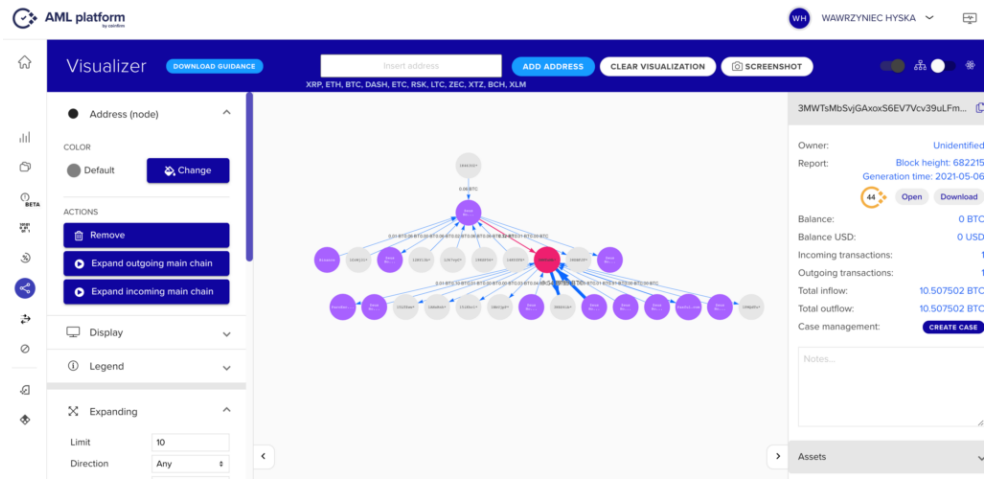

Figure 1.11: Coinfirm - AML Platform Visualizer[46]

Elliptic

Similarly to the Coinfirm, the Elliptic focuses on financial services, providing blockchain analytics for customers from different markets, such as crypto businesses, financial institutions and governments.

An example of Elliptic's visualizer can be seen on image 1.12, which is visualized part of a Twitter scam from 2020.

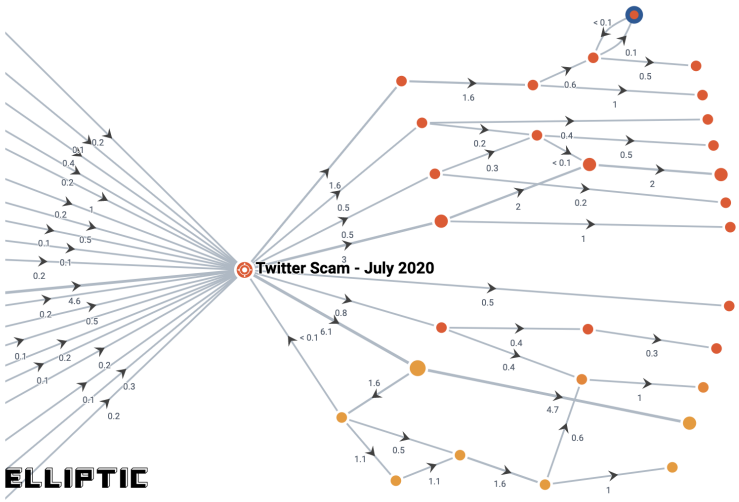

Figure 1.12: Elliptic visualizer[47]

1.8 Comparison of databases

For this thesis, I want to work with blockchain and make an analysis of the data, so I need to store the information about it and its structure. We will be working with blockchains that, at this time, have between 500 GB and 1.2 TB of data in binary form. After saving them, I will enrich them with more information from different sources that I need for the analysis, and the final size will be greater. That leads us to databases.

There are several types of databases. Relational, key-value, wide column or graph databases. Each of them has its use case for which it is developed and optimized.

1.8.1 Relational databases

Relational databases are almost universal as most use cases will work with this type of database. It can store large amounts of data, and it can create relations and then query them. As we will probably need to distribute databases across

multiple machines, and there are some that can scale horizontally. But it is not one of the strong sides of these machines.

1.8.2 Key-value databases

Key-value databases are made for one purpose, and that is storing data under one key and then retrieving those data with the mentioned key. It is possible to create links, but the databases are not optimized for them.

The Riak is a key-value store that allows each of its stored values to be augmented with link metadata. Each link is one-way, pointing from one stored value to another. Riak allows any number of these links to be walked (in Riak terminology), making the model somewhat connected. However, this link walking is powered by map-reduce, which is relatively latent. Unlike a graph database, this linking is suitable only for simple graph-structured programming rather than general graph algorithms[48].

1.8.3 Document databases

Stores data in a uniquely keyed document that can have varying schema and that can contain nested data. Examples include MongoDB and Apache CouchDB.

1.8.4 Graph databases

Graph database management systems (henceforth, a graph database) are online database management systems with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with transactional (OLTP) systems. Accordingly, they are normally optimized for transactional performance and engineered with transactional integrity and operational availability in mind[48].

JanusGraph

One of the graph databases is Janusgraph. Janusgraph database is an open-source, scalable graph database optimized for storing graph structures that contains billions of nodes and edges and making queries on them. It is also transactional, so a set of operations is part of the transaction, and multiple transactions can be active at the same time.

Janusgraph project was created from the Titan project. The development of the Titan project was stopped in 2015. Janusgraph is developed under Linux Foundation and multiple developers from companies like Expero, Google, IBM and Amazon[48].

```
MATCH (a:Person)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
WHERE a.name = 'Jim'
RETURN b, c
```

Code example 1.8: Example of query in Cypher language[48]

Neo4J

Another graph database management system worth mentioning is Neo4J. Neo4j is the most popular among the graph databases. It uses its own storage backend and processing engine. Version for non-commercial use is free to use, but once we want o use it for commercial purposes, the licence is paid. Neo4j uses Cypher language for querying. This language is similar to SQL, and that is why it is so popular. The basic syntax looks like this 2.1.

Thanks to its popularity, there are lots of resources from which developers can draw. For this project, the main obstacle is the licence because I am extending ClueMaker and its commercial product. And for example, clustering is only in the enterprise version. Another difference is consistency, as the Neo4j aims to be strongly consistent when Janusgraph can choose consistency with the chosen storage backend. For example, with HBase, it will be strongly consistent, and with Cassandra, it will be eventually consistent[49].

In the field of graph databases, we can find more projects similar to Janusgraph or Neo4j. Big companies like Amazon and Microsoft have their own solutions for storing graph data.

Azure CosmosDB

Microsoft is offering Azure CosmosDB is multi-model database supporting not only documents, key-value, and column-family data models but also graphs. The storage is horizontally scalable. For communication is used gremlin query language. The engine closely follows Apache TinkerPop specifications but some features supported by Apache TinkerPop are not available. Consistency can be configured on request level[50].

AWS Neputne

Another graph database is AWS Neptune, created and offered by Amazon. Neptune is purely a graph database. Supports gremlin query language and also SPARQL. The storage is immediately consistent. Similarly to the CosmosDB, it is hosted on the cloud, and users do not have to worry about operational overhead[51].

Worth mentioning are also OrintDB, Chronos, IBM Graph or Sqlg.

Chapter 2

Design

In the analytical part of this thesis, I introduced basic concepts of blockchain technology and introduced data sources and databases that could be suitable for this work. In this chapter dedicated to the design, I will use gathered knowledge and describe the architecture of the final product.

2.1 Data

Both clients, Bitcoin Core and Geth, provide data about the block and transactions, but not all information needs to be stored in the database. As the main goal of this thesis is to visualize the relations and movements inside the blockchain, I need to store only a subset of data provided via the client API. For example, information for validation of the block or raw hexadecimal values where decoded equivalent values are available. Those are values I have decided not to store to save the space that is precious resources. And that is why I describe available attributes and if it is going to be stored or not.

2.1.1 Bitcoin client API

Bitcoin client provides REST API endpoint with information about blockchain data as blocks, transactions, and mempool. There are also endpoints for validating the chain, creating blocks or transactions, mining, controlling the client or managing the wallet.

To limit the traffic, I decided to turn off the transaction relay by using parameter *blocksonly* when starting the bitcoin node. Forwarding transactions to peers increase peer-to-peer traffic.

This setting has following effects[52]:

- Fee estimation will no longer work.

```
POST http://127.0.0.1:8332/
Authorization: Basic user pass
Content-Type: text/plain;

{"jsonrpc": "1.0", "id": "curltest", "method": "getblock",
 "params":
   ["00000000a04a30baed00999ad971f807b5e74
   2f602e013519f89eb7248c7ddf5", 2]
}
```

Code example 2.1: Example of POST request with method getblock

- It sets the flag "-walletbroadcast" to be "0", only if it is currently unset. Doing so disables the automatic broadcasting of transactions from the wallet.
- Not relaying other's transactions could hurt your privacy if used while a wallet is loaded or if you use the node to broadcast transactions.
- If a peer has the force relay permission, we will still receive and relay their transactions.
- It makes block propagation slower because compact block relay can only be used when transaction relay is enabled.

By default, the bitcoin client listens on localhost on port 8332. The endpoint listens for a POST request with a body that contains information about the request JSON-RPC version, id, a method that we want to invoke and parameters.

In this thesis, I need to get data inside the blocks and iteratively create an entire chain. For this purpose, I will start with the method for returning information about the block. The method is called "getblock".

getblock

This method can be called using following POST request:

The method accepts two parameters. First required parameter is string representing the hash of the block and second optional parameter is verbosity. Verbosity allows developers to choose how much information they want to receive. There are 3 values:

- 0: for hex-encoded data
- 1: for JSON object containing data about the block and transaction identifiers.

- 2: for JSON object containing information about the transactions

Data model of response on request with verbosity set to 2 can be seen on figure 2.1.

Figure 2.1: Data model response getblock

Response of this call contains all the information that we need to create the blockchain. But some attributes are not so important for us so we need to select which we want to store in the graph.

In this part I want to describe each part of the response and decide if we want to store it.

Root object is containing following attributes:

- hash: Hash of the block. Hash serves us as the unique identifier of the block and is used as a pointer from other blocks in attributes *nextblockhash* and *previousblockhash*. Hash will be used to create the spine of the blockchain that the ledger of connected blocks.

- confirmations: Confirmations represent a number of nodes in the blockchain that confirm the validity of this block. If the returned value is equal to -1, the block is not on the main chain. As we are not verifying the validity of the chain, this information can be used to filter blocks on the main chain, but it is not necessary to store it.

- size: This attribute tells the size of the block in bytes. This information is redundant as there is no need to know the size when we process plain text responses.

- strippedsize: Size in bytes without the witness data that are added after publishing the block. Similarly to the size attribute, this information has no use for us. weight
- weight: Block weight is a measure of the size of a block, measured in weight units. The Bitcoin protocol limits blocks to 4 million weight units, restricting the number of transactions a miner can include in a block.
- height: Block height gives us an index of the block. Starting with the 0 representing the genesis block. This information can be helpful when we want to make sure we process all the data or, for some reason, skip a specific block and continue without knowing the next block hash. AS of now, the bitcoin blockchain consists of more the 720 000 blocks.
- version: Right know there are 4 versions of blocks. With accepting BIP (Bitcoin Improvement Proposal) that is not backward compatible. Version 1 was the version of the genesis block Version with number 2 came in September 2012 and added the height number into the block and changed rules for rejecting the blocks.. Third version came in February 2015 when the bitcoin required strict DER encoding of all ECDSA signatures in new blocks. Last version with number 4 added support for new opcode OP_CHECKLOCKTIMEVERIFY in the script.

  For this attribute I cant find usage in our current architecture as the changed across the versions does not changed the relations or shape of the data. Using the technologies chosen for this thesis, we can add the version number in the future if needed.
- versionHex: Representing the version number in hexadecimal format. This attribute wont be needed as its copy of the previous attribute just in different format.
- merkleroot: Merkleroot is used for creation of the block header and for our analysis is not important. That is why we wont be storing this attribute.
- time: Time of creation of the block in seconds since epoch (Jan 1 1970 GMT). The block time is chosen by miners and have to obey rules but it is possible to have block that is higher but has lower timestamp. For the purpose of connecting the transactions inside the block with real world events, the mediantime attribute is more suitable.
- mediantime: Mediantime is the median time of the past 11 block timestamps, and a block must have a timestamp greater than that median time, so the mediantime attribute always increases within blocks. That is why I choose this timestamp to be stored.

- nonce: Nonce is used purely for mining process and has no purpose for us right now.
- bits: The genesis block's target difficulty equal to "1d00ffff". This parameter is used when mining new block and for our analysis has no purpose.
- difficulty: Difficulty is variable that assures that mining of new blocks will take around 10 minutes even with increasing number of miners and computational power. For our analysis is not important.
- chainwork: Expected number of hashes required to produce the chain up to this block in hexadecimal form. For our analysis, it is unimportant.
- nTx: Number of transactions inside the block. It can be used in the process of transforming and storing the data. It might be useful to make some queries faster as we won't need to recount the transactions from number of relations. I choose to store this information.
- previousblockhash: Hash of the previous block. It's important for creating the relations between blocks.
- nextblockhash: Hash of the next block. In theory, we need nextblockhash or previousblockhash attribute to create the chain. Using the iterating strategy for constructing the chain, I choose to store the hash of the previous block, but a hash of the next block can be used for looking up the next blocks.
- tx: List of transaction objects. Detailed description in next section.

Transaction objects consist of the following attributes:

- txid: Transaction identifier equal to hash of the transaction. This attribute is important and will be stored.
- hash: Hash of the transaction that can differ from txid for segwit transactions. The difference is when the transaction is a segwit transaction, the calculation of hash does not include the witness data, whereas the txid does.
- version: Similarly to the block version, transactions have their own version specifying the rules they follow.
- size: Size of the transaction in bytes.
- vsize: Virtual transaction size is defined as Transaction weight / 4 (rounded up to the next integer). Not important for the purpose of this thesis.

- weight: Similar to block weight is not useful.
- locktime: Time in seconds when the inputs are locked, and cant be used in future transactions.
- hex: The serialized, hex-encoded data for 'txid'.
- vin: Transaction inputs.
- vout: Transaction outputs.

Transaction inputs and outputs are important and are worth describing in a separate section. They are both important parts for observing the flow of funds between addresses and wallets.

Transaction Input, in response called *vin*, consists of following attributes.

- txid: Reference to the other transaction, using the transaction identifier. Will be stored.
- vout: Numeric value used to specify transaction output index, that is being unlocked by this input. Will be stored.
- sequence: The script sequence number. For this thesis it is not important.
- txinwitness: Array of hex-encoded witness data. The data is present for segwit transactions and will be stored.
- scriptSig: JSON object with two attributes, *asm* and *hex*, that is used for unlocking the referenced output. Value in attribute *asm* is decoded value of the second attribute *hex*. Contains the unlocking script, and that is why it will be stored, but only the *asm* value, as it is in human-readable form.

Transaction Output, in response called *vout*, contains the value that is locked and the script that can be unlocked by other transactions with the correct unlocking script.

- value: The value representing a number of coins locked in this output. Will be stored.
- n: Index of the output inside the transaction. It is used to be able to be referenced by other transaction input.
- scriptPubKey: Object with locking script and related information. This object contains five more attributes:
  - asm: Contains decoded value of attribute *hex*. This value will be stored.

- hex: Hexadecimal value of the locking script. It will not be stored.
- reqSigs: Number of required signatures. This value is not needed.
- type: Type of the locking script is one of the standard types. Will be stored.
- addresses: List of addresses if known. Will be stored.

This is the main resource of information for the Bitcoin blockchain.

But there are other methods that can be called on the bitcoin client and retrieve new information or subset of information. I will introduce other methods, but I will not go into such detail as in the *getblock* method.

getblockchaininfo

This method return information about the state of the client. The response contains information about the current network name (main, test, regtest), number of fully validated blocks, pruning information, status of the soft forks and possible warnings. It can be used to monitor the client.

getblockhash

Returns hash of a block at provided height. The method getblockhash takes as a parameter hash of the block, and if only a height is known, this method can be used to look up the hash.

getrawtransaction

This method return detailed information about the transaction specified by the txid attribute. To make this method work, the client has to be set up in a specific way. If it isn't, it will work for transactions in the memory pool.

This method accepts three attributes: txid, verbose and blockhash. The parameter txid takes transaction identifier of wanted transaction. Verbose is a boolean values which sets the level of detail returned in the transaction. The blockhash refers to the block in which to look for the transaction.

This method can be used for the retrieval of transaction information without the need to request data in the block in which the transaction is included.

2.1.2 Geth client API

Geth client that I will be using in this work offers JSON RPC endpoint with methods that can be called. The methods are divided into five categories: web3, net, eth, db and shh.

For this thesis, I will use only methods from the eth category that serves for retrieving data about the Ethereum blockchain.

eth_getBlockByNumber

This method returns information about the block with a specified block number. The method takes two parameters: block number and verbosity. The block number is an integer representing the height of the block that should be returned. The verbosity is a boolean value that, if equal to true, the transaction objects will be included in the response. Compared to the Bitcoin block structure, Ethereum Block has a simpler structure.

The response object contains the following attributes (with enabled verbosity):

- number: The height of the block. Will be stored.
- hash: The 32 byte hash of the block. Will be stored.
- parentHash: Hash of the previous block. Will be stored.
- nonce: 8 byte value used for the proof-of-work. Is used for verification and will not be stored.
- sha3Uncles: Hash of the uncle block data. Uncle blocks does not change the state of the blockchain. Will not be stored.
- logsBloom: The 256 byte value used for filtering hashes of objects. Will not be stored.
- transactionRoot: The root of the transaction trie of the block. Will not be stored.
- stateRoot: The root of the final state trie of the block. Will not be stored.
- receiptsRoot: The root of the receipts trie of the block. Will not be stored.
- miner: Address of the miner. Will be stored.
- difficulty: Integer value representing the difficulty for this block. Will not be stored.
- totalDifficulty: The total difficulty is the accumulated sum of all blocks difficulty until the this block. Will not be stored.
- extraData: Extra data added by the miner. Will be stored as it can be analyzed in the future.
- size: The size of this block in bytes. Will not be stored.
- gasLimit: The maximum gas allowed in this block. As it does not have effect on the final fee, will not be stored.

- gasUsed: The sum of used gas by all transactions in this block.
- timestamp: The unix timestamp for when the block was collated. Will be stored.
- transactions: Array of transaction objects.
- uncles: Array of uncle block hashes. Will not be stored.
- baseFeePerGas: Base fee per unit of gas, which is added with EIP-1559.

The transactions attribute contain list of transaction objects. The object has following attributes.

- blockHash: The hash of the block where this transaction was included. Will be stored.
- blockNumber: The block number where this transaction was included. Will not be stored.
- from: Address of the sender. Will be stored.
- to: Address of the receiver. Will be stored.
- gas: Amount of gas provided by the sender. Will be stored.
- gasPrice: Gas price specified by the sender in wei. Will be stored.
- hash: Hash of the transaction. Will be stored.
- input: The data send along with the transaction. Will be stored.
- nonce: The number of transactions made by the sender prior to this one. Will not be stored.
- chainId: An identifier of the chain that serves as protection against replaying of the transaction between chains. Will not be stored.
- transactionIndex: The index of the transactions position in the block. In Ethereum it is not used for referencing the input or output. Will not be stored.
- value: Value transferred in wei. Will be stored.
- v: ECDSA recovery id. Will not be stored.
- r: ECDSA signature r. Will not be stored.
- s: ECDSA signature s. Will not be stored.

- maxFeePerGas: Sum of baseFeePerGas + maxPriorityFeePerGas. Will be stored.
- maxPriorityFeePerGas: Serve as a tip to the miner. Will be stored.
- accessList: List of addresses and storage keys that the transaction plans to access. Will not be stored.

Data model of response with verbosity set to true can be seen on figure 2.2.

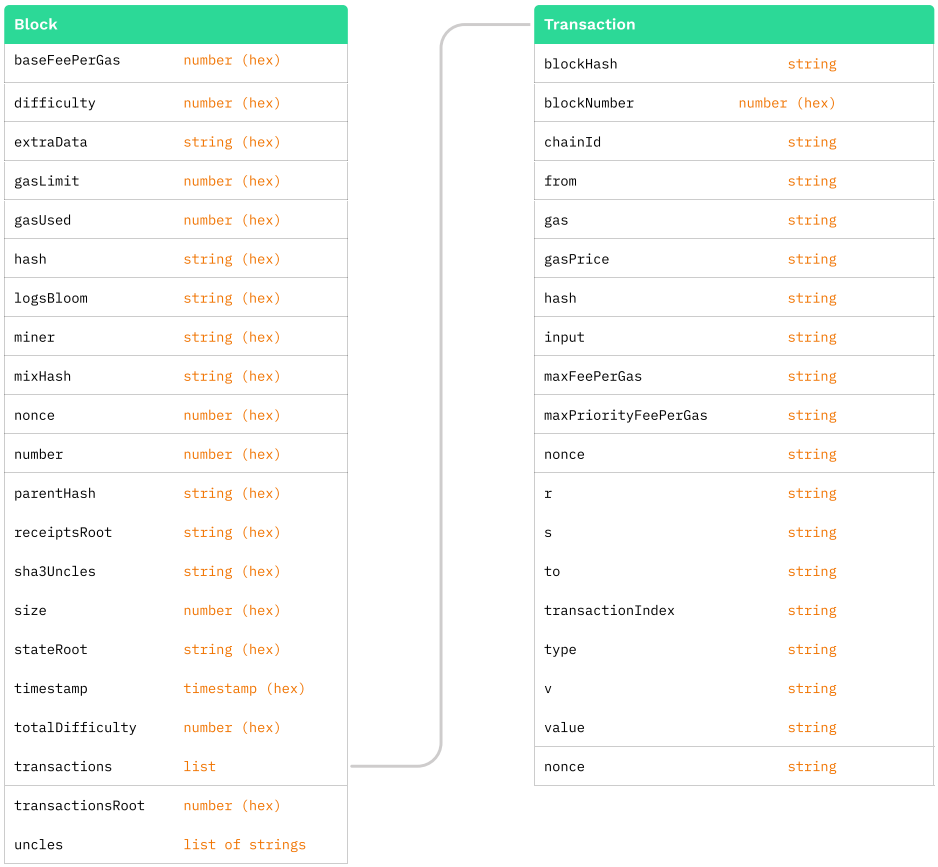

Figure 2.2: Data model response eth_getBlockByNumber

2.2 Database schema

In the previous section, I have selected attributes that will be stored. The structure in which they are returned from the API is not necessarily the most

suitable for storing. The goal is to keep the model simple and compact. For this reason, I will have created a schema that can be seen on the image 2.3 for Bitcoin data and image 2.4 for Ethereum data. In the following sections, I will describe each node and the relation between them.

For the blockchain data, I will be using the graph database JanusGraph, and that is why, when describing the model, I will be talking about the nodes and edges. The labels of the nodes follow the name convention where the first three characters represent the blockchain and then follow the name of the object.

2.2.1 Nodes

Nodes in the graph database represent the data structures of the blockchain. They contain most data from the blockchain. Some attributes do not hold information for possible investigation but have structural importance, or they are used for better performance. For each node, I will describe attributes that are specific or there are some changes from the data clients provide which were discussed in the previous section.

Each node in the schema has a label equal to its name. Schema for the nodes for Bitcoin or Ethereum network can be seen on images 2.3 and 2.4

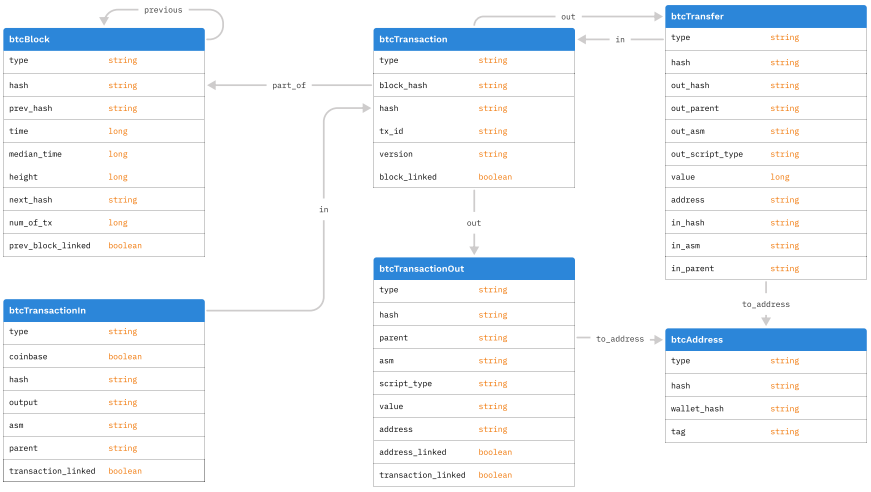

Figure 2.3: Schema of nodes for Bitcoin network

btcBlock

Node with label *btcBlock* represents the Bitcoin block structure. It serves as a structure that will be used for marking imported blocks. If the block exists

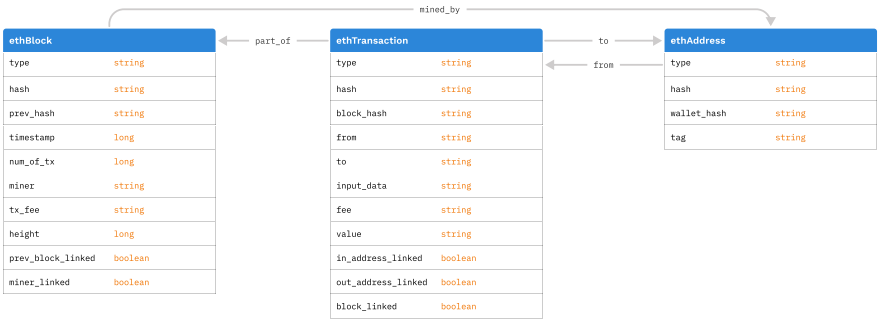

Figure 2.4: Schema of nodes for Ethereum network

in the database, the transactions and their inputs and outputs are also saved in the database. In case of need to start the import process, where only part of the chain is imported from the stored blocks, the application will recognize where it left off and where it should start.

- type: The attribute that has the same value as the label: *btcBlock*. Is used for indexing.
- hash, prev_hash, next_hash: The hashes used for identifying the blocks.
- time, median_time: Time values are represented as a numeric values.
- num_of_tx: Number of transactions in the block. It can be used for validation.
- prev_block_linked: Boolean value used for better performance when creating edges between blocks. Nodes with the value *true* have a link to the previous block.

btcTransaction

Node with label *btcTransaction* represents the transaction object. The transaction belongs to one block and has one or more inputs and one or more outputs. The transactions are essential in the schema as they cant be omitted.

- type: The attribute type has the same value as the label: *btcTransaction*. Is used for indexing.
- hash, txid, block_hash, version: Data to identify the transaction, the block it belongs to and transaction version.

- block_linked: Boolean value used for better query performance.

btcTransactionIn

Node to represent transaction input. Has label with value *btcTransactionIn*. It belongs to one transaction and can reference one or zero transaction outputs.

- type: Type of the node with value *btcTransactionIn* is used for Indexing.
- coinbase: The boolean value marking the transaction input created by the miner. If the transaction input has coinbase attribute to true, it does not contain attributes like output or asm.
- hash: The transaction input object does not have hash on its own, but I construct it to be used as unique identifier. It is constructed from prefix "in_", then the *tx_id* attribute of parent transaction and the index of the input inside the transaction.
- output: Attribute constructed from the attributes vout and txid of the transaction input object. It reference the transaction output the input is unlocking, as it is equal to the output *hash* attribute. Is created from prefix "out_" with concatenated *txid* and *vout* attributes.
- asm: Corresponds to the *asm* attribute of transaction object.
- parent: The *tx_id* of the parent transaction.
- transaction_linked: Mark if the edge to the transaction node have already been created. Used for faster queries.

btcTransactionOut

Node with label *btcTransactionOut* represents the transaction output. It belong to one transaction, can reference the address and on its own does not reference any inputs. But can be referenced by an input.

- type: Equal to string with value *btcTransactionOut*.
- hash: The hash is constructed from the "out_" prefix followed by the *tx_id* of parent transaction and the *n* attribute of transaction output object. Is referenced then by this attribute by the input.
- parent: The *tx_id* attribute of the transaction node it belongs to.
- asm, script_type and address: Corresponds to the object attributes.
- value: Value in satoshi units represented as string.
- address_linked, transaction_linked: Boolean values used for better performance.

btcTransfer

The node with label *btcTransfer* does not represent any structure retrieved from the Bitcoin client, but I have created it to represent output and input that is spending it. Without this node, the model would contain a lot of inputs and outputs and traversals from one transaction to other would go through 2 node using 3 edges. Therefore I designed a node that is combination of both nodes. It does reference the parent transaction of the input and output, and also references the address.

- type: String value equal to *btcTransfer* for every transfer node.
- hash: Hash of the transfer node is created from the output *hash* value, only the prefix "out_" is replaced for "tr_".
- out_hash, out_parent, out_asm, out_script_type, value, address: These attributes correspond to the attributes in transaction output node. As the attributes address and value are essential in in exploring the network, I used names without the prefix.
- in_hash, in_asm, in_parent: Values from the input node.

The merging of the input and output is depicted on image 2.5.

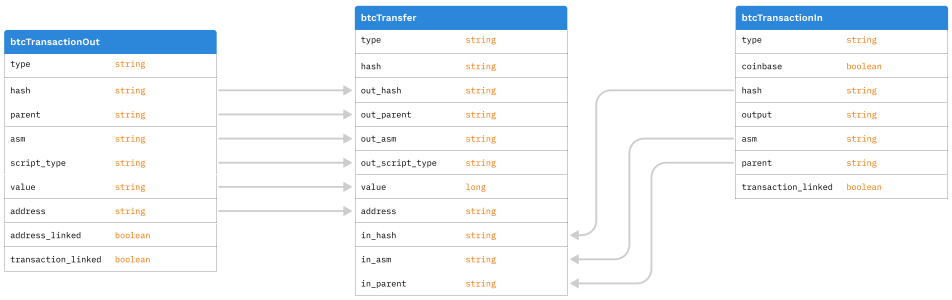

Figure 2.5: Merge of the btcTransactionOutput and btcTranasctionInput

btcAddress

Node with the label *btcAddress* serves as representation of the address. The combination of attributes *hash* and *type* are unique.

- type: String that is for all address nodes equal to *btcAddress*.
- hash: Hash of the address.

- wallet_hash: Hash of the wallet, it belongs to.
- tag: String containing information about the address.

ethBlock

The ethereum has only three nodes and first one is the node with label *ethBlock*. It serves as the maker for synchronization. If the node exists, the transactions inside should be imported also and there is no need to add them.

- type: String value equal to *ethBlock*.
- prev_hash, height, hash: These attributes correspond to ethereum block object attributes.
- timestamp: Timestamp of block creation represented as the numeric value.
- num_of_tx: Number of transactions included into this block. Used for validation.
- miner: Address of the miner. References *btcAddress* node.
- tx_fee: Sum of fees from transaction that is used as reward for the miner.
- prev_block_linked, miner_linked: Boolean values used for better query performance.
- wallet_hash: Hash of the wallet, it belongs to.
- tag: String containing information about the address.

ethTransaction

Node with label *ethTransaction* representing the transaction in Ethereum network.

- type: String equal to *ethTransaction*.
- hash: Hash of the transaction. Used for as an identifier.
- block_hash: Block hash referencing the *btcBlock*.
- from, to: Addresses referencing the *btcAddress* nodes.
- input_data: Input data as a string.
- fee: Fee calculated from the gas price and used gas. Represented as string.

- value: The value of transferred coins in wei units represented as string.
- prev_block_linked, miner_linked: Boolean values used for better query performance.

ethAddress

Node representing the address in Ethereum network. Each node has label *ethAddress*. It it separated from the *btcAddress* as in Ethereum the addresses can be divided into two types and in future it would be necessary to split them.

- type: Each to nodes label *ethAddress*. Used for better performance.
- hash: Hash of the address. Is used a unique identifier.
- wallet_hash: Hash of the wallet, it belongs to.
- tag: String containing information about the address.

wallet

Nodes with label *wallet* represent collection of addresses belonging to one owner.

- type: String value equal to *wallet*
- hash: Hash of the wallet, create from the name by hashing with SHA256.
- name: Name of the owner of the wallet. Can be name of an individual or the company.
- segment: Segment of the market in which the owner of the wallet is doing business.
- source: Name of the source from which it was retrieved.

Wallets are not specific for the blockchain and that is why the name does not have prefix ”btc” or ”eth”.

2.2.2 Edges

Edges are one of the unique properties of the graph database and allow users to traverse the nodes in the graph through the edges. The edges in schema for this thesis represent references from the one node to the other. In following part I will describe edges between nodes and their properties.

- previous: The edge with label *previous* is created between two *btcBlock* or *ethBlock* nodes. It is created based on the *prev_hash* and *hash* attributes. It is directed edge and each block node can have two such edges connected to it. One incoming and one outgoing. Except for the last blocks, all other should be connected into the chain.
- part_of: The edge with the label *part_of* is created between blocks and transactions. In Bitcoin between *btcBlock* and *btcTransaction*. In Ethereum between *ethBlock* and *ethTransaction*. Always directed from the transaction to the block. Each transaction can have only one outgoing edge. Each block can have one or more incoming edges with this label.
- in: The edges with label *in* serves in each blockchain as connection between different structures. In schema for Bitcoin node it serves as relation from *btcTransactionIn* or *btcTransfer* to the *btcTransaction*. In the schema for Ethereum nodes it from *ethAddress* node to the *ethTransaction* node. The edge is directed.
- out: The edge with the label *out* is used for two reasons. Fist is to connect *btcTransaction* nodes with their outputs nodes with label *btcTransactionOut* or already merged *btcTransfer* nodes. Transactions should have at least one outgoing edge and *btcTransactionOut* and *btcTransfer* should have exactly one incoming edge.
- to_address: Edge that connects *btcTransactionOut* and *btcTransfer* nodes with the *btcAddress* nodes has label *to_address*. One address node can have multiple incoming edges of this label and transaction output and transfer nodes can have one or none outgoing edges.
- mined_by: The edge leading from the *ethBlock* to the *ethAddress*, representing the relation between the address of the miner and the block.
- belongs_to: Nodes representing addresses can belong into the wallet. This edge is leading from the *btcAddress* or *ethAddress* into the *wallet* node.

2.2.3 Indexes

As the database will contain billions of nodes, the queries need to be as fast as possible. This can be done partially by the use of the indexes. In the application, I use queries that filter based on an attribute and these indexes should ensure that the database system does not have to iterate over all nodes in the database to find the wanted result.

- byTypeAndHashUnique: This is the most used index by the queries in this thesis. As the Janusgraph database system can not create indexes

on the *label* of the node, I have added attribute *type* with the same value, which can be used in the indexes. Also, the hashes of the object in the Ethereum and Bitcoin serve as the identifiers and when searching for one specific node, the most queries will filter the results based on the *type* and *hash* attributes. The index is marked as unique, which also ensures their wont be any duplicate values. All nodes are indexed, and this is a composite index.

- byType: Some queries are not looking for one specific node but for nodes with a specific type. For such queries, I have created this composite index.

- btcBlockHeight: The application I will describe in the implementation part of this thesis is made to be able to start once again after the termination. This is done by being able to find the last block created. For such queries I have added this composite index on the attributes *type* and *height* where only the *btcBlock* nodes are indexed. The index is unique, as only one block in the network can have a specific height.

- ethBlockHeight: Similar index, as for the *btcBlock*, was created for the Ethereum blocks. As Ethereum creates blocks faster than the Bitcoin network, this index is even more important. The index and is build on attributes *type* and *height* where only the *ethBlock* nodes are indexed.

- btcTransactionOutAddressLinked: The following indexes are created specifically for queries that are used when creating the edges between the nodes. In this type of query, the application is searching for nodes that have not been connected by a specific edge. As I encountered an issue with traversals filtering the nodes based on the existence of the specific edge, I created for each node attribute that do exactly this. Using the index on this attribute allows me to construct traversals that will return nodes with specific *type* and with or without the edge faster. For this specific index it is type with value *btcTransactionOut* and attribute *address_linked*.

- btcTransactionInTransactionLinked: Composite index for better performance of traversals searching for transaction inputs that have not been connected by the edge to the transaction. This index is build on the attributes *type* and *transaction_linked* where only the *btcTransactionIn* nodes are indexed.

- btcTransactionBlockLinked: Composite index for better performance of traversals on *btcTransaction* nodes with use of the attribute *block_linked*. This index is build on the attributes *type* and *block_linked* where only the *btcTransaction* nodes are indexed.

- ethTransactionInAddressLinked: Similar issues, as with the performance of traversals on nodes for the Bitcoin, have occurred for the Ethereum nodes. Therefore I created three indexes for traversals used frequently in the application. This composite index is build on *type* and *in_address_linked* attributes of the *ethTransaction* vertexes.

- ethTransactionOutAddressLinked: Similarly to previously described indexes this composite index is used for better performance of traversals making use of *type* and *out_address_linked* on *ethTransaction* nodes.

- ethBlockMinerLinked: Last index I use in the Janusgraph database is build on attributes *type* and *miner_linked* with only the *ethBlock* nodes being indexed.

In this thesis, I do not use indexing backends like Elasticsearch, Apache Solr and Apache Lucene. In future, it would be suitable to configure such backend and use it for faster traversals. Plus, it allows the use of mixed indexes that can provide support for geo, numeric range, and full-text search.

2.2.4 Relational Database schema

The main application is responsible for the import of data from the Ethereum network and Bitcoin network into the Janus graph database. As described in the analytical part of this thesis, there are other sources of information that can be used to retrieve additional information that can be used for future analysis. For storing the extracted data, I have decided to use the PostgreSQL database, and in this part of the thesis, I will describe the schema and use of the items in the tables on image 2.6.

data_source

Table with name *data_source* contains records of data_sources for which the process of extraction has begun or has been finished. Also, serves as the record for finding out why some addresses have been tagged. The table contains rows for basic identification.

- data_source_id: The primary key for this table, represented by auto incremented integer.

- name: Varchar representing the name of the resource.

- date_updated: Date, when the resource was updated for the last time.

- url: Url of the the resource. Should not be the url to the file of API, but the index page where more information about the resource can be found.

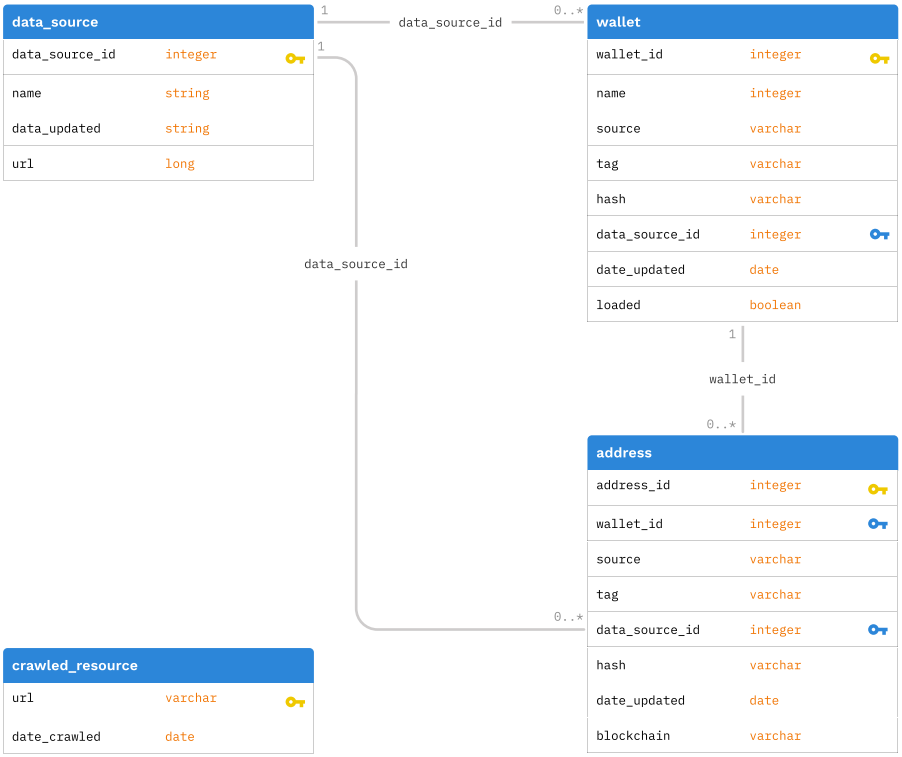

Figure 2.6: Database schema for PostgreSQL database

crawled_resource

The second table in the database is *crawled_resource*. This table has only two columns with name *url* that is also primary key and date_crewled with data it was crawled last time. This table serves to record specific crawled resources. This is specifically for the resources that serves the results in multiple files or on multiple pages, where the url is different from others just in few parameters.

For example the pages below differs only in the page parameter in the url. Contents are then used, when starting to extract new resource, to check if it has been already crawled.

- `walletexplorer.com/wallet/Huobi.com/addresses?page=1`
- `walletexplorer.com/wallet/Huobi.com/addresses?page=2`

wallet

Table with name *wallet* serves for the purpose of saving information about the potential wallet from the crawled resources. The software or hardware wallet can't be identified, but in this work I use the term wallet for describing the owner. So addresses owned by one institution or person are grouped together into one wallet. The columns are following.

- wallet_id: The primary key of the table represented as auto incremented integer.
- hash: The wallets does have hash on its own, but for us in the Janusgraph database where I build on nodes with hash attribute, it is handy to create hash for the wallets also. The hash is created from the wallet name attribute by hashing it with SHA256 algorithm.
- name: The name of the wallet. Can be any identification of the owner, for example name of the company as `Huobi.com` or name "Matej Adamec".
- source: Name of the source. This violates the Third Normal Form in the relational databases but allows for faster retrieval of information about the wallet.
- segment: The segment should describe the part of the market in which the owner operates. This attribute can also indicate if the wallet is stolen, used for criminal activity or is on the sanction list.
- data_source_id: The foreign key of the data source in which the wallet was found first.
- date_updated: The date the record was last updated.

address

- address_id: The primary key of the address record. It is auto incremented integer.
- wallet_id: Foreign key of the wallet record it belongs to.
- tags: Tags used for quick and short marking of the address. Can be used for marking as stolen, theft, sanction. The values are separated by the semicolon.
- data_source_id: The foreign key of the data source record as addressed in one wallet can come from multiple data sources.
- hash: Hash of the address.
- date_updated: The date the record was last updated.

- blockchain: The short name of the blockchain the address belongs to. As I am working with Bitcoin and Ethereum, the actual options are BTC or ETH.

Chapter 3

Implementation

In this chapter I will describe the implementation of the solution. I will go through the architecture I have chosen and the configuration of each component. I will describe how the process of importing data into database works. Also, I will go through the implementation of the connector for the ClueMaker and how the data can be displayed and explored.

The final implementation consists of several systems as can be seen on the image 3.1. Each part in this architecture has its role and communicate with other applications.

3.1 Janusgraph

I described the JanusGraph database shortly in the analytical part of the thesis 1.8.4. Here I want to go into bigger detail and describe features that are different from standard relational debases. Janusgraph is constructed to support the processing of large graphs that require storage and computational capacities beyond what a single machine can provide. The developers state that the system is built with scalability in mind. Scaling graph data processing for real-time traversals and analytical queries is JanusGraph's foundational benefit. The power of the database to process a large number of concurrent transactions scales with the number of machines in the cluster.

3.1.1 Storage backend

One of the features I did not encounter in traditional databases is a pluggable data storage layer. The storage backend is software components that tell JanusGraph how to talk to its data store. The database does contain the ability to store the data in the in-memory storage backend, but it is not suitable for a larger amount of data. However, this allows the architect to choose a set of features (performance, scalability, ease of maintenance, cost) that the backend provides.

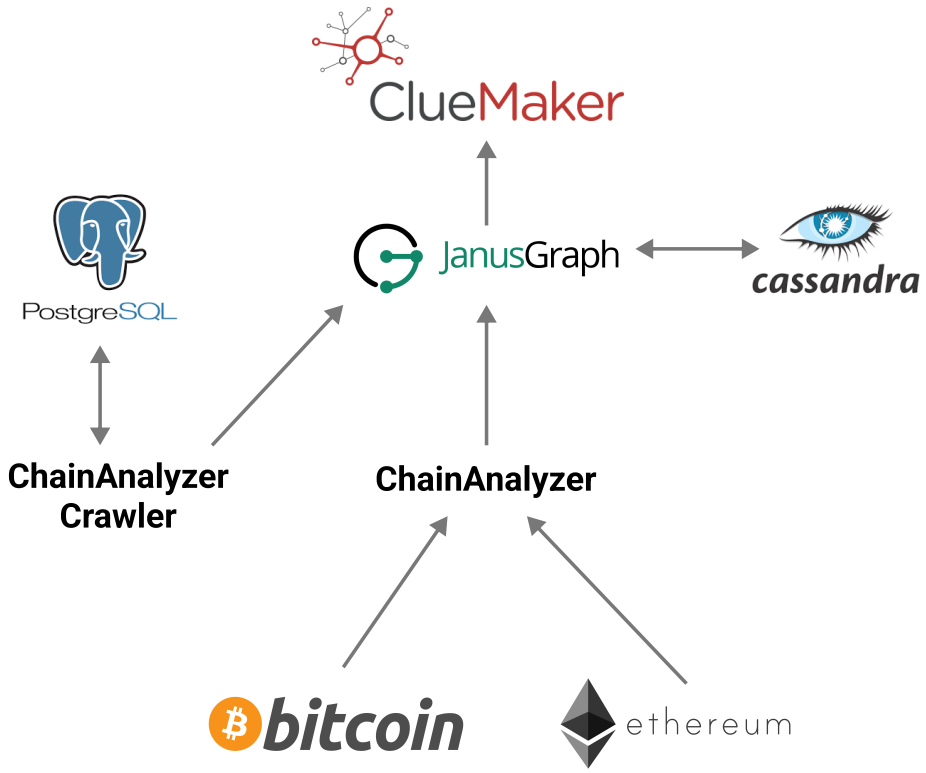

Figure 3.1: Architecture schema

The official documentation mentions the following storage backend that is supported by the JanusGraph[53].

- Apache Cassandra: The most popular choice based on a number of sources and discussions on the internet. It provides great performance, scalability, eventual consistency, fault-tolerant and high availability without compromising the performance[53].
- Apache HBase: It is an open-source, distributed, versioned, non-relational database modelled after Google's Bigtable leveraging the distributed data storage of Hadoop and HDFS[53].
- Google Cloud Bigtable: It is Google's NoSQL Big Data database service offering consistent low latency and high throughput. It is strongly consistent, and the transfer of data is done through the internet connection[53].
- Oracle BerkeleyDB: Is an open-source, embeddable, transactional storage engine written entirely in Java. Supports very high performance and

concurrency for both read-intensive and write-intensive workloads. The licence is paid for commercial use[53].

- ScyllaDB: It is not officially supported by the JanusGraph as it is not included in the documentation, but from the ScyllaDB documentation is evident, that it can be used with JanusGraph. The project comes from Apache Cassandra. The database is NoSQL and provides high availability with great performance[53].

3.1.2 Indexing backend

Another feature of the JanusGraph database is a pluggable indexing backend. There are two types of indexes. Composite and mixed indexes. The composite indexes are supported natively through the primary storage backend. However, the mixed indexes that are more powerful require the configuration of the indexing backend.

Again, it is up to the architect to choose the backend based on the features that are supported, as well as the performance and scalability of the index. Currently, JanusGraph supports the following index backends[54].

- Elasticsearch: Elasticsearch is a distributed, RESTful search and analytics engine. It has good support for JanusGraph distributed across multiple machines.
- Apache Solr: Solr is highly reliable, scalable and fault-tolerant. Providing support for distributed indexing, replication and load-balanced querying.
- Apache Lucene: Performs better in small scale, single-machine applications. Suitable for unit tests.

In this work, I do not use the indexing backend, but I consider it as one of the main ways how to improve the final solution.

3.1.3 Schema

The JanusGraph's graph is comprised of vertexes and edges between them. Both can have labels and property keys. By defualt, the schema is implicit and created with the insertion of new data. It can also be explicit by definition of the schema in graph management.

The vertexes or nodes as I refer to them in this thesis, are representation of objects or records. They have properties with values. JanusGraph natively supports common types like String, Character, Boolean, Integer and more. The properties can have multiple types of value cardinality associated with the key on any given vertex. Supported are SINGLE, which is a default, LIST and SET.

They can be connected by the edges. Similarly to the relation in relational databases, the edge can have multiplicity. The multiplicity of an edge with specific label defines a multiplicity constraint on all edges of this label, that is, a maximum number of edges between pairs of vertices. Janusgraph distinguishes between several types of edge multiplicity: MULTI, which is a default, SIMPLE, MANY2ONE, ONE2MANY, ONE2ONE.

The schema can also contain definition of the indexes. As mentioned before in the description of indexing backends, the JanusGraph supports two types of indexes.

Composite indexes are stored in the storage backend. Composite indexes retrieve vertices or edges by one or a (fixed) composition of multiple keys. That composite graph indexes can only be used for equality constraints and do not allow full-text searches or indexes on numeric ranges. The composite indexes allow adding constraints on the uniqueness of the value in the graph.

Mixed indexes are the second type and enable the database to retrieve vertices or edges by any combination of previously added property keys. They provide more flexibility than composite indexes and support additional condition predicates beyond equality. On the other hand, mixed indexes are slower for most equality queries than composite indexes[54].

3.1.4 Gremlin

The gremlin query language is used for executing traversal and modifications in the graph databases. It is developed as part of the Apache TinkerPop project and is path oriented functional language which allows the construction of paths by concatenation of the operators for traversal. Gremlin is not dependent on the JanusGraph database as it is used by other graph databases. It offers abstraction above the graph databases and allows to easily switch to different graph database system without big changes in the implementation[55].

In the code example 3.1 can be seen how the traversal can be constructed.

```
g.V().has('name', 'hercules').out('father')
 .out('father').values('name')
```

Code example 3.1: Gremlin traversal example

The character `g` at the start of the query represents current graph traversal. The function `.V()` selects all vertexes. By the function `.has('name', 'hercules')`is the selection limited on vertexes with property `name` equal to `hercules`.

From selected nodes is then sent gremlin, that traverse two times outgoing edges with label `father`. And from the vertexes where the gremlins are currently positioned is selected property with the key `name`.

More function and operators will be described later in through the implementation.

3.1.5 Configuration

In this thesis I use the version 0.6.0 that is compatible with chosen Apache Cassandra release.

When the Janusgraph is started, it starts also the gremlin server, that is listening for the traversals and communicate with the Janusgraph database. In its configurations is several properties that needs to be changed for correct behaviour.

Apart of basic configuration of the host and port where the gremlin server will be listening, the configuration also contains following properties I have chosen to change.

- evaluationTimeout: The amount of time in milliseconds before a request evaluation and iteration of result times out. By default set to 30 seconds but for our queries is, the default problematic. Therefore I have decided to set it on a value of 30000, which is equal to 5 minutes. Even as I do several steps to make queries faster, some traversals have to travel across millions of nodes.

- serializers: A List of Map settings, where each Map represents a MessageSerializer implementation to use along with its configuration. This defines the message format in which the gremlin server will communicate with other applications. For the purposes of this thesis, I allow the use of GraphBinaryMessageSerializerV1, GryoMessageSerializerV3d0 and GraphSONMessageSerializerV3d0 serializers.

- storage.batch-loading: Enabling the storage.batch-loading configuration option will have the biggest positive impact on bulk loading times for the applications. Enabling batch loading disables JanusGraph internal consistency checks in a number of places. Most importantly, it disables locking. In other words, JanusGraph assumes that the data to be loaded into JanusGraph is consistent with the graph and hence disables its own checks in the interest of performance.

- schema.default: If set to none, the automatic type creation is disabled and only the types and attributes inside the schema are allowed. This is important due to the batch loading.

- ids.block-size: Each newly added vertex or edge is assigned a unique id. JanusGraph's id pool manager acquires ids in blocks for a particular JanusGraph instance. The id block acquisition process is expensive because it needs to guarantee a globally unique assignment of blocks. Increasing ids.block-size reduces the number of acquisitions but potentially leaves many ids unassigned and hence wasted. Therefore I set the value to 1000000.

- storage.buffer-size: JanusGraph buffers write and executed in small batches to reduce the number of requests against the storage backend. The size of these batches is controlled by storage.buffer-size. When executing a lot of writes in a short period of time, it is possible that the storage backend can become overloaded with write requests. In that case, increasing storage.buffer-size can avoid failure by increasing the number of writes per request and thereby lowering the number of requests. Based on the experiments I set the value to 2048.

- ids.authority.wait-time: This property configures the time in milliseconds the id pool manager waits for an id block application to be acknowledged by the storage backend. The shorter this time, the more likely it is that an application will fail on a congested storage cluster. Therefore I set the value to 1000.

After proper configuration, the schema described in the chapter dedicated to the design can be imported and the application can start to work with the database[56].

3.2 Cassandra

Chosen storage backend for the Janusgraph database is Cassandra. It is installed on the same server as Janusgraph but can be extended and used as a distributed cluster. The Cassandra database works correctly when it is started and no special configuration is needed to make it work. The data inside are fully managed by the Janusgraph database.

However, during the implementation, I encountered several issues caused by the tombstone creation. Cassandra uses a log-structured storage engine. Because of this, deletes do not remove the rows and columns immediately and in-place. Instead, Cassandra writes a special marker, called a tombstone, indicating that a row, column, or range of columns was deleted. These tombstones are kept for at least the period of time defined by the gc_grace_seconds per-table setting. Only then a tombstone can be permanently discarded by compaction.

This scheme allows for very fast deletes (and writes in general), but it's not free: aside from the obvious RAM/disk overhead of tombstones, developers might have to pay a certain price when reading data back if they haven't modelled their data well.

Specifically, tombstones will cause issues if a lot of deletes (especially column-level deletes) is performed and later perform slice queries on rows with a lot of tombstones.

The default value or gc_grace_seconds is 864000 seconds (10 days). In a single-node cluster, it can safely be set to zero. This can be done by the cqlsh tool that is packed together with the Cassandra release. When the cluster is

extended for more nodes, the setting should be edited for correct behaviour and good performance.

In this thesis, I use Cassandra version 3.11.10, that is compatible with used Janusgraph database.

3.3 Bitcoin Core

The Bitcoin Core is the application for downloading the Bitcoin blockchain and is used as the source of most information about Bitcoin transactions.

The Bitcoin Core comes as an application with graphical, console and JSON RPC interfaces. In this thesis, I will use only the console and JSON RPC interfaces. To run the Bitcoin Core in server mode, the client needs to be properly configured.

Following settings need to be set in the configuration file.

- server: Enables the bitcoin daemon to listen for the JSON-RPC commands if set to 1.
- rpcuse and rpcpassword: Configuration of authentication credentials that needs to be provided in the JSON-RPC calls.
- rpcallowip: By default, only RPC connections from localhost are allowed. This property specifies address to be allowed to connect with other hosts.
- rpcbind: Binds the server to given address to listen for JSON-RPC connections.
- rpcport: Port on which the server will listen for requests.
- rpcthreads: The rpcthreads parameter is the number of independent API requests that can be processed in parallel.
- txindex: By default, Bitcoin Core builds a database containing only the transactions related to the user's wallet. To be able to access any transaction with commands like gettransaction. If the txindex property is set to 1, the Bitcoin Core will build a complete transaction index that will be used for such searches.

After configuring mentioned properties, the Bitcoin Core can be started and it will start synchronizing the blockchain and listening for the requests[4].

3.4 Geth

The Geth is the implementation of the Ethereum client which will be used in this for downloading the Ethereum blockchain and retrieving the details about the transactions.

The configuration of the Geth node is done by the parameters when executing the application. The parameters I use are the following.

- syncmode: Geth has three different syncmode options that determine the network that will be used and synchronized with. The default option is "snap", but for this thesis, I use the "full" mode that downloads all blocks (including headers, transactions, and receipts) and generates the state of the blockchain incrementally by executing every block. This allows importing data before the node is fully synchronized. The "snap" mode downloads the current state and then the blocks but does not allow to query the transaction from the start. There is also "light" mode, which relies on other nodes to provide the data, but as there is not enough of nodes that would allow other nodes to connect.
- cache: Megabytes of memory allocated to internal caching. In this thesis I use value 2048.
- maxpeers: Maximum number of network peers. If the value is low, I can cause issued with synchronization. I have chosen the value 50.
- blocksonly: This setting was described in the section where I described the data retrieved from the JSON RPC endpoint. The most important effect these settings have is downloading only the confirmed blocks and lowering the bandwidth.

When the Geth Client is executed, it starts to listen for HTTP requests and creates the geth.ipc file for IPC communication.

Together with the Bitcoin Core client are the main sources of information prepared. However, the clients need to be at least partially synchronized to be able to serve the data. The amount of time it takes clients to fully synchronize is significant[5].

3.5 ChainAnalyzer

The ChainAnalyzer is the main application for extracting the data from blockchain clients and importing them into the graph database Janusgraph.

The application is composed of several Spring services.

- BitcoinClient
- EthereumClient

- DatabaseClient
- BlockChainImporter

Simple schema of the service communication can be seen on 3.2/

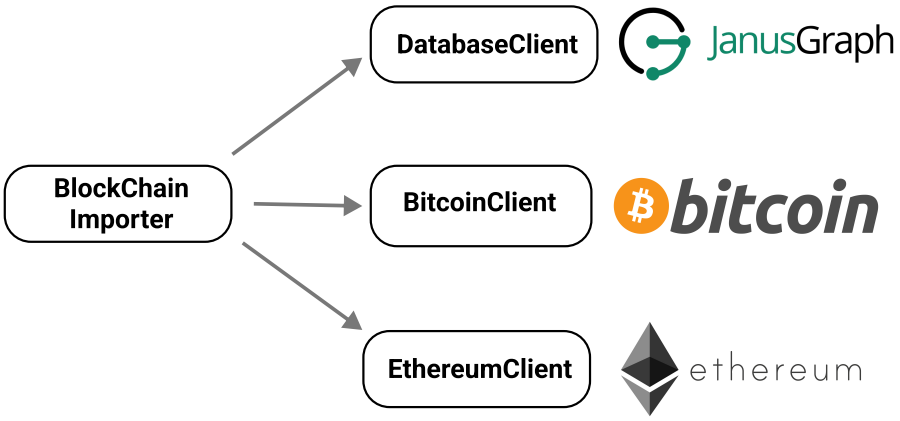

Figure 3.2: Service architecture

In following sections I will describe each service, it's purpose and functions.

3.5.1 BitcoinClient

The *BitcoinClient* service is responsible for retrieving information from the Bitcoin Client. As it is using the JSON-RPC, it is not dependent specifically on the Bitcoin Core client but other data sources, that will use the same interface, can be used.

During the early implementation I considered using a java library created for communication with Bitcoin Clients. An example of such a library is bitcoinj. It is a library for working with the Bitcoin protocol. It can maintain a wallet, send/receive transactions without needing a local copy of Bitcoin Core and has many other advanced features. However the library and classes it is using to represent the historical data from the blockchain do not allow to retrieve all information that can be extracted using the JSON-RPC endpoint. Example of such information is address of the transaction output[58].

For communication via the JSON-RPC endpoint, the body of the HTTP POST requests needs to be JSON with valid format. For that I created have created a data transfer class *BtcRequestDTO* that can be serialized into JSON using the jackson library.

For retrieving the data from the response, I have created several DTO classes that are used for deserializing the response upon arrival.

The majority of the DTO classes represent some parts of the response of *getblock* method called on Bitcoin Core client. The classes contains only the attributes that I want to store or use during the import. Other attributes are ignored.

For retrieval of configuration and creating the bean with *BitcoinClient*, I have implemented the *BitcoinClientDefaultConfiguration* class. This class loads the configuration from the `application-btc.properties` file and creates the *BitcoinClient* object with correct credentials and url.

3.5.2 EthereumClient

The EthereumClient, similarly to the BitcoinClient, is responsible for communication with the Ethereum node client, in this case Geth.

Web3j

For the communication with the client, I have decided to use the web3j library. The web3j is a lightweight, highly modular, reactive, type-safe Java and Android library for working with Smart Contracts and integrating with clients (nodes) on the Ethereum network. It has complete implementation of Ethereum's JSON-RPC client API over HTTP and IPC. It also supports a connection to Infura, so it is easy to switch the clients. This allows you to work with the Ethereum blockchain, without the additional overhead of having to write my own integration code for the platform.

EthereumClientImpl

The EthereumClient implementation use the web3j contains functions to retrieve the block by its height. Actual implementation does not rely on other methods, but in future, other functions can be easily added.

EthereumClientDefaultConfiguration

The class EthereumClientDefaultConfiguration creates Web3j bean with configuration loaded from the configuration file. It is prepared be able to use IPC or REST for communication with the client. The selection is made by setting the property ethereum.method on "ipc" or "rest".

3.5.3 DatabaseClient

DatabaseClient class is responsible for communication with the database. Its implementation offers functions to add nodes into the database or execute traversals or start processes for creating the edges.

Apache Tinkerpop

For working with the Janusgraph database, I have decided to use Apache Tinkerpop library. Apache TinkerPop is an open source Graph Computing Framework. Within itself, TinkerPop represents a large collection of capabilities and technologies and, in its wider ecosystem, an additionally extended world of third-party contributed graph libraries and systems. The library has several implementation for different languages, so it can be used by applications written in Java, C# or Python[55].

The Tinkerpop provide interface for constructing the traversals and then executing them by sending the traversals to the listening gremlin server which will evaluate them and return the results.

DatabaseClientDefaultConfig

The class `DatabaseClientDefaultConfig` serves as a configuration class and create bean with the `GraphTraversalSource` which is then used in the implementation of `DatabaseClient`. After creating the connection to the remote gremlin server, it execute a control check on the connection.

DatabaseClientImpl

The implementation is not in many ways straightforward as it could seem. In the process of development I have encountered several issues I had to work with or around.

The main issue I have faced is the request timeout. As the database after some time contains millions or even billions of nodes, the queries that retrieve nodes or edges from the database get slower. Also, as there are more nodes of some type, some traversal needs to visit majority or even all nodes or edges in the database. This is partially mitigated by the use of indexes but not entirely.

For this reason I construct the traversals with following rules in mind:

- If we know number of the vertexes that should be returned, the traversal should be limited by the `.limit(x)` function.
- Traversals that create edges or nodes and can be divided into smaller segments should be divided and the segments should be executed separately.
- If it is possible to search the node using its internal id (generated by the Janusgraph) it faster then searching by other identifiers, for example hashes.

Those three rules change for example process of deleting all nodes in the database. Instead of calling drop function on all nodes, it is more reliable

to iterate over small portion of vertexes and dropping them first. After the deletion is finished, iterate over next portion of vertexes. This way the gremlin server will not timeout and the the chance or issues is mitigated.

The database client offers 4 types of functions.

- Function for inserting data into database
- Function for creating edges between nodes
- Function for operations over the data in the database
- Support functions

In this part I will describe each type of the function with examples.

Function for inserting data into database has purpose of creating traversal that will insert provided object into the database and executing it over the database. For each node is created corresponding function. For example for the `BtcBlockNode` exists function called `insertBtcBlockNode` that takes one argument of type `BtcBlockNode`.

The code of the function, where the logging is removed, is in the code example 3.2.

```
@Override
public Vertex insertBtcBlockNode(BtcBlockNode node)
    throws IllegalArgumentException {
    if(node == null){
        throw new IllegalArgumentException("Block cant be null");
    }
    Vertex v = g.addV(BTC_BLOCK)
            .property(Att.TYPE, node.getType())
            .property(Att.HASH, node.getHash())
            .property(Att.PREV_HASH, node.getPreviousBlockHash())
            .property(Att.TIME, node.getTime())
            .property(Att.MEDIAN_TIME, node.getMedianTime())
            .property(Att.HEIGHT, node.getHeight())
            .property(Att.NEXT_HASH, node.getNextBlockHash())
            .property(Att.NUM_OF_TX, node.getNumOfTx())
            .property(Att.VERSION, node.getVersion())
            .property(Att.PREV_BLOCK_LINKED,
                node.getPreviousBlockLinked()).next();
    return v;
}
```

Code example 3.2: example of insertBtcBlockNode function without the logging

The function first checks if the value is not null and in case it is null it will throw `IllegalArgumentException`. After the check the function proceeds to creating the traversal by calling the `g.addV(...)` which will create the vertex in the database with specified label. Then by calling the `.property(key, value)` it adds properties to the node. Each attribute of the block is copied. After constructing the traversal, the function for executing `.next()` is called and the Janusgraph database return the vertex containing id and label of created vertex. And that is returned by the function.

Function for creating edges between nodes works on the principle of identifying the nodes that don't have specific edge, then creating the edge between node based on the equality of one or combination of parameters.

Example of such method can be `createBlockBlockLinkBtc`. In this function can be seen application of third rule I mentioned before. The whole body of the function is inside while loop which repeats until stopped from the inside, when there are no more nodes that meet the criteria. The example 3.3 show how the search traversal can look like and how the look is interrupted if there are no results returned. In this example can be also seen the limitation of the number of results.

Then the function continues by iterating over the returned results and executing the traversal which finds both nodes and creates the edge between them. Before executing the traversal, the property, that indicates the node has the edge connected, is changed to true and therefore will not show in the next iteration.

```
GraphTraversal<Vertex, Map<Object, Object>> traversal = g.V()
    .has(Att.TYPE, BTC_BLOCK)
    .has(Att.PREV_BLOCK_LINKED, false)
    .limit(iterationSize).valueMap().with(WithOptions.tokens);
if(!traversal.hasNext()){
    break;
}
while(traversal.hasNext()){
    Map<Object, Object> blockMap = traversal.next();
    BtcBlockNode block = convertVertexValuesToBlockBtc(blockMap);
    g.V(block.getId()).as("b")
            .V().has(Att.TYPE, BTC_BLOCK)
            .has(Att.HASH, block.getPreviousBlockHash())
            .as("p").addE(PREVIOUS).from("b").to("p").select("b")
            .property(Att.PREV_BLOCK_LINKED, true).iterate();
}
```

Code example 3.3: Simplified body of createBlockBlockLinkBtc function

Function for operations over the data in the database is the third

type of function in the `DatabaseClient` and example of such function is `mergeInputAndOutputsBtc`. This function starts similarly to previous type of function as it creates loop that iterates over the returned vertexes and when none are returned, the loop ends.

However the difference is in the body of the loop. This function retrieves two vertexes representing the the input and output of transaction. Merges them together and creates the `btcTransferNode`. This node is inserted into the database. The edges to the transactions and address are also created. And the input and output is deleted.

Support function is the last type I want to describe. As support function I consider functions for committing or roll backing the transaction or retrieving the block with the highest value in height attribute.

3.5.4 BlockChainImporter

The core service in the ChainAnalyzer application is the `BlockChainImporter`. This is a spring service that contains `run()` function that is executed upon the start of the application. Based on the configuration it start to import chosen blockchain.

The import of the each blockchain is executed in iterations. Each iterations consists of importing nodes from the data source and creating connections between imported data. Both parts can be disabled by the configuration and only one of can be executed. This can be useful if the main goal is to import data and the edges will be created afterwards.

Importing of the nodes consists of finding the last block that was imported into the database and retrieving the next one from the blockchain. Then follows the loop in which following steps are executed until the there are blocks to import. Import actual block into the database together with the structures inside. Based on the parameters inside actual block, retrieve from the blockchain client next block. If the block exists, set it as actual block and if it does not exists stop the loop. By this mechanism it is guaranteed the loop will end when the application reached the top of the blockchain ledger.

Importing of edges is in structure much more simple. It consists of calling a functions for creating the edges on the `DatabaseClient`. After one function finishes, next one is called until all functions are called and all edges or transformations are finished. The functions and their count differs with the blockchain.

3.6 Scrapers and data extractors

As described in the analysis, there are several projects that contains valuable information and can be extracted and used to enrich the data from the blockchain client for use in the network analysis. To retrieve the data I have decided to implement set of web scrapers and data extractor that will retrieve

```
private void importEthereumNodes(){
    EthBlock.Block actualBlock = null;
    Long height = databaseClient.getBtcBlockMaxHeight();
    if(height == null){
        actualBlock = ethereumClient.getFirstBlock();
    } else {
        actualBlock = ethereumClient.getBlock(height+1);
    }
    while(actualBlock != null){
        //insert block and included structures into database
        importBlockEth(actualBlock);
        //set actualBlock on next one
        EthBlock.Block nextBlock = ethereumClient.getBlock(
            actualBlock.getNumber().longValue()+1);
        if(nextBlock!= null){
            actualBlock = nextBlock;
        } else {
            actualBlock = null;
        }
    }
}
```

Code example 3.4: Simplified body of importEthereumNodes function

the data and store it inside the relational database. After the data are retrieved, another application will insert them into the database and they can be viewed by other tools or applications like ClueMaker.

In this section I will describe the resources and implemented content scrappers and data extractors, together with the scripts that are used. The resources are following.

- WalletExplorer
- BitcoinAbuse
- OFAC

For each source is created separate python script that extracts the data and insert them to the database. For communication with the databases are created clients.

- PostresClient
- JanusClient

The schema showing the architecture can be seen on image 3.3.

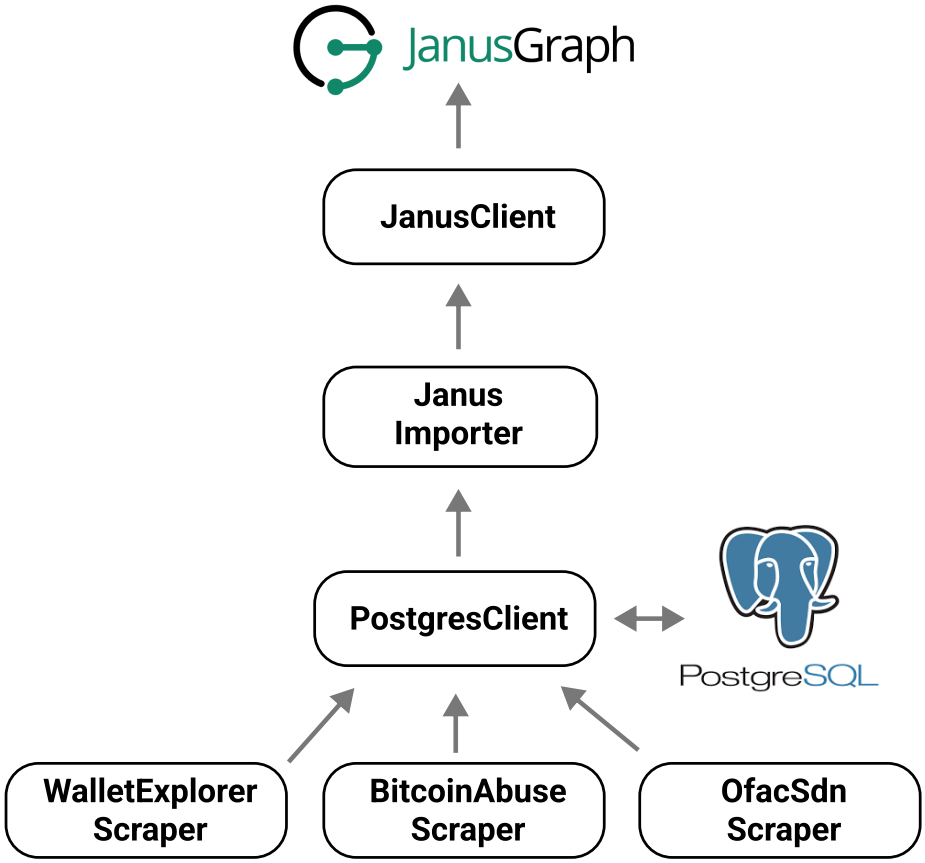

Figure 3.3: Scrapers and data extractors architecture schema

WalletExplorer

The first resource is WalletExplorer. The page does not provide one endpoint that would provide the data. For each wallet, it does provide endpoint with the CSV file containing data about the addresses belonging to the wallet, but to be able to construct the address of the site, one needs to know the name of the wallet. Therefore I have decided to create python script that will crawl the website and gather wanted resources.

Before I started the implementations I checked if the website had the `robots.txt` file. This file gives instructions to web crawlers, bots and robots about the way they can use and move around on the website. It can't be enforced but can help the bots that do not want to harm the server or violate the wishes of the owner. As this file does not exist on the page, I assume the owner is not against my intentions.

On this site I am interested especially in the wallets, their names and addresses that belong to the wallet. Most of the wallets on the site are listed on the main page of the WalletExplorer web.

The main page consists of three parts from the HTML point of view. Topbar, main and foot. Topbar and foot do not contain any useful information. The main part is composed of forms for searching on the web using transaction identifier, address hash or internal wallet identifier. Under the form is table with five columns where each row contains the name of known wallet. Each column has its type of service and can be used as tag for the wallet and addresses. Some rows have more than one wallet inside them. Services with more than one wallet contain links on other wallets with one-word descriptions (old, old2, cold, incoming).

On this page I construct wallet objects for each row in the column. This wallet object contains the name from the row, tag from the name of the column and a link on the site with the detail. If additional wallets are in the row, they are also added. After all columns and rows are collected, the application holds list of wallet objects and can continue to retrieval of addresses.

The application iterates over the wallet objects and for each wallet create the record in the database and then it constructs the URL where the CSV file is located. The pattern of the URL is following:

- `/wallet/{name}/addresses?page={page}&format=csv`

By replacing the name and page for valid values the application construct the url. Using this url the application retrieve the file and extracts its contents. The file contains four columns with address, balance, number of incoming transactions and the height of the last block the address appeared in. The example of the contents can be seen in table 3.1. The names of the columns are replaced with shorter names.

Table 3.1: examples of opcodes and their gas cost

address	**balance**	**in**	**last**
13Q8Us3TDi9ofhSZEJoz9qT7FrH52crvtc	0	1	263880
19cYwZ6bNZN2Xx49jWkYyGFc4hKPcBwPH	0	1	263880
1JExdShGRMRExUMZT7qavdz3NoLNTUdcmt	0	1	263878
12PqYY9Cg388iNcz9keAm159zdRGNr5j8L	0	1	263860
1MU48NGCoe6mS9p5BV4CkaszinmHLw4CyC	0	1	263850
1FMJ7mFFX5eStuRJCTc7MSKGRxSGRJSRC	0	1	263777
1KQqEKtb1wgA9HrwDZpDhDKpEZXahMon92	0	1	263752

For my use case I extract only the address. The application parses the file using the Pandas python library and extracts first column. Then iterates over the addresses and inserts them into the database together with reference

to the wallet it belongs, tag and information about the source so it can be backtracked.

Bitcoinabuse

The Bitcoinabuse site is second web resource I have decided to use in this thesis. The site provides several endpoints that can be used.

First API allows developers to look up the abuse_type_id for use with the report address API. The response is a JSON with the id and name of each type. Right now the Bitcoinabuse distinguish between five type and one for unspecified.

- ransomware: Ransomware is a form of malware designed to encrypt files on a device, rendering any files and the systems that rely on them unusable. Malicious actors then demand ransom in exchange for decryption. Nowadays, it is common to demand the use of cryptocurrencies for the transaction[36].

- darknet market: The darknet is the vast portion of the Internet which can only be accessed using specialized software. Criminals misuse cryptocurrencies for it pseudo-anonymity as it can be used for sale of drugs, firearms, explosives and many more can all be facilitated by these technologies.

- bitcoin tumbler: Also called ”mixers” are systems which randomly crisscross coin with other user's bitcoins so that at the end is created a clean address that the blockchain cannot connect with any of the addresses from which the coins were stolen[36].

- blackmail scam: This is a practice when strangers threaten someone in exchange for cryptocurrencies as a means of extortion. They can claim they hacked the computer of the victim and will release the passwords of files[36].

- sextortion: The sextortion is a form of blackmail scam where the sender of and message claims they have video of the receiver from their webcam performing sexual acts in private, and ask them to pay the amount in cryptocurrencies to keep the video (which does not exist) private[36].

- other: Option for reports that do not qualify for previous types.

The URL has following form.

- `https://www.bitcoinabuse.com/api/abuse-types`

To this URL needs to be appended URL parameter with token to call the API.

The second API endpoint is used for retrieving the distinct reports. The endpoint accepts three URL parameters: api_token to authorize the call, page number as only hundred records is returned and reverse to retrieve oldest reports first. This report is cached and only updates once per hour.

The endpoint listens for GET requests and has following form:

- `https://www.bitcoinabuse.com/api/reports/distinct`

The response is structured as a main JSON object containing properties with information about the response like number of results per page, from and to index of returned records, and others. In the data attribute is a list of objects containing three attributes. Address, count and last_reported_at.

```
{
  "current_page": 1,
  "data": [
    {
      "address": "bc1q4z4wwlrx4c2qjkkp0u9dz0cjeqvvt5a9z2z7z5",
      "count": 1,
      "last_reported_at": "2022-05-03 11:52:02"
    },
    ...
  ],
  "first_page_url": "http:\/\/www.bitcoinabuse.com\/
                     api\/reports\/distinct?page=1",
  "from": 1,
  "next_page_url": "http:\/\/www.bitcoinabuse.com\/
                    api\/reports\/distinct?page=2",
  "path": "http:\/\/www.bitcoinabuse.com\/api\/reports\/distinct",
  "per_page": 100,
  "prev_page_url": null,
  "to": 100
}
```

Code example 3.5: JSON response for distinct reports on Bitcoinabuse

This endpoint could be used to retrieve the addresses but I would not be able to use the type of the abuse to tag the address.

Next endpoint is for checking concrete address. The endpoint is listening for GET request on following URL:

- `https://www.bitcoinabuse.com/api/reports/check`

The endpoint accepts two URL parameters. First is api_token a second is address.

The example of response can be seen on 3.6. It returns the address and data about the reports where the address was included.


```
{
  "address": "12LwjZ6yTBMn3RL7CURwmeDrsNmKtLKAeD",
  "count": 1,
  "first_seen": "2022-05-03 11:06:27",
  "last_seen": "2022-05-03 11:06:27",
  "recent": [
    {
      "abuse_type_id": 4,
      "abuse_type_other": "Relationship scam",
      "description": "Relationship scam",
      "created_at": "2022-05-03T11:06:27.000000Z"
    }
  ]
}
```


Code example 3.6: JSON response for address check on Bitcoinabuse

The last endpoint is to download reports for period of time. The response is a CSV file that with following columns. Id, address, abuse_type_id, abuse_type_other, abuser, description, from_country, from_country_code, created_at.

The endpoint requires the parameters where the first one is api_token and second one is the time_period. Allowed options are 1d, 30d, or forever.

The endpoint URL for downloading the report for last 30 days is following:

- `https://www.bitcoinabuse.com/api/download/30d`

This is the endpoint I have decided to use for this thesis. For extracting data from this file I have created the python script that access the file as a stream and iterates over the rows in the file and extract the address and abuse type. Other columns contain values that does not have specific form and can include unstructured text for the most parts.

During the development I have tested use of function `read_csv` from Pandas library, but the CSV file have non standard format. Even with specific configuration of the function I was not able to parse it into a Pandas Dataframe. The file uses the comma as a delimiter and values that contain quotes, the comma or new line are enclosed in the double quotes. For this reason and other anomalies it is hard to define the rule for parsing. Example of the record can be seen on code example 3.7.

```
96,1HCDAr5zTuBqkNBS4KsqNaDZu73DhWcnH3,1,,951@283.152,
"Ransom: ""Well, I believe, $300 is a fair price for our
little secret. You'll make the payment via Bitcoin to the
below address (if you don't know this, search
""how to buy bitcoin"" in Google).

BTC Address: 1HCDAr5zTuBqkNBS4KsqNaDZu73DhWcnH3
(It is cAsE sensitive, so copy and paste it)"""
,,,2018-07-08T16:37:18.000000Z
```

Code example 3.7: Record from the CSV report from Bitcoinabuse

As the format is not easy to parse, I have implemented the script that reads data from the file and iterates over the lines and extract data from the start of the line. As shown in the example 3.7, it is not true that each line is new record. Some record are in spread across multiple lines.

However the records have at least some structure and rules that can be used for the extraction.

- Each record start at new line.
- The first column values are increasing integers starting from 1.
- Bitcoin addresses have length between 26 and 35 characters.
- Address contains only alphanumeric characters.
- The columns are divided by the comma.
- Type of abuse is represented by integer.

Using these rules I constructed regular expression that can be used to check if line of the document is the start of new record.

The regex can be defined in python with following line of code 3.8

```
regex = re.compile("^\d+,\w{20,45},\d+,")
```

Code example 3.8: Record from the CSV report from Bitcoinabuse

Using this expression I check each line and those that match the regular expression and split then by with the comma character to get list of columns. First three columns I am interested in does not contain the comma character, so I can be sure to retrieve valid values. The second and third columns contain the data that are used for creating the address record in the PostgreSQL database together with the source and type of the cryptocurrency.

OFAC

The OFAC shares the Specially Designated Nationals and Blocked Persons list, also called SDN, that contains companies or institutions that are under the sanctions from the US government. This list is published on the website and also on the following address.

- `https://www.treasury.gov/ofac/downloads/sdn.csv`

The file is in the CSV format and does not have header with name of columns. But they can be extracted from other file formats and they are following: sdn_id, sdn_name, sdn_type, program, title, call_sign, vessel_type, tonnage, gross_tonnage, vessel_flag, vessel_owner, remarks. The file is using the comma as a delimiter and double quotes for strings.

For purposes of this thesis I focus on the sdn_name which contains name of the company or individual. This will be used to represent the wallet. Other columns does not contain information I want to use except the last column with remarks.

```
29584,"POTEKHIN, Danil","individual","CYBER2",-0- ,-0- ,-0- ,-0-
,-0- ,-0- ,-0- ,"DOB 14 Sep 1995; alt. DOB 14 Sep 1990;
alt. DOB 08 Aug 1990; Email Address potekhinl4@bk.ru; Gender Male;
Digital Currency Address - XBT 1Q9UAQbcDezmyouFrzt94t4dSMxgsUfW1X;
alt. Digital Currency Address - XBT
1Kys8fqDen8NGFUJ6AFcXfFW5qquuTH4eh; Digital Currency Address - ETH
0x7F367cC41522cE07553e823bf3be79A889DEbe1B; a.k.a. 'cronuswar';
a.k.a. 'SERGEY, Kireev Valerievich'."
```

Code example 3.9: Record from the SDN list

As can be seen from the example 3.9 the last column contains several parts of information, delimited by the semicolon. In this example it contains dates, email address, gender, addresses of owned addresses in specified blockchain and other names under which the individual is know.

I am concerned about the addresses. Remarks with the addresses contains one main address and then alternative addresses mark with the word alt. Before each address is a shortcut representing the blockchain, for example XBT for Bitcoin, ETH for Ethereum, LTC for Lite Coin , ZEC for the Zcash and more. I want to extract only the XBT and ETH addresses.

The implementation uses the Pandas library and its function `read\_csv` to load the file and transform it into the data frame. Before extracting data it checks, it the last column contains string ”Digital Currency Address” that is in the records with addresses.

From records with mentioned string is extracted name of the individual and new wallet is inserted into the PostgreSQL database. The the last column

is divided into separate parts of information by the splitting the string with the semicolon as delimiter. This string is then cleared of redundant information and only the type of the currency and hash of the address is stored.

After extraction of wanted data, the address is inserted into the database together with link to the data source, wallet, tag, marking the address as under the sanctions, and blockchain.

PostgresClient

After the data are extracted from the data sources by mentioned python scripts, they are stored in the PostgreSQL database. For communication with the database is created separate class PostgresClient which serves for communication with the database.

The client implements following functions.

- connect: Creates the connection to the database.
- close: Serves to close the connection to the database.
- insertDatasource: Inserts the data source record into the database if it does not exists.
- insertAddress: Inserts the address record into the database if it does not exists.
- insertCrawledResource: Inserts the crawled resource record into the database if it does not exists.
- insertWallet: Inserts the wallet record into the database if it does not exists.
- getCrawledResource: Retrieves the crawled resource from the database based on the provided url.
- rollback: Rollbacks current transaction.
- commit: Commits the changes.
- getWalletAddressIterator: Retrieves data about the addresses and wallet they belong to. Returns iterator, which is used to stream the data from the database.

JanusClient

Second database client is JanusClient. Similarly to the PostgresClient provides interface for communication with the JanusGraph database. For purpose of inserting wallet and address is not necessary to implement all CRUD operations and only a small subset is implemented.

It implements following functions.

- connect: Creates the connection to the database and prepares the traversal.
- close: Closes the connection to the database.
- insertWallet: Inserts the wallet into the JanusGraph database if it does not already exists.
- insertAddress: Inserts the wallet into the JanusGraph database if it does not already exists. If it does, it updates the tag and wallet.

JanusImporter

The final orchestration of the data transfer from the PostreSQL database to JanusGraph is done by the JanusClient. It uses the JanusClient and PostgresClient to retrieve data from one and insert them into the other.

The PostgresClient provides the iterator that is used to gradually retrieve the wallets and addresses and if the data are in correct form, they are inserted into the database. If the address or wallet already exists, the missing information is added.

3.7 ClueMaker

One of the main goals of this thesis is to extend ClueMaker applications for connector to the selected graph database. As I have chosen the Janusgraph, the connector will be implemented for this database system. The ClueMaker consists of two parts that have separate projects. ClueMaker and ClueMaker Configurator. I will describe changes in each application separately.

3.7.1 ClueMaker Application

Technologies

Both ClueMaker and ClueMaker Configurator are developed in the Java programming language. For management of the dependencies is used Apache Maven. For the development is used the NetBeans Platform which is generic framework primarily for Java desktop applications. It offers development of applications with pluggable components and also provide tools for building a graphical interface.

JanusConfiguration

The `JanusConfiguration` is class that implements `DatasourceConfiguration` and serves as a class to represent the basic configuration of the data source in form of location of the database and credentials needed for establishing the

connection. The object with the values is created in the Configurator with the data the user entered and then imported into the ClueMaker Application.

The connection to Janusgraph database can be established by the host address and port on which the gremlin server is listening. The credentials in form of username and password are not required in default configurations, but can be defined in the Janusgraph configuration. For this reason the `JanusConfiguration` contains the host, port, username and password. I also prepared the

JanusCypherDialect

The `JanusGremlinDialect` is the builder of the traversals that are executed over the database and thanks to which the data are retrieved. This class allow the ClueMaker to translate user defined criteria and relations between the attributes into the gremlin query. User s the builder of the traversals that are executed over the database and thanks to which the data are retrieved. This class allow the ClueMaker to translate user defined criteria and relations between the attributes into the gremlin query. User can use the interface of the ClueMaker application to define tables, entities, attributes and relations. When the definition is finished and user wants visualize and interact with the data source, ClueMaker needs to make sure the the defined configuration can be translated into a query that can be send to the target system and it will be able to understand the the query and return desired results. For this purpose each importer implements class that can do such translation.

The class JanusCypherDialect extends AbstractDialect and overrides the implementation of visit methods. Visit methods are creating the the gremlin queries based on the criteria passed as the parameter. Criteria can be of several types. Before I start to describe parts I implemented, I would like to describe key components I will be working with.

ComposableCriteria: ComposableCriteria are structures that represent nodes in abstract syntax tree, also known as AST. One criteria contains more operators inside and this way creates the hierarchical structure typical for the AST. Criteria that directly extends the ComposableCriteria class and can behave as a nodes in the tree are right now following:

- And: And is a criteria that connects it's children with the logic operator AND.
- Or: This criteria logically joins one or more children using the logical operators OR
- Not: The negation of whole criteria sub tree.

AttributeCriteria: The criteria that usually works with the mapped attribute. Typically don't have any children criteria and only express condition

on the attribute, behaving as a leaf in the AST. It can contain multiple values. Classes that extends directly the AttributeCriteria are following:

- Equals: Represent equality relation between the attribute and the value.
- Contains: Criteria that indicates if specified value is part of the attribute value.
- GreaterThan: Express the relation "greater then" between the attribute and provided value. It's possible to set the criteria to be inclusive.
- In: Criteria that indicates if the attribute's value is part of the provided set of values.
- NotIn: Indicates if the attribute is not equal to any of the values in provided set of values.
- SmallerThan: Express the relation "greater then" between the attribute and provided value. Criteria that indicates if specified value is part of the attribute value.

The AttributeCriteria classes accept value that can be represented as a number, string, date or list. Most query languages treat each type differently and therefore the builder of the query needs to know what it is working with. For such cases, the ClueMaker implementation has defined value types that implement the interface called `Value`. Such classes are StringValue, NumberValue, DateTimeValue, ListValue and BooleanValue.

Now when the basic components used in the JanusCypherDialect are introduced, I can describe the implementation. I will describe first example into bigger detail then then the others.

And is a first CompositeCriteria that can be passed to the visit method. It can contain zero, one or more children criteria. It serves for composing criteria where all children needs to be true.

The operator is in gremlin expressed by `.and(... , ... , ...)` operation. It accepts one or more conditions or boolean values divided by the comma. Example where is used the and operator in the gremlin query can be seen in 3.10

```
g.V().and(has('code','AUS'),has('icao','KAUS'))
```

Code example 3.10: example of insertBtcBlockNode function without the logging

By this logic, the transaction can be composed from following steps.

- start the query with ".and("

- translate each child criteria
- append them to the query and insert comma between them
- end the query with closing bracket

Described steps are close to the final implementation. There are few cases that needs to be taken into the account.

- What if the there are two And operators inside, for example the expression `AND(AND(expr))`. This would create `.and(.and(expr))`. The dot can be only before the first operator. For this reason there is a need for implementing mechanism that will track number of opened operators and only before the first one will be added the starting dot.
- What if the the And criteria is empty. This would in mentioned case result in `.and(and())` and JanusGraph does not allow empty operator. This can be mitigated by conditioning the creation of the operator if there are child criteria inside.

The final implementations of the visit and support methods is in code example 3.11.

analogically to the And CompositeCriteria are implemented others with the change in the gremlin operator.

The AttributeCriteria can be implemented with similar way, just without the iteration across children. They are predicates, related to the property, that are true or false. Basic predicates are constructed with following query `.has("key", ...)`. However there are special properties where the syntax differs.

- label: For the label property the syntac of the predicate has following form: `.has(label, )`. The difference is that name of the property is not enclosed in double quotes.
- inV and outV: Those attributes represent id of the the vertex from or to which the edge is connected. The query with the predicate on inV or outV properties needs to have following form, plus they need to be closed with two brackets instead of one:
  - `where(__.inV().has(id, ...))`
  - `where(__.outV().has(id, ...))`

As an example of implementation can be used the implementation of the visit method for `GreaterThan` criteria, seen in 3.12.

The `startPredicate` method does the decision on which form of predicate form choose based on the property name.

```
@Override
public void visit(And and) {
    if(and.isEmpty()) {
        return;
    }
    startLogicalConnective("and");
    for (Criteria child : and.getChildren()) {
        if (!child.isEmpty()) {
            child.accept(this);
            this.builder.append(",");
        }
    }
    endLogicalConnective();
}

private void startLogicalConnective(String logicalConnective) {
    if (openLogicalConnective == 0) {
        this.builder.append(".");
    }
    this.builder.append(logicalConnective).append("(");
    openLogicalConnective++;
}

private void endLogicalConnective() {
    openLogicalConnective--;
    this.builder.append(")");
}
```

Code example 3.11: visit(And and) method implementation

In the example 3.12 can be seen the value is also translated in a some way and then inserted into the `gt()` predicate. This is due to different behaviour for different types of values. The distinction is done by the visit method that takes the type as a parameter.

This distinction can be seen in example 3.13, where the numeric value is appended without structures around, but the StringValue class is enclosed by the double quotes.

By defining the visit methods, and correctly translating the hierarchical structure of operators and predicates, the ClueMaker application is able to communicate with the database with already defined mechanisms to transform users commands.

```
@Override
public void visit(GreaterThan greaterThan) {
    startPredicate(greaterThan.getName());
    this.builder.append("gt(");
    greaterThan.getValue().accept(this);
    this.builder.append(")");
    endPredicate();
}
```

Code example 3.12: visit(GreaterThan greaterThan) method implementation

```
public void visit(NumberValue value) {
    this.builder.append(value.getValue().longValue());
}

@Override
public void visit(StringValue value) {
    this.builder
            .append("\"")
            .append(value.getValue())
            .append("\"");
}
```

Code example 3.13: visit(NumberValue value) and visit(StringValue value) method implementation

JanusDatasouceType

The JanusDatasouceType defines the type of the data source and the importer that can be used to communicate with it. Without this class the Configurator would not know the details about the datasource and what implementations are compatible.

It also serves for creation of empty configuration, that is filled in the Configurator.

JanusGetMetaDataTask

When the ClueMaker application retrieves the data from the data source and displays them, it is done in form of task. The task for the interaction with JanusGraph database consists of following attributes.

- token: Token is representing the current state of the task.
- metadata: Metadata represent the result returned from the data source. Janusgraph returns the attributes in Map collection. where the the keys

are names of the properties.

- message: Message shown to the user about the state of the task.
- importer: Importer that is used for communication with the database.
- userQuery: The gremlin query that will be executed by the importer on the database.
- exception: Storage of exceptions, if some are thrown.
- canceled: The flag used to indicate, if the task have been closed by the user in the middle of process.

JanusImporter

The JanusImporter class is the main connection between the database and the ClueMaker. It extends AbstractImporter and implements ReportImporter.

The JanusImporter is used for following goals.

- Initialize connection to the database.
- Stores the connection in the GraphTraversalSource object, that is then used later for communication.
- Executes the traversals provided by the tasks and retrieves the response.
- Maps properties of returned vertexes and edges to properties defined in the mapping together with their type.

In early stages of development, I used for the creation of the connection basic approach used in the ChainAnalysis application. The approach can be seen on the code example 3.14.

```
GraphTraversalSource g = traversal().withRemote(
    DriverRemoteConnection.using(host, port));

g.V().limit();
```

Code example 3.14: Basic approach for connection to the JanusGraph database

This approch creates the GraphTraversalSource and by calling functions on this object, the traversal is created. Then by the `.next()` is executed upon the database. However this approach is no possible to use in this case. Instead of calling functions on the traversal object, it is needed to execute query provided in the text form.

For this reason, I have decided to use GremlinGroovyScriptEngine. This engine provides methods to compile and evaluate Gremlin scripts in the text form. Before executing the queries, it needs to be configured with the binding to the graph in the database and the GraphTraversalSource.

JanusMappingConfiguration

The configurations holding the gremlin query specified by the used in the configuration.

JanusUtils

The JanusUtils class serves as a collection of static functions used in the importer and the dialect. For example traslation of the Java data type to the Attribute Type. Or function for resolving the type of the identifies provided by the JanusGraph.

SpecialElementProperties

Serves as an Enum class for storing names of special properties that have different behaviour then

3.7.2 Configurator

After implementing mentioned parts, the ClueMaker application is able to connect to database and execute generated queries. But before the users can use the application, they need to define the data source. As the JanusGraph connector is newly defined importer and thus a new form in the Configurator needs to be created.

During the development, I have identified two parts of GUI, where the already existing implementation can not be used. The definition of the data source and mapping of entities in the ClueMaker on nodes and edges in the JanusGraph database. For this reason the JanusQueryEditorPanel form and JanusEditorPanel form will be implemented. To support the functionality they should provide, I have created classes JanusEditorFactory, JanusQueryEditorPresenter and JanusQueryEditorPresenter.

JanusEditorFactory

This class implements a DatasourceEditorFactory which servers as a creator of objects, which are parts of data source configuration.

The factory implements three functions that are required from the DatasourceEditorFactory.

First is `getDatasourceEditor` function that creates the editor for the connection to the database. In this function is the JanusEditorPresenter is initialised for the provided data source type. When the presenter is constructed,

the JanusEditorPanel that is the view of the editor is created and passed to the presenter, so it has access to its components.

Second function is `getQueryEditor` which is the text field for constructing the queries in the mapping configuration. It initialise the `JanusQueryEditorPanel` which is the view of the query editor and is passed back to the presenter.

Third function is `isSuitableFor` and it only checks if the passed data source type is instance of correct data source. In this case the `JanusDatasourceType`.

JanusQueryEditorPresenter

The `JanusQueryEditorPresenter` is the component for executing logic related to the test of the connection to the database. Aside from the constructor and setters for the panel with the connection details, it also implements test of the connection to the database. This is done by creation of the `TestJanusConnectionTask` and then executing it. After the connection check, it pass the information to the panel, so the user is informed about the result.

TestJanusConnectionTask

The TestJanusConnectionTask extends TestConnectionTask and is meant for testing the ability of ClueMaker Configurator application to create a connection to the data source defined by the configuration.

With the TinkerPop library it is possible to create a traversal on remote database. When the traversal is created, the connection is not established as it is created when the first query is executed. So for the purpose of checking the connection, I constructed the traversal, that should be as minimal as possible, as seen on the code example3.15. The traversal consists of retrieving at most one random vertex. The limitation ensures the query does not need index or walk over larger amount of node.

JanusQueryEditorPanel

JanusQueryEditorPanel represents form for creating the gremlin queries for mappings defined in the configuration. It serves for loading attributes of entities in the database so the user does not have to list each attribute and its type.

It consists of two parts. The java implementation and the GUI form definition. Together they create the component that can be inserted into the Configurator and displayed to the user.

The GUI definition is a file with the .form file extension is XML document with the definition of the field, types of the components included and layout. It can be defined or edited in the NetBeans IDE or by direct changes in the text of XML file.

```
private void testConnection() throws Exception {
    GraphTraversalSource g = traversal().withRemote(
        DriverRemoteConnection.using(
            configuration.getHost(), configuration.getPort()));
    GremlinGroovyScriptEngine engine =
        new GremlinGroovyScriptEngine();
    Bindings bindings = engine.createBindings();
    bindings.put("g", g);
    try {
        DefaultGraphTraversal eval = (DefaultGraphTraversal)
            engine.eval("g.V().limit(1)", bindings);
        if (!eval.hasNext()) {
            throw new IllegalStateException(
                "Could not find any verticies configured graph.");
        }
    }catch(ScriptException ex) {
        throw ex;
    } finally {
        g.close();
    }
}
```

Code example 3.15: Implementation of the connection test

The JanusQueryEditorPanel implementation controls the form behaviour and the components inside. It initializes the components, sets their properties, access or edit its values and add listeners for events on the form. Part of the source code is generated by the NetBeans IDE and should not be tempered with. This specific form has simple layout with following components.

- Label marking the query text field
- TextArea where the gremlin traversal can be entered.
- The status label, where the possible warnings are displayed.
- Button that starts the execution of the traversal when clicked.

The result can be seen on the image 5.19.

JanusEditorPanel

JanusEditorPanel is also consisting of two parts. The form layout is composed of following components.

- Row with the label and select box for selection of the protocol. Right now is implemented only one protocol but the TinkerPop supports the HTTP and web socket connection, so it can be extended in the future.
- Row with the label and text field for entering the host address.
- Row with the label and text field for entering the port on which the server listens.
- Row with the label and text field for entering username.
- Row with the label and text field for entering password.
- Label for the state of the connection.
- Button to execute the test of the connection.

The name and password are not required. The JanusGraph database with default configuration does not validate the access rights to the database, but it can be configured.

The result can be seen on the image 5.19.

3.7.3 Results

By implementing described components of the JanusGraph connector and correctly integrating them into the project modules, the connection to the database can be established, nodes and relations can be defined and mapped. The ClueMaker application is the able to use defined configuration to retrieve data from the database and use the application visualising and analytical capabilities to inspect them.

3.8 Testing

In this work I have implemented application to import transaction data, from the Bitcoin Core and Geth clients, into the JanusGraph database. To view and inspect stored data I added a connector for the JanusGraph into the ClueMaker application.

To test the implementation I have included the Unit test in the ChainAnalyzer application which will be described in the following section 3.8.1.

To test the functionality of the connector in the ClueMaker application, I have decided to do a test, where I will use reliable sources as `blockchain.com/explorer` and `etherscan.io` to check selected transactions in the ClueMaker for validity of the data and if the connections were created properly. This will be done in the section 3.8.2.

3.8.1 Unit Tests

For testing the ChainAnalyzer Application I use unit tests that check the functionality of separated parts of the code. Except unit test on transformations of the data from DTO object to the Map of values, returned from the database, I use also use testing JanusGraph database to test the interactions and edge creations.

The `DatabaseClient` implementation is tested using the testing configuration bean calles `DatabaseClientTestConfig`. This configurations creates `GraphTraversalSource` by using Tinkerpop `JanusGraphFactory` which can be used to create connection to remote database but in this case is used for creating in-memory database. This database is kept in memory so it should not be used for large import of data but for the unit testing is very useful. Together with the storage backend is set property for schema creation so the database does not create the types automatically and uses only defined schema. After initialising the in-memory database, the management manager is opened and the schema is imported. If during the tests is inserted node or edge with unknown label, the test will not pass and developers can investigate more.

This in-memory database is used then to check functionality of the functions or operations. Before each test the database is cleared by dropping all inserted nodes and edges. For testing of insertion of nodes or edges are used real data extracted from the Bitcoin or Ethereum client. The real data are then used for testing the creation of edges and transformation.

3.8.2 Test using public explorer

The Unit tests validate the separate small parts of code are working properly. To test the application as a whole, I have chosen to do a test of the data against the public explorers `blockchain.com/explorer` and `etherscan.io`. For each blockchain analyzed in this thesis I select one block with transactions and inspect, if the values in the JanusGraph databases retrieved by the ClueMaker are identical.

Bitcoin

First blockchain I will test is the Bitcoin. For purpose of this testing I have found block with 4 transactions inside. It is a block number 89182 and the transactions have from one up to three inputs and one output each. The data displayed in the web tool can be seen on image 5.9.

- Hash: 000000000013c2e9b07ef258efdabf7c7c8
  2e8ca8c307cc73e0353f1f69c06b8
- Height: 89182

- Timestamp: 2010-11-02 16:10

From the image 5.14 is seen the hash and heigh are identical, only the timestamp is different. The timestamp stored in the JanusGraph is in numeric representation. If the value is converted into the unix timestamp, we receive same results, which I tested by using the online tool on the `epochconverter.com` where the numeric value can be easily converted.

The block contains 4 transaction. Will choose one with most inputs and that is the second in the block and second on the image 5.10. The transaction on the image 5.11 has following attributes that are also being stored in the JanusGraph database. Other attributes are derived or now useful for future analysis.

- Hash: 4a04bc650f9ee7ce4723d7fbe9ec7c5e895a
  eec20ae4ec53fd3887e32db307c2
- Received Time: 2010-11-02 16:10

As can be seen in 5.15 the node in the ClueMaker contains some data that are not displayed by the `blockchain.com/explorer`.

The transaction has 3 inputs and 1 output. I will compare the second input with the transfer node incoming into the transaction. The transfer node as mentioned before is merged input and output so it will contain also information about the output, that is unlocked by the input on the image 5.12. The attribute most people will be interested in is the value. If we compare the images 5.12 and 5.16, the value is correct and the address also.

Now I move the attention to the transaction output on images 5.13 and 5.17 where the values are also the same. The same value is spent and the address is also correct.

By this test I have confirmed that almost randomly selected Bitcoin transaction is correctly stored in the database and can be explored within the ClueMaker application.

Ethereum

For retrieving single information about the Ethereum blockchain is suitable website called `etherscan.io`. It is one of the most popular explorers as it contains a lot of additional data, but it does not visualize the data in a graph but in a table. But to test the application this will be sufficient.

For the testing I have chosen block with the number 55875 as it has 17 transaction. Testing with more transactions inside a block it would be to spread in the space and there would be a lot of items on the screen.

On the image 5.1 can be seen the summary of information about the block. The `etherscan.io` does store all the data about the block, but I will list the attributes and values that stored in the JanusGraph.

- Block Height: 55875
- Timestamp: Aug-09-2015 12:01:13 AM +UTC
- Number of transactions: 17
- Mined by: 0x9746c7e1ef2bd21ff3997fa467593a89cb852bd0
- Block Reward: 5.021232251018888 Ether
- Extra Data: Geth/v1.0.1/windows/go1.4.2
  (Hex:0x476574682f76312e302e312f77696e646f77732f676f312e342e32)

On the image 5.5 can be seen visualization of the block and its transactions. On the left side inside the panel, there are data stored in the block node. In the list below are the values from the node, with the labels translated into the form used for the previous list of values.

- Block Height: 55875
- Timestamp: 1439078473
- Number of transactions: 17
- Mined by: 0x9746c7e1ef2bd21ff3997fa467593a89cb852bd0
- Block Reward: 5021232251018888000
- Extra Data: 0x476574682f76312e302e312f77696e646f77732f676f312e342e32

If we compare the values, the block height, number of transaction, miner, are exactly the same. Other attributes differs but only in the representation. The block reward differs in the decimal places as the value in the database is stored in wei units and etherscan shows the value in Ether units. The last attribute is containing extra data and the online explorer shows the value already decoded.

The website is able to list the transaction in table, seen in 5.2. For the test I will compare first and second transaction against the ClueMaker.

The first transaction is can be seen on the image 5.3, where it is displayed by the etherscan explorer. The stored values are following.

- Transaction Hash: 0xaf5e314c62c84adb7767c051cd11d4f
  5a930e5bd13ab8dd3331372355c02c8f9
- From: 0xe6a7a1d47ff21b6321162aea7c6cb457d5476bca
- To: 0x3b7ce4d16316f594a3c5ebb7ccc4c022fc2858c6

- Value: 3.716433643197706 Ether
- Fee: 0.001315380928488 Ether
- Input data: 0x

On the image 5.6 can be seen the transaction in the ClueMaker. The values are the same, except the representation of the value and transaction fee.

And the second transaction in this block is shown on the image 5.4. The values are following.

- Transaction Hash: 0x34049218c868a1df994e734020f8eca1 7d1473af756b209d3bcd3f3a755771c4
- From: 0xe6a7a1d47ff21b6321162aea7c6cb457d5476bca
- To: 0xe53fe5b196a315de96297181e99f6c22a9313fd3
- Value: 3.682035800243123 Ether
- Fee: 0.001255590886284 Ether
- Input data: 0x

And as it was in the first case, the values in the ClueMaker application, on image 5.7, the values are the same as in the etherscan tool.

From the image 5.2 is seen, the there is one sender who send all the transaction in the block. I expanded the transactions for addresses from which were the funds sent and to which address. On the image 5.8 can be seen how the transactions between addresses can be nicely visualized in the ClueMaker.

Chapter 4

Optimizations and support tools

Chapter 5

Attachments

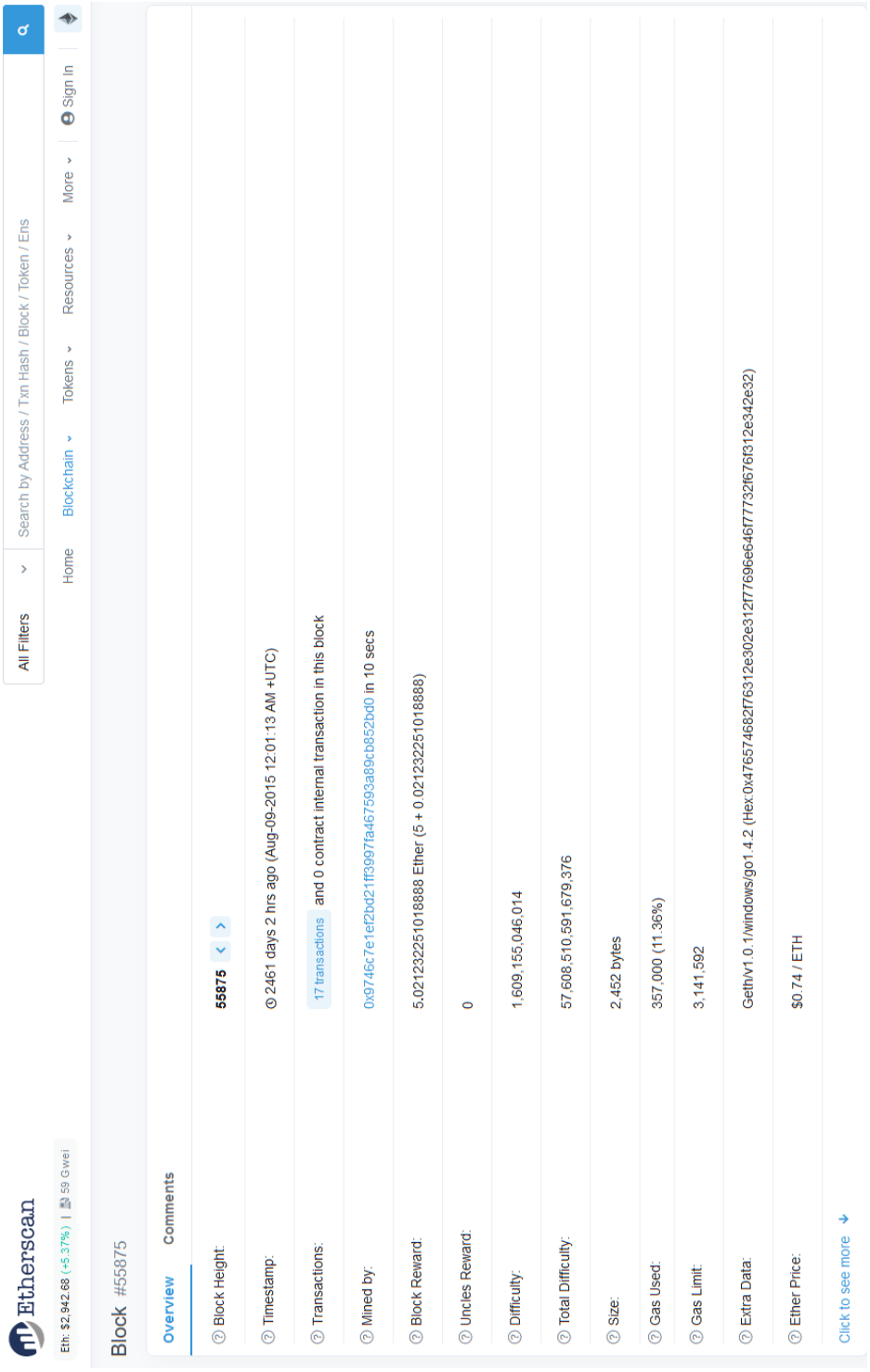

Figure 5.1: Etherscan.io Block #55875

A total of 17 transactions found

First < Page 1 of 1 > Last

Txn Hash	Method ⓘ	Block	Age	From	To	Value	Txn Fee
0xaf5e314c62c84adb77...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0x3b7ce4d16316f594a3...	3.716433643197706 Ether	0.00125559
0x34049218c868a1df99...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0xe53fe5b196a315de96...	3.682035800243123 Ether	0.00125559
0x2fcc53d09d70017f997...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0x360a45bb0fd4b19809...	3.600964007852423 Ether	0.00125559
0x016a18171a5ae933ac...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0x6b4971f56172fe5656...	3.578216183474669 Ether	0.00125559
0xb6ea12166aeaa70ce1b...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0x6dc8a46ab80ed97e40...	3.565520175049329 Ether	0.00125559
0xc0f63848441d9aee0c...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0xd79aff13ba2da75d46...	3.534821852337626 Ether	0.00125559
0xdc613a59452f4d911e...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0xaa357779ef4dfbed0b3...	3.532233011993705 Ether	0.00125559
0x81f53a0794b6be3fe6...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0xbb9fd5277ee852d8a7...	3.513485746944437 Ether	0.00125559
0x6efbe0735d3829db2c...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0x34bdaf0a5ed72734a4...	3.473281310677717 Ether	0.00125559
0xd3ad117c2bb1f08fd57...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0xf64ee501546e0d74461...	3.398459083065179 Ether	0.00125559
0x523c08553d0a0f927f1...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0x853bff007d2694e0526...	3.356016737120956 Ether	0.00125559
0x8a2514a2756aadd0a2...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0xcadcea36ec3feb7401...	3.355441398484546 Ether	0.00125559
0x7a5fff5f188bc5232b02...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0xd9de820a23c38bcfe2...	3.32771956494992 2 Ether	0.00125559
0xeaf894dea9a5df8228...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0x9b80bd38c4afe0942f8...	3.276718937733633 Ether	0.00125559
0x49cbd3e94ad2b5a341...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0x64be1c1a0198370f53...	3.273140972881982 Ether	0.00125559
0xb3b04d36f18c8281a1...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0xf178805ea39c903a49...	3.257484508205 4 Ether	0.00125559
0x5c3d6418522999 38c71...	Transfer	55875	2461 days 2 hrs ago	Ethpool	0x523e93ed2d488b01e...	3.189574191134373 Ether	0.0014279

Show 50 Records

First < Page 1 of 1 > Last

Figure 5.2: Etherscan.io Block #55875 Transaction list

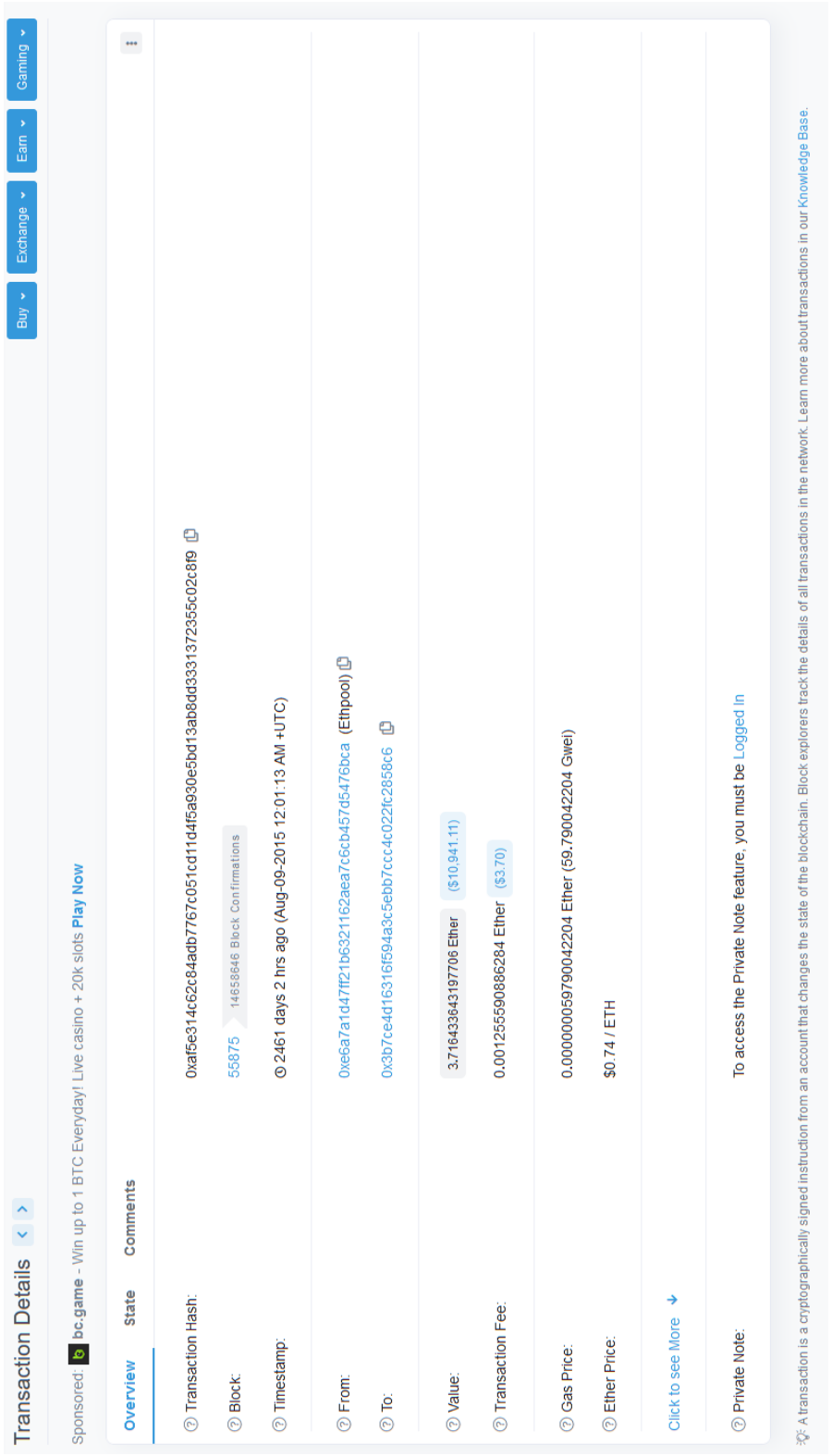

Figure 5.3: Etherscan.io Block #55875 Transaction #1

Figure 5.4: Etherscan.io Block #55875 Transaction #1

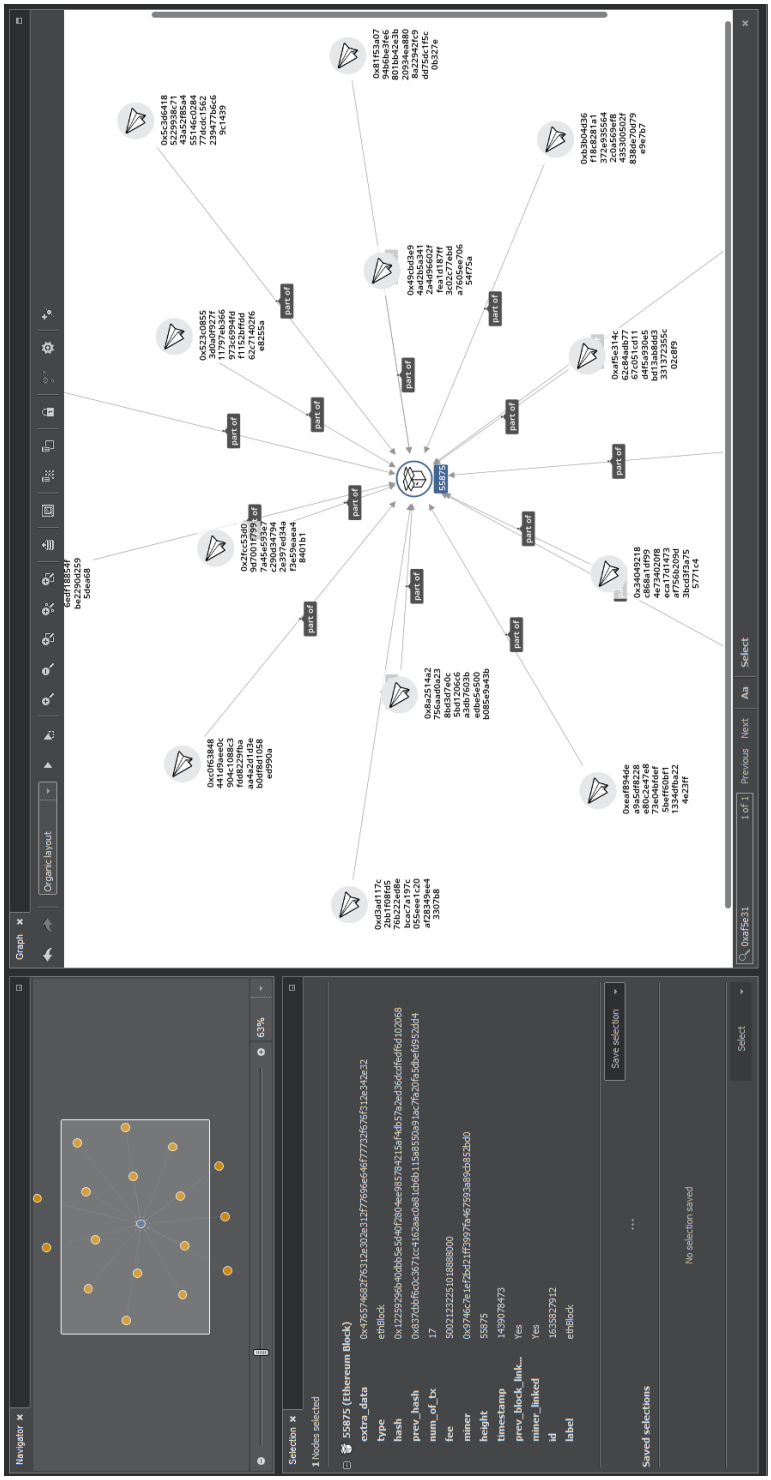

Figure 5.5: ClueMaker Ethereum Block #55875

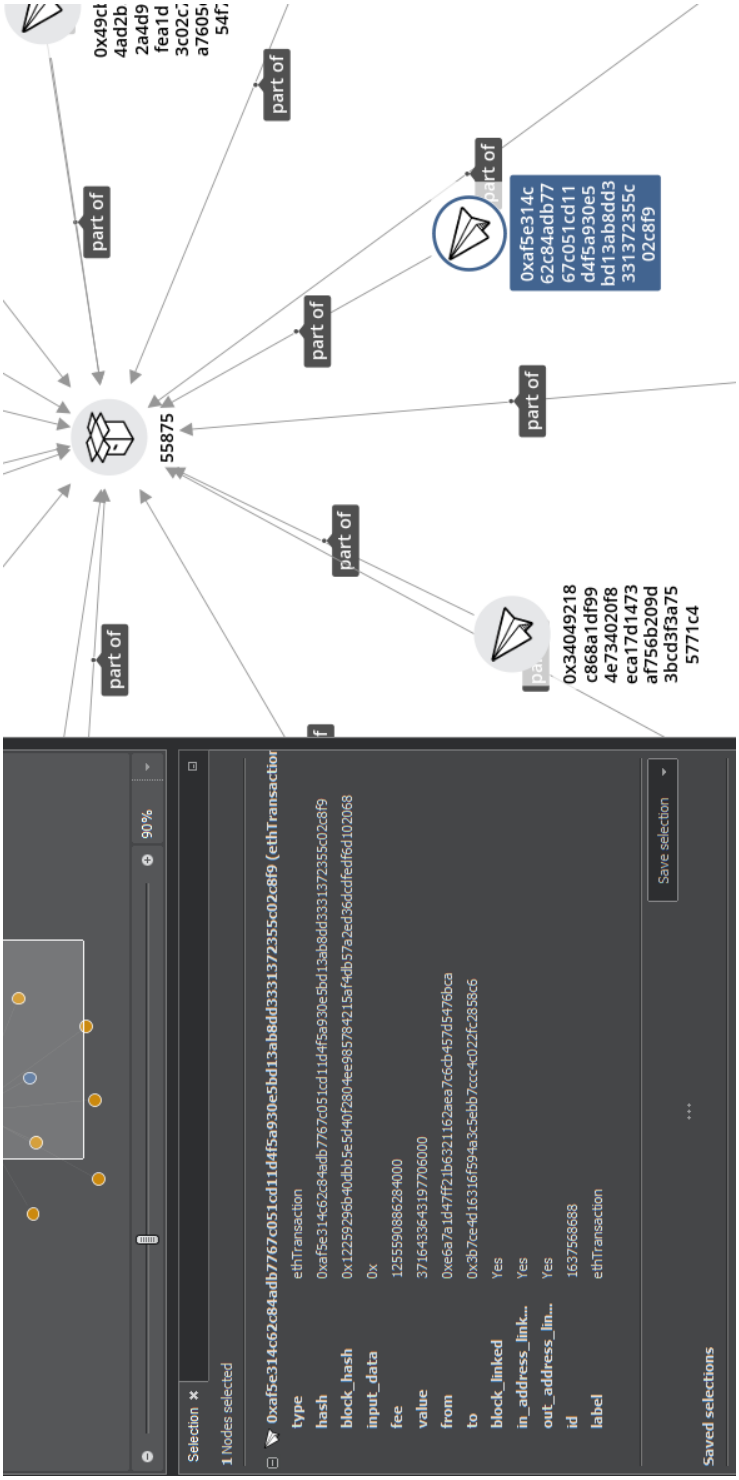

Figure 5.6: ClueMaker Ethereum Block #55875 Transaction #1

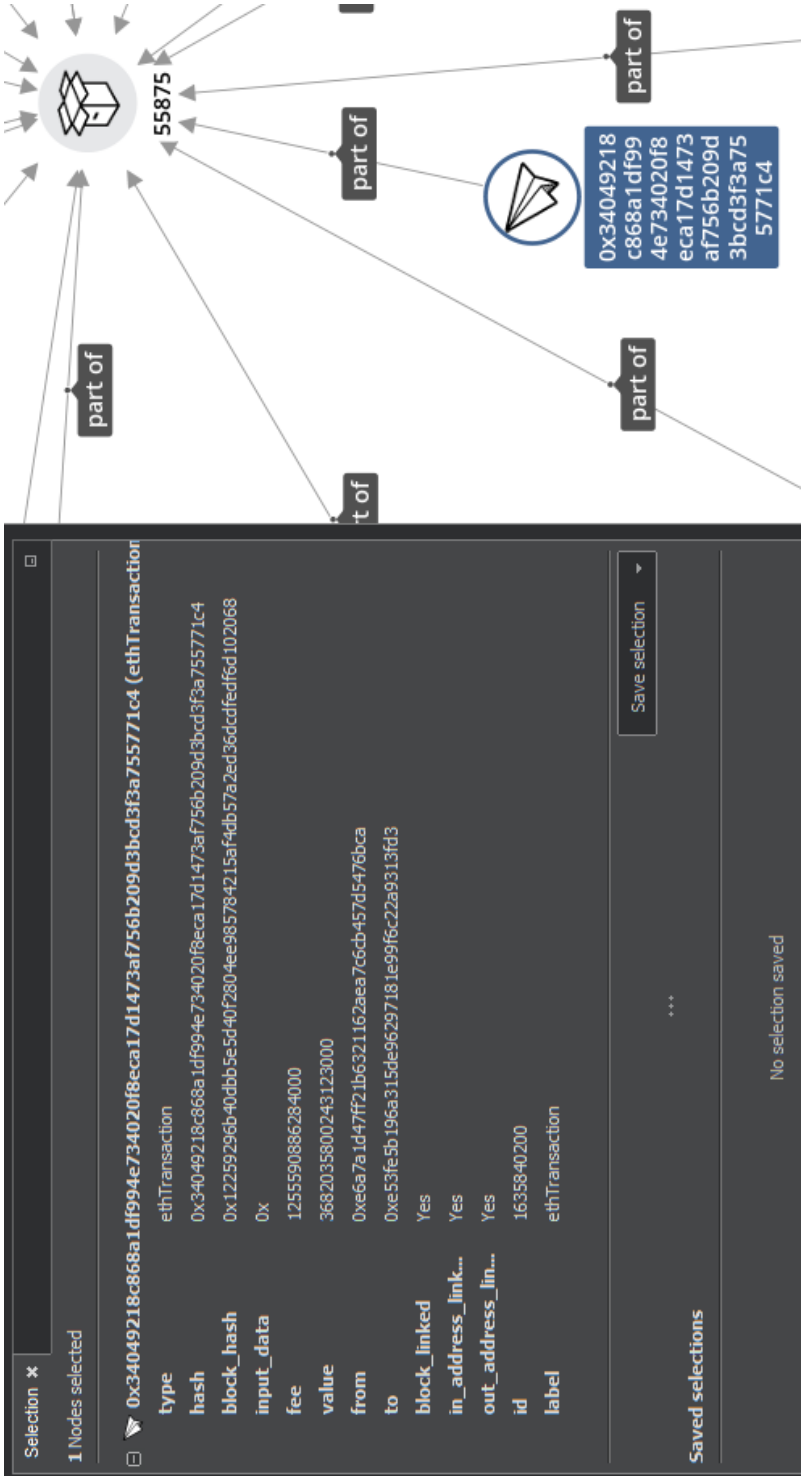

Figure 5.7: ClueMaker Ethereum Block #55875 Transaction #1

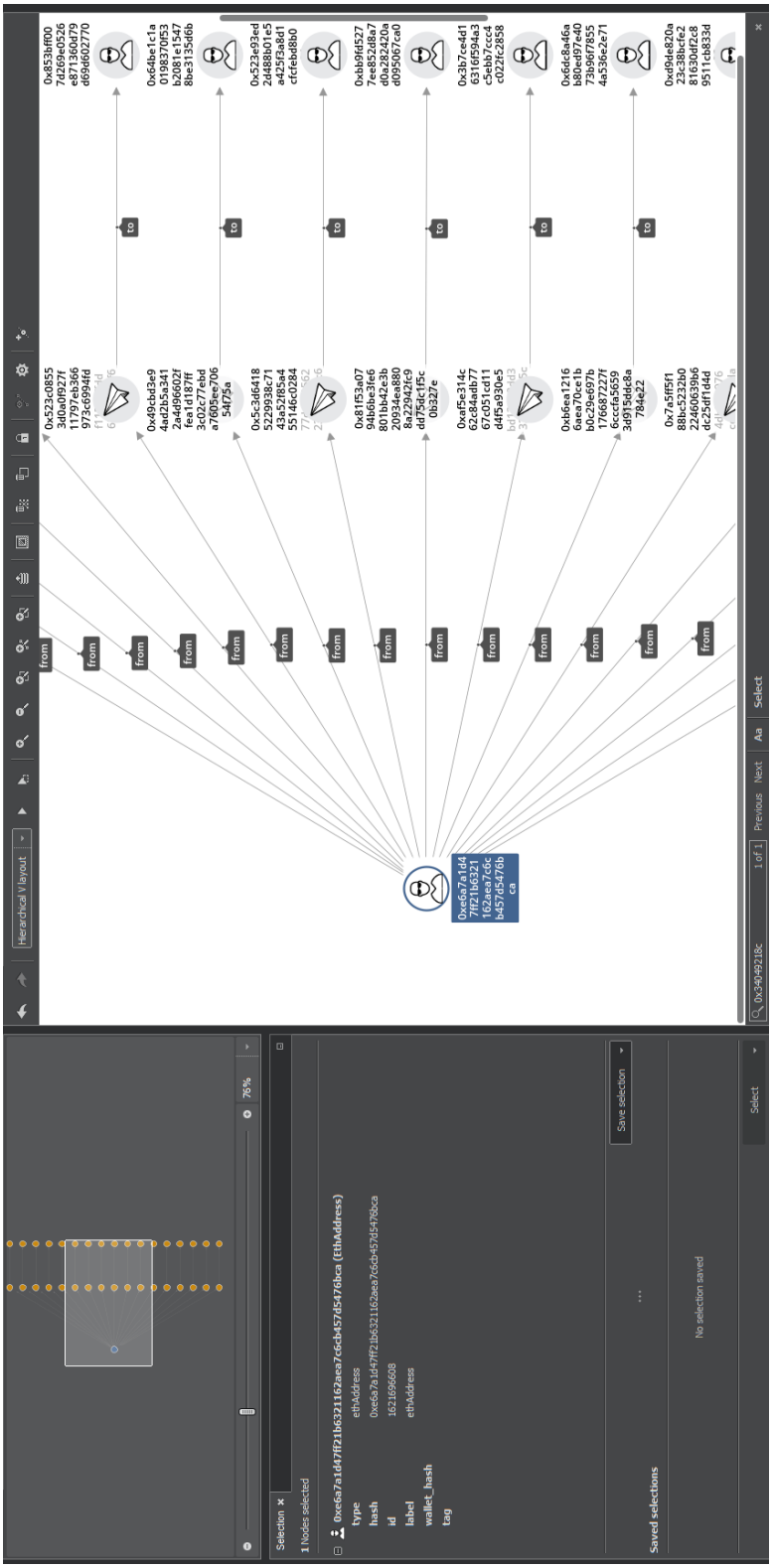

Figure 5.8: ClueMaker Ethereum Block #55875 Sender

Block 89182

This block was mined on November 02, 2010 at 4:10 PM GMT+1 by Unknown. It currently has 645,781 confirmations on the Bitcoin blockchain.

The miner(s) of this block earned a total reward of 50.00000000 BTC ($1,986,103.00). The reward consisted of a base reward of 50.00000000 BTC ($1,986,103.00) with an additional 0. block. The Block rewards, also known as the Coinbase reward, were sent to this address.

A total of 606.05000000 BTC ($24,073,554.46) were sent in the block with the average transaction being 151.51250000 BTC ($6,018,388.62). Learn more about how blocks work.

Hash	000000000013c2e9b07ef258efdabf7c7c82e8ca8c307cc73e0353f1f69c06b8
Confirmations	645,781
Timestamp	2010-11-02 16:10
Height	89182
Miner	Unknown
Number of Transactions	4
Difficulty	3,091.74
Merkle root	7abadf00d5e61c766228450037501ea6e90223f0568e63e79b0ae8fdde1cf097
Version	0x1
Bits	454,373,987
Weight	6,172 WU
Size	1,543 bytes
Nonce	4,088,585,065
Transaction Volume	606.05000000 BTC
Block Reward	50.00000000 BTC
Fee Reward	0.00000000 BTC

Figure 5.9: Explorer Bitcoin Block #89182

Figure 5.10: Explorer Bitcoin Block #89182 Transactions

Figure 5.11: Explorer Bitcoin Block #89182 Transactions detail

Figure 5.12: Explorer Bitcoin Block #89182 Transactions Input

Figure 5.13: Explorer Bitcoin Block #89182 Transaction Output

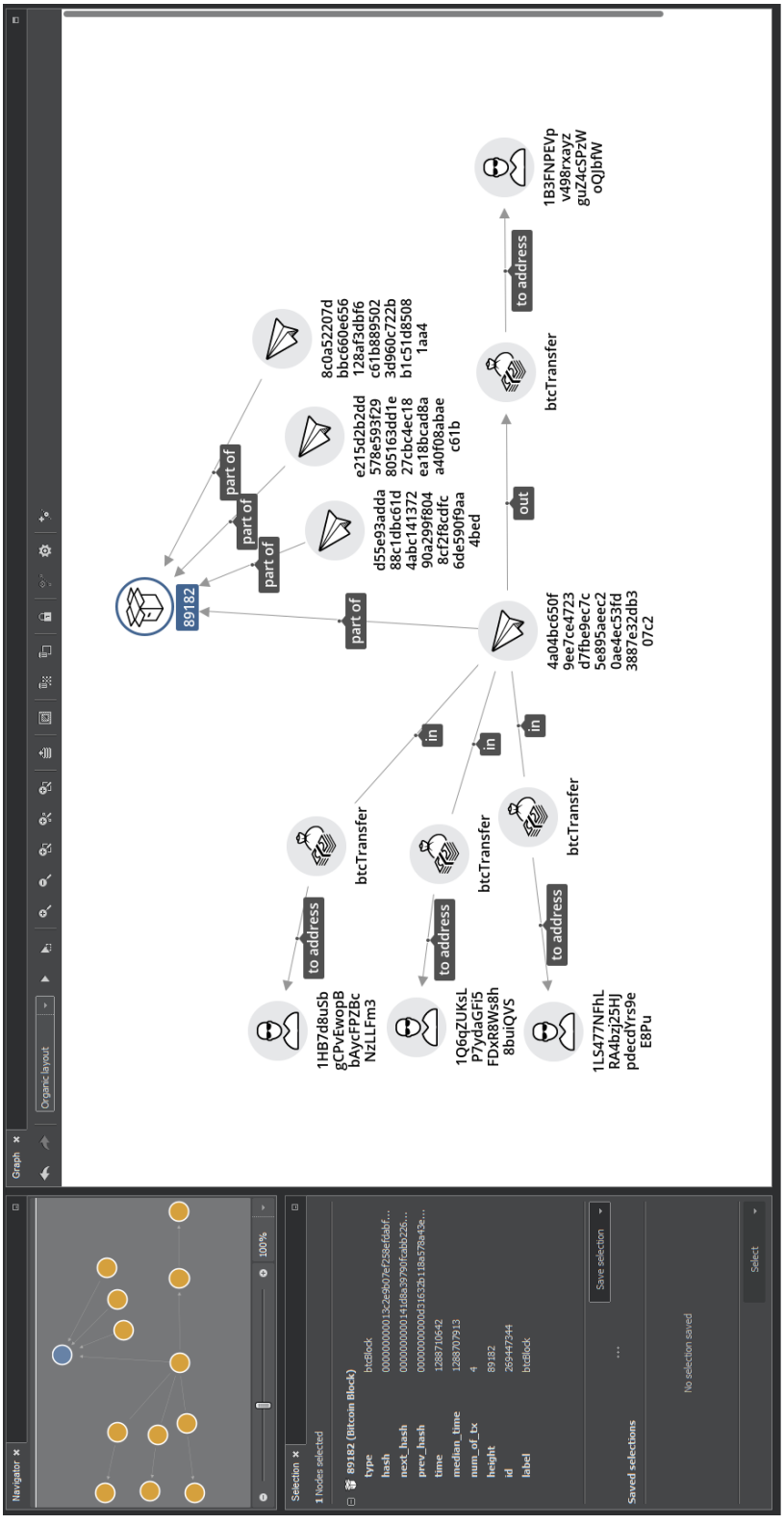

Figure 5.14: ClueMaker Bitcoin Block #89182

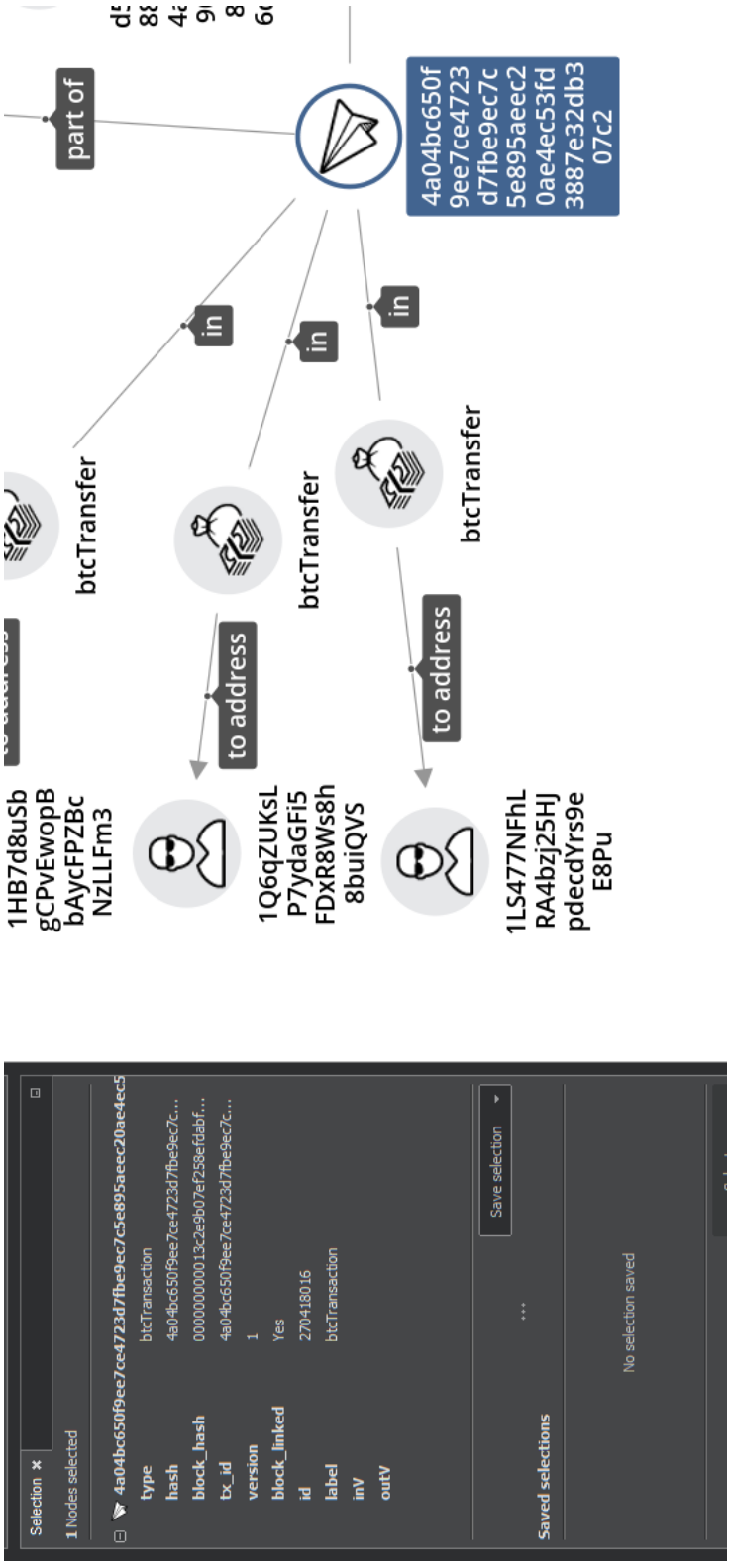

Figure 5.15: ClueMaker Bitcoin Block #89182 Transaction

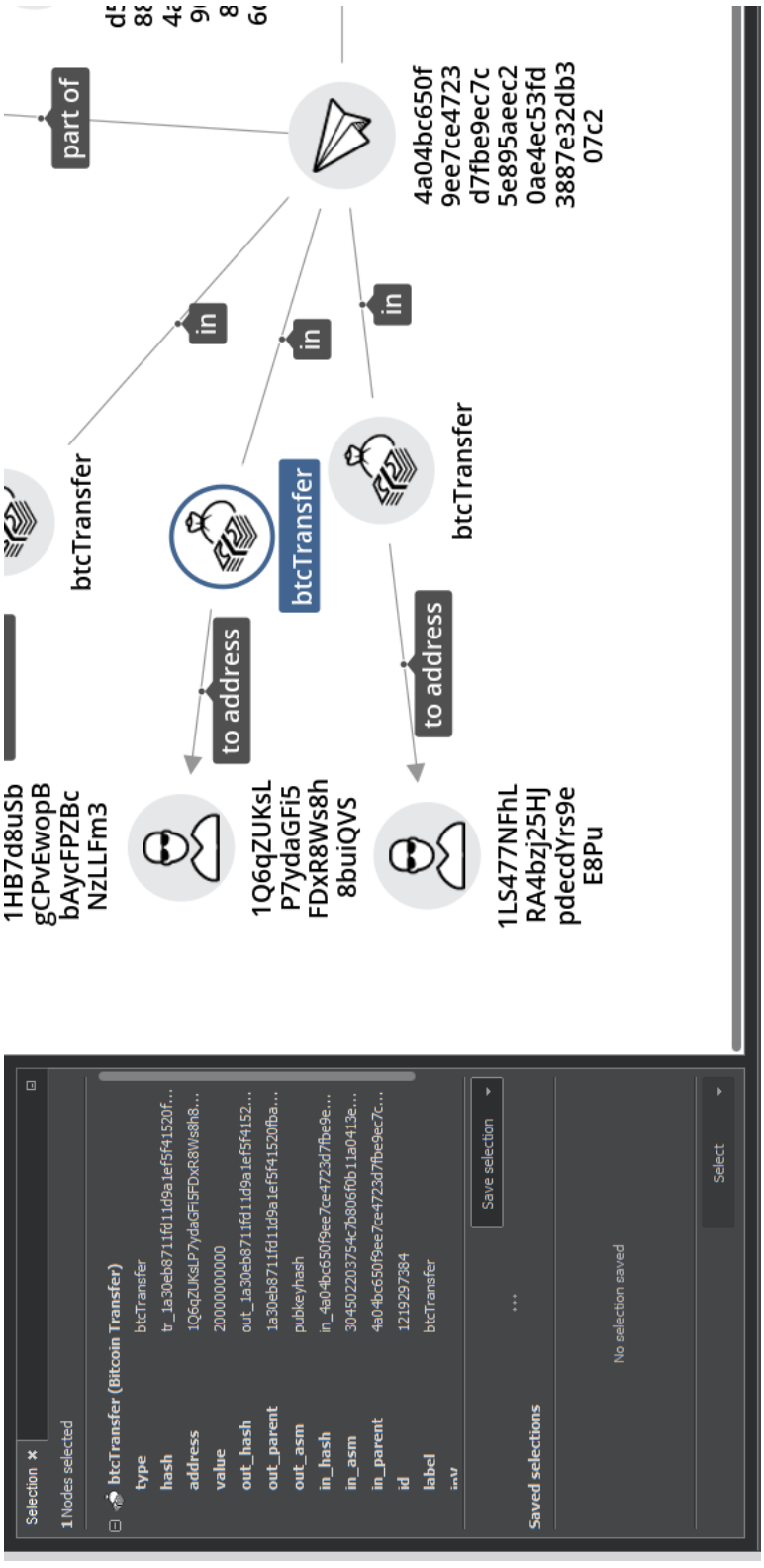

Figure 5.16: ClueMaker Bitcoin Block #89182 incoming Transfer

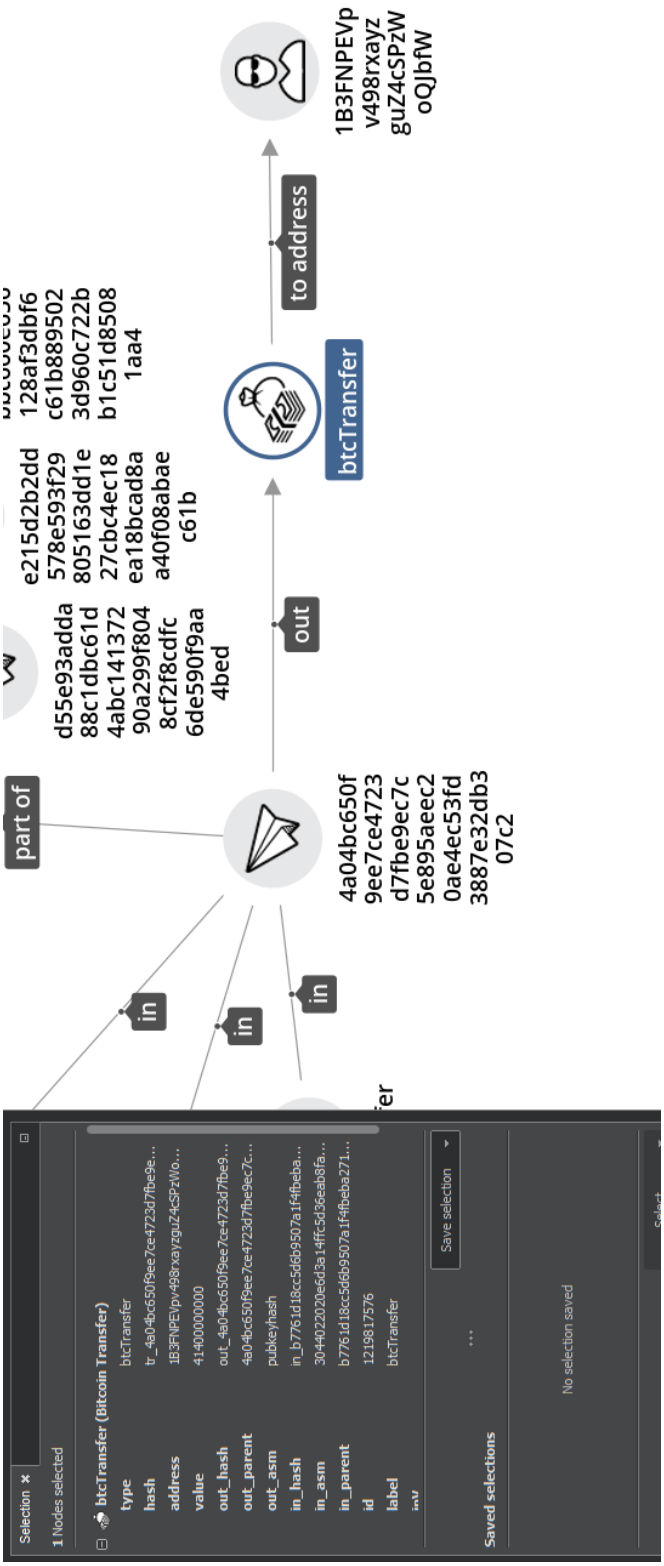

Figure 5.17: ClueMaker Bitcoin Block #89182 outgoing Transfer

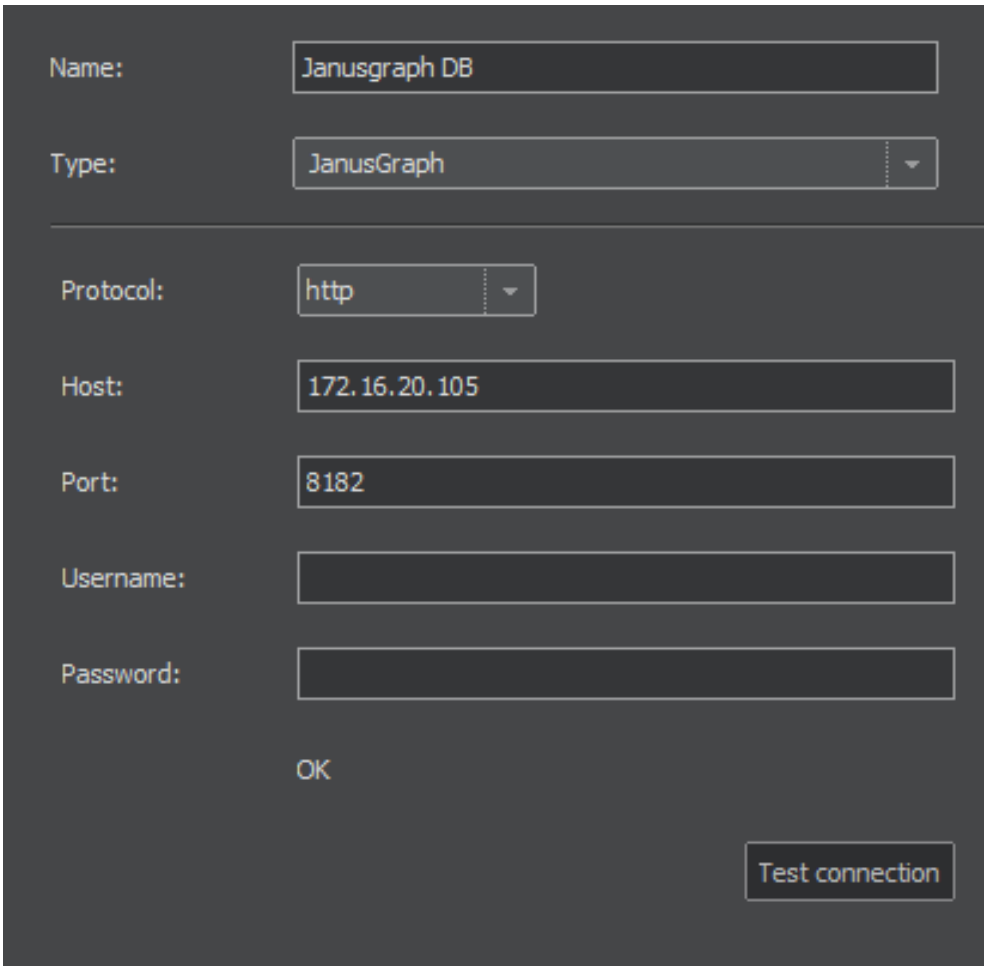

Figure 5.18: ClueMaker Configurator JanusEditorPanel

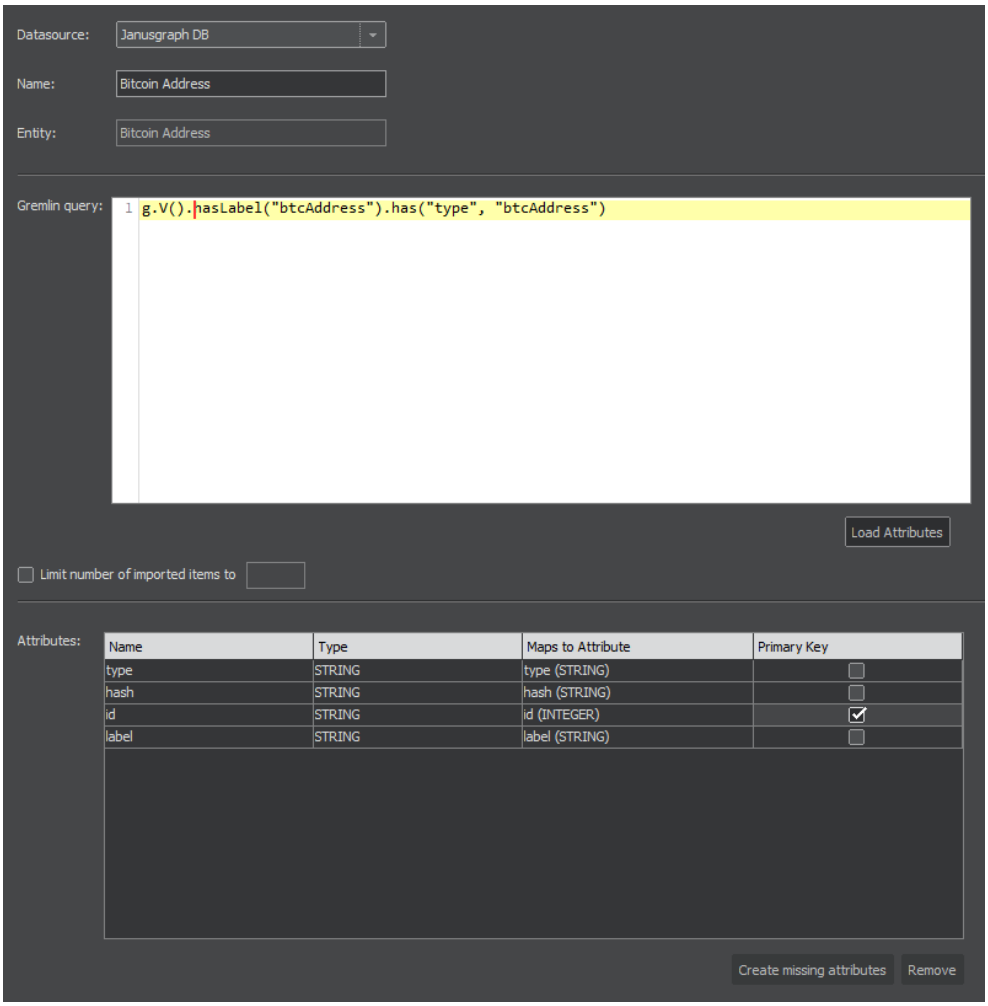

Figure 5.19: ClueMaker Configurator JanusQueryEditorPanel

Conclusion

In this thesis I have described the blockchain technologies together with concrete representatives as Bitcoin and Ethereum. I have analyzed how they function, how they can be retrieved and what data are stored in the clients. I have describe the structures and attributes that are provided by the Bitcoin Client for the Bitcoin Blockchain and Geth for the Ethereum. As part of the analysis, I have contacted developers of similar tool and discussed the abilities of their tool in the fraud investigation for comparison and inspiration.

After collection of all necessary details, I have designed and implemented a application that extracts the data from the blockchain clients, transform them into a structures I have designed and store them in the JanusGraph database together with the relations between the entities in form of edges between the nodes.

Additionally to the blockchain clients, I have searched for additional data sources on public websites and described three of them. The Bitcoinbuse website, Walletexplorer and the SDN list that is published by Office of Foreign Assets Control of the US Department of the Treasury. For these data sources I have implemented data extractors and designed a system of transferring the date from the PostreSQL database to the JanusGraph database.

To explore the data in the graph database, I have implemented a connector to the JanusGraph for the ClueMaker application from the company Profinit EU. In the last part of the thesis, I have tested the final work by comparing the values to public web explorers of the blockchain data and pointed out the advantages of using the ClueMaker tools for visualising the movements on the network.

There are many opportunities for future development of the thesis as the amount of the data can't be stored on one machine and should be distributed to improve the power and speed. Also the stored data hold more information that can be seen at the first sight.

Bibliography

[1] DANNEN, Chris. *Introducing Ethereum and Solidity* [online]. Berkeley, CA: Apress, 2017 [cit. 2022-05-01]. ISBN 978-1-4842-2534-9. Accessible from: doi:10.1007/978-1-4842-2535-6

[2] Panda, Sandeep & Jena, Ajay & Swain, Santosh & Satapathy, Suresh. 2021. *Blockchain Technology: Applications and Challenges.* ISBN 10.1007/978-3-030-69395-4.

[3] Bitcoin address. *WhatIs.com* [online]. [cit. 2022-05-01]. Accessible from: `https://www.techtarget.com/whatis/definition/Bitcoin-address`

[4] ANTONOPOULOS, A. M. (2015). *Mastering Bitcoin: Unlocking digital crypto-currencies.* (Online access: O'Reilly Media, Inc. O'Reilly Online Learning Platform: Academic edition (EZproxy Access).)

[5] ANTONOPOULOS, A. M. Gavin Wood (2015). *Mastering Ethereum Building Smart Contracts and DApps* (Online access: O'Reilly Media, Inc. O'Reilly Online Learning Platform: Academic edition (EZproxy Access).)

[6] Different Types of Crypto Wallets – Explained. *101 blockchain* [online]. [cit. 2022-05-01]. Accessible from: `https://101blockchains.com/types-of-crypto-wallets/`

[7] Top PoS Tokens by Market Capitalization. *CoinMarketCap* [online]. [cit. 2022-05-01]. Accessible from: `https://coinmarketcap.com/view/pos/`

[8] Serpent language. *Serpent* [online]. [cit. 2022-05-01]. Accessible from: `https://eth.wiki/archive/serpent`

[9] Top Storage Tokens by Market Capitalization. *CoinMarketCap* [online]. [cit. 2022-05-01]. Accessible from: `https://coinmarketcap.com/view/storage/`

[10] Tasca, P., & Tessone, C. J. (2019). *A Taxonomy of Blockchain Technologies: Principles of Identification and Classification. Ledger,* Vol 4. https://doi.org/10.5195/ledger.2019.140

[11] NAKAMOTO, Satoshi. *Bitcoin: A peer-to-peer electronic cash system* [online]. 2009. Accessible from: http://www.bitcoin.org/bitcoin.pdf

[12] Transactions. *BitcoinDeveloper* [online]. [cit. 2022-05-01]. Accessible from: https://developer.bitcoin.org/reference/transactions.html

[13] Transaction. *blockchain.com* [online]. [cit. 2022-05-01]. Accessible from: https://www.blockchain.com/btc/tx/581d30e2a73a2db683ac2f15d53590bd0cd72de52555c2722d9d6a78e9fea510

[14] P2MS *learn me a bitcoin* [online]. [cit. 2022-05-01]. Accessible from: https://learnmeabitcoin.com/technical/p2ms

[15] Multi-signature. *Bitcoin Wiki* [online]. [cit. 2022-05-01]. Accessible from: https://en.bitcoin.it/wiki/Multi-signature

[16] Script. *learn me a bitcoin* [online]. [cit. 2022-05-01]. Accessible from: https://learnmeabitcoin.com/technical/script

[17] What is Segregated Witness? *learn me a bitcoin* [online]. [cit. 2022-05-01]. Accessible from: https://learnmeabitcoin.com/faq/segregated-witness

[18] BIP: 141. *GitHub* [online]. [San Francisco]: GitHub, c2020 [cit. 3. 5. 2020]. Accessible from: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki

[19] The Bitcoin Lightning Network. *The Lightning Network* [online]. [cit. 2022-05-01]. Accessible from: https://lightning.network/lightning-network-summary.pdf

[20] Why in old blocks I can't see the miner address using RPC API calls? *StackExchange* [online]. [cit. 2022-05-01]. Accessible from: https://bitcoin.stackexchange.com/questions/110151/why-in-old-blocks-i-cant-see-the-miner-address-using-rpc-api-calls

[21] Address. *learn me a bitcoin* [online]. [cit. 2022-05-01]. Accessible from: https://learnmeabitcoin.com/technical/address

[22] Derivation paths. *learn me a bitcoin* [online]. [cit. 2022-05-01]. Accessible from: https://learnmeabitcoin.com/technical/derivation-paths

[23] Documentation. *Ethereum.* [online]. [cit. 2022-05-01]. Accessible from: https://ethereum.org/en/whitepaper/

[24] EIP-1559. *Ethereum*. [online]. [cit. 2022-05-01]. Accessible from: https://ethereum.org/en/developers/docs/gas/#eip-1559

[25] All You Need to Know About EIP-1559. *ETH Gas Station*. [online]. [cit. 2022-05-01]. Accessible from: https://ethereum.org/en/developers/docs/gas/#eip-1559

[26] Solidity vs Vyper. *Quicknode*. [online]. [cit. 2022-05-01]. Accessible from: https://www.quicknode.com/guides/vyper/solidity-vs-vyper

[27] Solidity Documentation. *Ethereum*. [online]. [cit. 2022-05-01]. Accessible from: https://docs.soliditylang.org/en/v0.8.13/

[28] Frequently Asked Questions. *Uniswap Labs*. [online]. [cit. 2022-05-01]. Accessible from: https://uniswap.org/faq

[29] Dark Forest zkSNARK space warfare. *Dark Forest*. [online]. [cit. 2022-05-01]. Accessible from: https://zkga.me/

[30] NFT. *Ethereum*. [online]. [cit. 2022-05-01]. Accessible from: https://ethereum.org/en/nft/

[31] Infura API. *Infura Inc*. [online]. [cit. 2022-05-01]. Accessible from: https://infura.io/

[32] Requirements. *Bitcoin Project*. [online]. [cit. 2022-05-01]. Accessible from: https://bitcoin.org/en/bitcoin-core/features/requirements

[33] Libbitcoin, A C++ Bitcoin toolkit library for asynchronous apps. *Libbitcoin Project*. [online]. [cit. 2022-05-01]. Accessible from: https://libbitcoin.info/

[34] Client diversity. *Ethereum*. [online]. [cit. 2022-05-01]. Accessible from: https://ethereum.org/en/developers/docs/nodes-and-clients/client-diversity/

[35] Bitcoinový block explorer se sekupováním adres a tagovanými peněženkami. *Walletexplorer*. [online]. [cit. 2022-05-01]. Accessible from: https://www.walletexplorer.com/

[36] Bitcoin Abuse Database. *Bitcoinabuse*. [online]. [cit. 2022-05-01]. Accessible from: https://www.bitcoinabuse.com/

[37] Office of Foreign Assets Control - Sanctions Programs and Information. *US Department of the Treasury*. [online]. [cit. 2022-05-01]. Accessible from: https://home.treasury.gov/policy-issues/office-of-foreign-assets-control-sanctions-programs-and-information

[38] JSON for Beginners – JavaScript Object Notation Explained in Plain English. *Free Code Camp*. [online]. [cit. 2022-05-01]. Accessible from: https://www.freecodecamp.org/news/what-is-json-a-json-file-example/

[39] Remote Procedure Call RPC. *Techtarget*. [online]. [cit. 2022-05-01]. Accessible from: https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC

[40] UNIX domain sockets. *IBM Corporation*. [online]. [cit. 2022-05-01]. Accessible from: https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=considerations-unix-domain-sockets

[41] How to reduce the chances of your Ethereum wallet getting hacked?. *Ethereum StackExchange*. [online]. [cit. 2022-05-01]. Accessible from: https://ethereum.stackexchange.com/questions/3887/how-to-reduce-the-chances-of-your-ethereum-wallet-getting-hacked?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

[42] Installation and User Guide. *Profinit EU s.r.o.*. [online]. [cit. 2022-05-01]. Accessible from: https://docs.cluemaker.com/latest/en/manual/index.html

[43] About ClueMaker. *Profinit EU s.r.o.*. [online]. [cit. 2022-05-01]. Accessible from: https://cluemaker.com/

[44] Chainalysis Inc.: Reactor image. [online]. [cit. 2022-05-01]. Accessible from: https://www.chainalysis.com/wp-content/uploads/2020/03/reactor-page2x-3000x1595.png

[45] About. *Chainalysis Inc.* [online]. [cit. 2022-05-01]. Accessible from: https://www.chainalysis.com

[46] AML Platform. *Coinfirm Limited* [online]. [cit. 2022-05-01]. Accessible from: https://www.coinfirm.com/products/aml-platform/

[47] Insights from Elliptic: The Twitterhack and Bitcoin Money Laundering *Elliptic* [online]. [cit. 2022-05-01]. Accessible from: https://www.elliptic.co/blog/insights-from-elliptic-twitterhack-and-bitcoin-money-laundering

[48] ROBINSON, Ian, James WEBBER a Emil EIFREM. *Graph databases.* Second edition. Beijing: O'Reilly, [2015]. ISBN 9781491930892.

[49] What is difference between Titan and Neo4j graph database? *StackOverflow* [online]. [cit. 2022-05-01]. Accessible from: https:

//stackoverflow.com/questions/45347593/what-is-difference-between-titan-and-neo4j-graph-database

[50] Introduction to Gremlin API in Azure Cosmos DB. *Microsoft , 2022* [online]. [cit. 2022-05-01]. Accessible from: https://docs.microsoft.com/en-us/azure/cosmos-db/graph/graph-introduction

[51] What Is Amazon Neptune?. *Amazon Web Services, Inc.* [online]. [cit. 2022-05-01]. Accessible from: https://docs.aws.amazon.com/neptune/latest/userguide/intro.html

[52] Reduce Traffic. *GITHUB* [online]. [cit. 2022-05-01]. Accessible from: https://github.com/bitcoin/bitcoin/blob/master/doc/reduce-traffic.md

[53] Storage backend. *JanusGraph Authors* [online]. [cit. 2022-05-01]. Accessible from: https://docs.janusgraph.org/storage-backend

[54] Index backend. *JanusGraph Authors* [online]. [cit. 2022-05-01]. Accessible from: https://docs.janusgraph.org/index-backend/

[55] TinkerPop Documentation. *The Apache Software Foundation* [online]. [cit. 2022-05-01]. Accessible from: https://tinkerpop.apache.org/docs/current/reference/

[56] Bulk loading. *JanusGraph Authors* [online]. [cit. 2022-05-01]. Accessible from: https://docs.janusgraph.org/operations/bulk-loading/

[57] Cassandra Documentation. *The Apache Software Foundation* [online]. [cit. 2022-05-01]. Accessible from: https://cassandra.apache.org/doc/latest/

[58] What is bitcoinj? *Bitcoinj* [online]. [cit. 2022-05-01]. Accessible from: https://bitcoinj.org/

Chapter 6

Contents of SD card

- `exe` ..... Directory the executables
- `src` ..... Directory with the source code
  - `ChainAnalyzer` ..... The ChainAnalyer project
  - `ChainanalyzerCrawler` ..... The data extractor scripts
  - `janusgraph-importer` ..... The ClueMaker connector module
  - `janusgraph` ..... The ClueMaker Configurator module
- `text` ..... Folder with the thesis in PDF format