BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

BLOCKCHAIN ABSTRACTION LIBRARY

KNIHOVNA PRO ABSTRAKCI PRÁCE S BLOCKCHAINY KRYPTOMĚN

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. ĽUBOMÍR GALLOVIČ**
AUTOR PRÁCE

SUPERVISOR **Ing. VLADIMÍR VESELÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Gallovič Ľubomír, Bc.**
Program: Informační technologie Obor: Počítačové sítě a komunikace
Název: **Knihovna pro abstrakci práce s blockchainy kryptoměn**
Blockchain Abstraction Library
Kategorie: Počítačové sítě
Zadání:

1. Nastudujte si teorii za nejdůležitějšími kryptoměnami (Bitcoin, Ethereum, Ripple, EOS, Stellar, Cardano) a seznamte se s jejich provozní praxí.
2. Seznamte se s knihovnami pro lokální práci s Bitcoin blockchainem (např. Bitcore-Lib, Insight-API), které poskytují agregování obsahu, jiné indexy nad daty, lokální dotazování skrz API.
3. Dle doporučení vedoucího navrhněte vlastní knihovnu, která by podporovala obdobnou funkcionalitu pro měny, které nejsou založené na Bitcoin projektu (primárně Ethereum) a podobnou knihovnu nenabízí.
4. Implementujte řešení a vytvořte pro práci s knihovnou i RESTové API.
5. Proveďte validační testování, diskutujte možná rozšíření.

Literatura:

- Narayanan, A., Bonneau, J., Felten, E., Miller, A., & Goldfeder, S. (2016). *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press.
- Bitpay, *Guides - Bitcore*, [online] https://bitcore.io/guides, [2018-10-19].

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Veselý Vladimír, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 30. října 2018

Abstract

This thesis deals with cryptocurrencies and the underlying principles they are based on. It describes the blockchain technology and explores various cryptocurrencies that utilize it with the focus on their similarities and differences. The next part analyzes InsightAPI, the existing tool for real-time Bitcoin blockchain exploration. It then describes the proposed solution for the Ethereum blockchain explorer, highlights the implementation details, and shows results of its testing. The created tool allows the user to quickly gather information about desired blocks, users, and transactions of the Ethereum blockchain.

Abstrakt

Táto práca sa zaoberá kryptomenami a princípmi, na ktorých sú založené. Opisuje technológiu blockchain a skúma rôzne kryptomeny, ktoré ju využívajú so zameraním na ich rozdiey a podobnosti. Nasledujúca časť analyzuje existujúci nástroj na získavanie informácií z Bitcoin blockchainu v reálnom čase, InsightAPI. Potom popisuje navrhované riešenie pre nástroj na získavanie informácií z Ethereum blockchainu, opisuje jeho implementačné detaily, a ukazuje výsledky jeho testovania. Vytvorený nástroj umožňuje užívateľovi rýchlo získavať informácie o požadovaných blokoch, užívateľoch a transakciách nachádzajúcich sa v Ethereum blockchaine.

Keywords

Bitcoin, blockchain, cryptocurrencies, Insight API, Bitcore Node, Ethereum

Klíčová slova

Bitcoin, blockchain, kryptomeny, Insight API, Bitcore Node, Ethereum

Reference

GALLOVIČ, Ľubomír. *Blockchain Abstraction Library.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Veselý, Ph.D.

Rozšířený abstrakt

Kryptomeny sa stali v uplynulých rokoch veľmi populárnymi. Medzi dôvody ich popularity patrí anonymita, prístupnosť, rýchlosť prenosu a možnosť obísť existujúce finančné inštitúcie. Kryptomeny sú postavené na technológii nazývanej blockchain, ktorá bola predstavená svetu v roku 2008 a vytvorená osobou alebo skupinou ľudí pod pseudonymom Satoshi Nakamoto. Blockchain ponúka riešenie pre dosiahnutie decentralizovanosti, čo je jeden z najdôležitejších požiadaviek pre funkčnosť online mien. Decentralizovanosť znamená, že neexistuje centrálna inštitúcia, ktorá by zabezpečovala správne fungovanie danej meny. Blockchain funguje ako verejná účtovná kniha, ktorá obsahuje zoznam všetkých transakcií ktoré v danej mene prebehli. Pomocou týchto záznamov je možné zistiť zostatok na účte jednotlivých užívateľov, a takisto sledovať vybavenie vytvorenej transakcie. Samozrejme, technológia blockchain obsahuje mechanizmy ktoré znemožňujú modifikáciu existujúcich dát. Originálna mena, ktorá vznikla nasadením predstavenej technológie sa nazýva Bitcoin. Dnes však existuje veľké množstvo kryptomien ktoré spĺňajú rôzne účely a rozdielnym spôsobom modifikujú pôvodnú technológiu. Jedna z najpopulárnejších kryptomien sa nazýva Ethereum, a jej unikátnou vlastnosťou je možnosť behu akéhokoľvek užívateľom definovaného kódu priamo v blockchaine. Táto vlastnosť dáva užívateľom nové možnosti ako vytvorenie vlastnej digitálnej meny, definícia kontraktov, alebo umožnenie priebehu transakcie len po splnení definovanej podmienky. Použitie technológie blockchain síce zabezpečuje validitu vykonaných operácií, avšak užívateľské možnosti čítania jeho histórie sú obmedzené a časovo náročné. Tieto informácie môžu byť použité k monitorovaniu užívateľskej aktivity, detekcii podozrivých operácií a odhaleniu krádeží a finančných podvodov. Predmetom tejto diplomovej práce je vytvorenie nástroja, ktorý by umožňoval jednoduchý prístup k týmto informáciám so zameraním na rýchlosť ich získania a veľkú množinu typov sledovaných informácií. Komunikácia s vytvoreným nástrojom prebieha cez REST API, vďaka čomu je nezávislá na platforme a je možné ju vykonávať aj so vzdialeného serveru. Aplikácia získa dáta priamo z blockchainu, spracuje ich, a zapíše ich do vlastnej databázy, ktorá má vyššiu prístupnosť obsiahnutých informácií. Program je implementovaný v jazyku Python 3, a použitá je key-value databáza RocksDB. Tento typ databázy ponúka dobrú škálovateľnosť a vysokú rýchlosť získavania požadovaných informácií. Nepodporuje však jazyk SQL, teda vnútorná štruktúra dát je založená na pevne danej množine operácií, ktoré program poskytuje. Beh programu sa dá rozdeliť na dve fázy. Prvou fázou je proces, v ktorom sú dáta získavané z blockchainu, a následne sú spracované a pridané do vytváranej databázy. Táto fáza trvá pokým nie je spracovaný celý blockchain. V závislosti od systémových zdrojov a typov požadovaných informácií môže tento proces trvať aj niekoľko dní. Jeho vykonanie je však nutné podstúpiť iba raz. Druhou fázou je otvorenie komunikačného kanálu a poskytovanie informácií podľa užívateľovych požiadaviek. Zároveň sa však v pozadí inkrementálne pridávajú nové dáta z blockchainu, aby poskytované informácie boli vždy aktuálne. Aplikácia ponúka možnosti vyhľadávať informácie o blokoch, transakciách, a adresách, ktoré tieto transakcie vykonávajú. Takisto má možnosť sledovať vnútorné transakcie, ktoré sa v blockchaine štandardne nenachádzajú. V blockchaine sa tiež nachádzajú zameniteľné a nezameniteľné digitálne meny definované štandardmi ERC-20 a ERC-721. Aplikácia má schopnosť zaznamenať informácie o týchto menách, rovnako ako aj o ich transakciách. Sledované transakcie je možné v rámci požiadaviek vyfiltrovať podľa časového rozmedzia, alebo množstva preneseného kapitálu. Testovaný bol ako synchronizačný proces databázy, tak aj validita poskytnutých informácií a rýchlosť ich získavania. Synchronizačný proces je stabilný, avšak relatívne časovo náročný,

čo je spôsobené predovšetkým pomalým získavaním dát z blockchainu. Tieto funkcie boli implementované tvorcami meny Ethereum, a teda existujú len veľmi obmedzené možnosti ich zefektívnenia. Na overenie validity bola vytvorená sada testov, ktorá porovnáva zozbierané informácie s informáciami získanými z online nástrojov na prezeranie Ethereum blockchainu. Meranie rýchlosti databázy ukázalo, že jednoduché požiadavky sú vybavené za niekoľko desatín sekundy, a vykonávanie požiadaviek pre stotisíce až milióny záznamov trvá desiatky sekúnd až jednotky minút. Vytvorený program poskytuje obšírnejšie a detailnejšie dáta ako existujúce nástroje, a dosahuje vyššiu rýchlosť jednotlivých vyhľadávaní. Hromadné získavanie dát nie je poskytované žiadnym z týchto nástrojov, teda sa jedná o unikátnu funkciu vytvoreného programu.

Blockchain Abstraction Library

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Mr. Ing. Vladimír Veselý Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Ľubomír Gallovič
May 20, 2019

Acknowledgements

I would like to thank Mr. Ing. Vladimír Veselý, Ph.D for his help and guidance in the making of this master's thesis. I would also like to thank my family for their continuous support and for giving me the opportunity to study at Brno University of Technology.

I have decided to share a food recipe with any lost soul that somehow managed to stumble upon this text. I like this recipe because it is quick, simple, needs few ingredients, can be prepared in one pot, and almost resembles a balanced meal. We need the following ingredients: two sausages, a can of diced tomatoes, a can of beans, pesto, garlic, and some seasoning. First, we cut the sausages into smaller pieces and drain the beans. We add some oil into the pan, heat it up, and fry the sausages for a few minutes. Then we add the beans, tomatoes, 1-2 teaspoons of pesto, and a few crushed cloves of garlic. We stir it, bring it to a boil, and let it simmer until most of the water evaporates (stirring occasionally). We season it with salt and pepper. The recipe is very versatile, as we can skip/switch out some of the ingredients and change the seasoning based on our personal preference.

Contents

Chapter 1

Introduction

The goal of this thesis is to create a tool that allows the user to quickly gather the desired information that is part of the Ethereum blockchain. This includes simple operations like retrieving specific blocks or transactions, as well as more complex queries like data from a specified DateTime range or all transactions of a user. The queries will be performed with the REST API interface. Since the purpose of the blockchain implementation is to ensure security and accountability instead of speed, it is not well suited for quickly retrieving the information and does not explicitly track certain kinds of data relation, like the transactional history of a specific user. It also does not offer a RESTful API interface. The tool creates own database from the blockchain node that is better suited for information retrieval, and offers the user an interface containing queries for various use cases.

The diploma thesis is divided into six chapters. Chapter 2 describes the theory behind the cryptocurrencies and the blockchain technology, the basic structures, and principles used to ensure the security and verifiability of decentralized digital currency. It then explores the specifics of various cryptocurrencies and highlights the differences and common themes between them.

Chapter 3 analyzes Insight API, which is an existing tool for querying information about the Bitcoin blockchain using REST API. The chapter explores the query types the library offers, as well as its implementation details in order to ensure comparable functionality and performance in the developed Ethereum blockchain explorer.

Chapter 4 defines the proposed design of the tool created this thesis. It describes endpoints, applications, libraries and database that are used in its implementation. It also proposes the data structures which allow fast querying of the relevant information, while trying to achieve a low level of redundancy.

Chapter 5 describes the implementation of this project, issues that were encountered, as well as their solutions. It also describes the extensions to the original design and the application's interface. Chapter 6 focuses on testing on the created application. It examines the viability of the synchronization process, validity of the returned data, and comparison with the existing tools.

Finally, chapter 7 summarizes the work performed in this diploma thesis, highlights the successes of the created application and mentions possible improvements. It also argues about the usability of the used approaches in the future works.

Chapter 2

Blockchain and cryptocurrencies

This chapter describes the most important principles of cryptocurrencies and blockchain technologies, how they work, and what is their purpose. Then it explores differences between various cryptocurrencies and their ways of storing relevant data.

2.1 Blockchain

The blockchain is the underlying technology behind all cryptocurrencies. It was created by a person/group of people going by pseudonym Satoshi Nakamoto, with the goal of creating a public ledger system that is not governed by a single centralized authority, while still being trustworthy and resistant to tampering. The original paper describing the blockchain was published in October 2008 [31] and is the basis for the first cryptocurrency, Bitcoin. As the name suggests, the ledger is comprised of a series of blocks, all of which are chained together in one continuous line. New blocks, containing recent transactions, are being periodically created, and put at the end of the blockchain, continuously extending it.

2.2 Requirements

The main characteristics a blockchain aims to satisfy the following attributes[38]:

- Decentralization - it means, that unlike classic currencies, each transaction has to be validated without the presence of a central trusted agency (e.g., central bank). The purpose of this is that cryptocurrencies do not have to be reliant on a single point of authority. The authority itself can perform nefarious actions, cease its existence, or fall victim to government tampering. This can compromise the integrity of the currency, or completely stop the possibility of its usage.
- Persistency - it means, that all the transactions will have to be validated, and stored permanently within the ledger records, without the possibility of removal or modification. This creates the requirement that the validity of cryptocurrency blockchain has to be internally provable by any user, despite its decentralized nature.
- Anonymity - it means that each user can interact with the blockchain without revealing their identity, yet still have the means of verifying themselves within transactions.
- Auditability - it means, that transaction history, and user's account balance has to be easily trackable to identify and prevent fraudulent transactions and double spending.

2.3 Implementation

2.3.1 Block

One of the most important components of the blockchain system is called a block. If blockchain represents a ledger of transaction history, then block represents one page of that ledger. Any new transactions that have happened are legitimized by being included within the block and subsequently appended to the end of the blockchain. Other than processed transactions, every block also has a header, which contains information used for chaining it together with other blocks, and verifying block's legitimacy [3]. The very first block in a blockchain is referred to as a Genesis block. New blocks are created periodically at roughly the same intervals. These intervals differ between various cryptocurrencies, but they typically range from a few seconds to a few minutes.

2.3.2 Node

An important requirement of a cryptocurrency is its decentralized nature. This means, that the ledger (blockchain) is not located at one central, trusted place, but every potential user has to have own copy. Independent copies of the blockchain are referred to as nodes. Of course, a user has the potential to tamper with their copy of the ledger, and they also cannot be sure that the copy of ledger they received came from a trustworthy source. This means that every user must have the ability to independently verify the integrity of the ledger without relying on anyone else's authority. When multiple nodes have the same blocks in their blockchain, they are considered to be in consensus.

2.3.3 Proof-of-work

As suggested in the previous section, there has to exist a mechanism that can internally verify that the blockchain is legitimate. The most widely used version of this mechanism is called Proof-of-work. Proof of work is a piece of data, that is difficult to create but easy to verify, and which satisfies certain requirements [14]. In the case of cryptocurrencies, the most widely used proof-of-work is an attempt to get the desired value of SHA-256 hash function. A hash function is a type of function that takes an input of any length and generates an output of fixed length (in the case of SHA-256 it is 256 bits). For the same input, the output is always the same as well. However, it is not possible to do the reverse action of calculating input back from the output. Also, any small change in the input will result in a completely different output. This means, that the only way to generate the desired output is to keep altering the input, and running the hashing function until it generates the correct output. In Bitcoin, the goal is to generate a hash of the block header containing a certain number of leading zeroes. The desired number of zeroes changes over time, with more zeroes requiring more computational work to solve. This is done to compensate for the increasing number of people trying to solve the hash. The blockchain header contains a field called nonce, which is used to store an arbitrary number, generated to modify the header to satisfy the leading zeroes hash requirement [1].

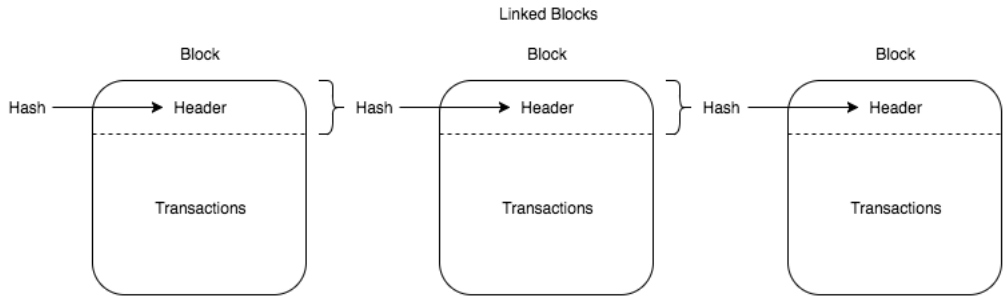

Figure 2.1: Simplified diagram of blockchain, taken from [4]

2.3.4 Mining

The process of trying to solve the proof-of-work is called mining, and people doing the mining are referred to as miners [1]. To incentivize people to do the proof-of-work, a reward of some units of cryptocurrency is given to the first person who figures out the correct nonce value. This value is generally the same for every block, and in some cryptocurrencies, it diminishes over time creating a theoretical ceiling for the number of units in circulation. As every block contains a number of transactions, mining is the process by which they are legitimized and become a part of the blockchain. While generating the correct nonce value is a computationally difficult process, its verification is trivial by only running the hash function and validating the output. When a miner figures out a correct nonce value, they broadcast the block to every other node. The nodes verify the correctness of the block and its hash and subsequently add it to the end of their blockchain. Every block header contains the hash of a previous block header, which is why these blocks are considered to be in a 'chain'. This is done so that it will be virtually impossible to change the history of the ledger. For every change in the block, a new nonce has to be computed to satisfy the proof of work, and since the new hash will be different, the subsequent block will have to be recomputed as well and so on. This means, that after a block has a certain number of successors (in Bitcoin it is 6), it becomes computationally nearly impossible to catch up to the ever-increasing canonical chain of blocks with the altered one.

It is possible for different miners to compute the proof-of-work solution simultaneously, thus creating two different valid blocks, which contain different transactions. When nodes receive both of these valid blocks, they fork the blockchain, and whichever chain is longer is considered to be the valid one [1]. When two chains of the same length are present, whichever one was received first is considered to be the valid one. Due to the possibility of blockchain forking, the block is not yet considered valid when it is simply added to the blockchain. Of the multiple sub-chains created by the fork, only one will be eventually considered part of the main chain. The blocks contained in the abandoned sub-chains are called orphaned blocks, and the transactions within them are not considered valid. Thus, even when user's transaction has been added to the blockchain, they need to wait until their block has at least 6 (in case of Bitcoin) following blocks to be certain that their transaction has truly been legitimized.

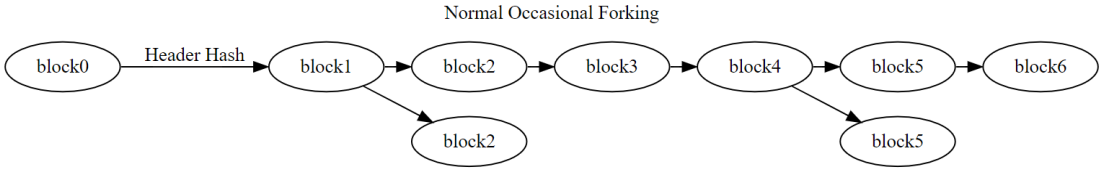

Figure 2.2: Blockchain forking, taken from [1]

2.3.5 Transactions

The main purpose of any currency is to allow the transfer of units of this currency between its users. The difference between classic physical currencies and blockchain based cryptocurrencies is how the capital is represented. In classic currencies, the person is considered to be in possession of a certain number of units of this currency if they have ownership of physical representations of these units (i.e., coins or banknotes). As such, these physical representations can be unofficially and fraudulently created, meaning that the main security concern is to create objects that are difficult to falsify. If there is a desire to add more units to the circulation, this decision is made by the central authority, whose responsibility is to also distribute these new units. If the decision is done poorly, it could cause hyperinflation and economic collapse. On the other hand, cryptocurrencies do not have physical representation of their capital. The only way to prove that a person is in ownership of the currency's units is to find the relevant transaction in blockchain history. This transaction shows that the units have been transferred from another user and the owner has a way to verify that they truly are the recipient of this transaction. Also, new units are created at a predictable rate, as a reward for completing the proof of work.

Various cryptocurrencies have different ways of storing and representing transactions. In general, each transaction contains information like sender's ID, the amount transferred, condition for accessing the transferred units, and transaction fee for the miner [20]. To ensure the validity of the transaction, the sender's current balance is calculated by checking their previous transaction history. The transferred currency can be accessed by the recipient only if certain conditions are met. The most basic and the most used condition type is for the recipient to verify their identity. This is usually accomplished with the use of asymmetric cryptography. The recipient of the transaction is represented by their public key, and they can spend the transaction units if they can verify themselves using their private key. First, the recipient generates their unique private and public key pair. Their public key is then hashed and given to the sender as the recipient's address. Sender then creates the condition that anyone, who can prove that they control the private key corresponding to the included public key can spend the output of the transaction. However, many different types of conditions can be defined, which is accomplished with the programming language provided by the cryptocurrency. These conditions are called smart contracts, and despite being relatively unexplored, they have capabilities to create conditional transfers of funds without the use of a middleman [17]. Last but not least, the transaction fee for the miner is generally included in each transaction in order to create a financial incentive to add the transaction to the mined block. Larger fees create a higher incentive to include the transaction inside the block, meaning that they are processed faster, and the sender can set the fee based on the urgency of the transaction. Every cryptocurrency has a set limit of a maximum number of transaction that can be included in one block.

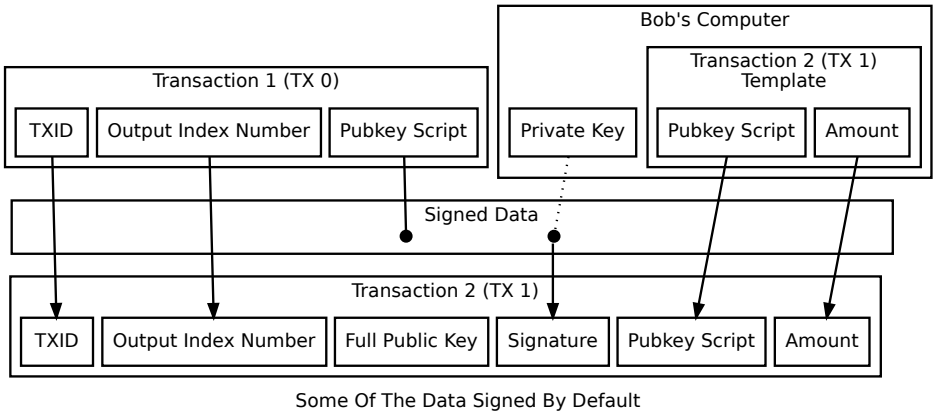

Figure 2.3: Transaction verification illustration, taken from [4]

2.4 Cryptocurrencies specifics

As the goal of this thesis is to create a unified REST API interface that can be used with various cryptocurrencies, it is important to analyze the specifics of the most popular cryptocurrencies and find their differences and common elements.

2.4.1 Bitcoin

Overview

Bitcoin is the oldest and the most popular cryptocurrency today. It has been in operation since 2009 and is the first practical application of blockchain technology created by Satoshi Nakamoto. It has the largest market share, as well as the highest monetary value per Bitcoin, and has 17,000,000 units in circulation at the time of writing in December 2018. Bitcoin has a decreasing mining reward in place, meaning that every 210,000 mined blocks, the reward is halved. This means, that the currency has defined the maximum number of units that can ever be in circulation, specifically an asymptote at 21 million [5]. This could cause a deflation if Bitcoin becomes more popular in the future. One bitcoin is not the base currency unit, as it can be additionally divided. The smallest unit that can be transferred between addresses is called Satoshi, and every Bitcoin contains 100,000,000 Satoshis. Bitcoin blocks limit the number of transactions included in them by setting the maximum block size to 1 Mb. The proof of work difficulty is set so that a new block is generated roughly every ten minutes. As the number of miners changes over time, which causes the proof to be calculated at different rates, the required hash value is continuously modified so that new blocks are generated at consistent speeds.

Blocks

Bitcoin block header contains the following information: block version, previous block header hash, Merkle root hash, the time when miner started hashing the header, target threshold this block's hash must be lower than or equal to, and nonce [3]. Block version number indicates which set of block validation rules have to be followed. New rules can be added over time, and they define a block and transactional restrictions or provide new functionality. For example, adding the requirement of block height parameter in the new

block, or defining a new type of transaction condition that locks the funds until a specified time. Transactions in Bitcoin blocks are contained in a data structure called Merkle tree. In this tree, leaf nodes are labeled with the hash of their data, and non-leaf nodes are labeled with the hash of their child nodes. This hashing is done until only a single node remains, called the Merkle root, which is then included in the block header. As the values of Merkle nodes are dependent on the value of their children, this data structure was chosen to easily verify the validity of a transaction and to prevent tampering. Target threshold of the block header hash defines the number that the final hash has to be lower than. This number is recalculated every 2016 blocks, and the new difficulty is defined based on historical data to satisfy the 10 minutes per block time requirement. The nonce is the field, where miners put an arbitrary number that satisfies the block's proof of work condition.

Transactions

The top-level format of a transaction contains the following fields: transaction version, number of inputs in the transaction, transaction inputs, number of outputs in the transaction, transaction outputs and lock time [20]. Transaction version number works similarly to the block version number, as it defines a set of rules that the transaction has to abide by. To increase security, Bitcoin recommends usage of a new address for every transaction a user performs, meaning that every user generally owns more than one address. For each new transaction, the user generates a new public-private key pair, and their total balance is the sum of balances on all the addresses they have ever used. This creates the requirement for the transaction to include more than one input address. Bitcoin transaction supports more than one output address, and each transaction output contains two fields: the number of satoshis transferred, and the public key script that defines what conditions must be fulfilled for those Satoshis to be spent further. Lock time defines the earliest time or earliest block that the transaction may be added to the blockchain. This adds the functionality to define delayed transactions.

Scripts

Condition defining scripts are called Signature scripts, and they are written in Bitcoin's stack-based scripting language. This language is intentionally not Turing-complete and contains no loops to prevent the creation of malicious code. Due to security concerns, a set of believed to be safe script templates was created, and conformant transactions are called standard transactions [19]. These templates are: Pay to Public Key Hash (P2PKH), Pay to Script Hash (P2SH), Multisig, Pubkey, and Null Data. Transactions that do not use any of these templates are called non-standard and are generally not accepted by the Bitcoin miners. Bitcoin miners have two ways of receiving Bitcoin for their work: reward for solving the hash that generates new Bitcoin, and transaction fees given by the transaction senders. The very first transaction in each block is called a coinbase transaction and contains only the address of the miner claiming their reward. On the other hand, transaction fees are not explicitly given to the miner, they are implicitly defined as the unspent Bitcoin in every transaction. this creates a problem that the unspent Bitcoin is rarely the desired transaction fee of the sender. This is solved by adding another output address to the transaction, that sends the desired portion of the unspent Bitcoin back to the sender.

Nodes

In Bitcoin, there are two main methods for clients to verify the blockchain: Full nodes, and Simplified Payment Verification [33]. The full node is the most secure method. However, it comes with some disadvantages for regular clients. In this form of validation, the client downloads the entire blockchain and independently verifies the hash in every block from the genesis block to the most recent one. The only way to fool the client is to compute a blockchain longer than the valid one, which is computationally nearly impossible. The disadvantage of this method is that the blockchain has a large size and the initial synchronization can take even a few weeks, based on client's hardware. The second method is Simplified Payment Verification, which only downloads and verifies the block headers and requests transactions from full nodes as needed. SPV client can prove that transaction is included in the block with only Merkle root, and desired transaction's Merkle branch. However, full nodes providing the information to the client can lie by stating that the transaction was not included in any block, which SPV has no way to verify. Additionally, since clients ask the full nodes only for specific transactions, this allows the node to construct the identity of the user, creating a privacy issue.

2.4.2 Ethereum

Overview

Ethereum is an important and influential cryptocurrency, that, despite sharing many similarities with Bitcoin has also introduced many new concepts, use cases, and philosophies. It is not a derivative of Bitcoin, meaning that it was created from ground up with the aim of making a more modular, universal, agile, and simpler system. It borrows many ideas first introduced by Bitcoin, like blockchain technology and proof of work to create a trustworthy and tamper-resistant system. It also introduces new concepts that help execute on its altered design philosophy. The starkest difference is its introduction of Turing-complete scripting language that transforms it from a cryptocurrency to a fully programmable virtual machine [34]. This change opens doors for many new uses, like the creation of different token systems, stable-value currencies, identity and reputation systems, decentralized file storage, decentralized autonomous organizations, and many others. Ethereum was launched on 13th July 2015, and its base unit of currency is called ether. Ether can be further divided into smaller units, the smallest of which is called Wai and has the value of $1/10^{18}$ ethers. Aside from being used as a tradable currency, ether can be used as a payment for executing any user-defined code. Another difference to Bitcoin is that Ethereum is inflationary, meaning that it does not have a set value of maximum units in circulation. The reward for mining a block does not get continuously halved over time, so similar amounts of ether are introduced to the system every year. To combat over-inflation, the reward can be decreased, but this decision is always carried out by people responsible for maintaining Ethereum. In September 2019, the reward for mining a block will be decreased by 33% from three ether to two.

Differences to Bitcoin

To increase modularity and usability by third party users, Ethereum has many differences in its implementation compared to Bitcoin. Bitcoin records its world state as a history of all transactions across its entire blockchain, while Ethereum contains a tree structure defining the world state in each new block. On first glance, this appears to be significantly more

taxing in terms of storage requirements and computational difficulty. However, Ethereum uses a different tree structure than Bitcoin called Patricia Merkle trie [37]. This tree allows its branches to be defined as pointers, meaning that only differences that occurred within the block need to be recorded. The unaltered parts are simply defined as references to sub-trees in older blocks. As Ethereum uses a Turing complete language for its scripting, the transactions are more varied and can produce many different outcomes. This means, that each block, in addition to having a tree containing all new transaction definitions also contains a data structure called a receipt tree. This tree contains the outcome of each new transaction, such as post-transaction state, the cost of its computation, and logs created during its execution. There are also differences in how Ethereum handles user accounts. In Bitcoin every address in a transaction is generally only used once, and user's balance is calculated as the sum of all the unspent transaction outputs they are in possession of. On the other hand, Ethereum has introduced the concept of accounts which usually take part in more than one transaction. Two types of accounts exist in Ethereum to differentiate between real people and autonomous contracts. The first type is called an externally owned account and works similarly to an address in Bitcoin. It has a defined public-private key pair and is controlled by an external actor (a.k.a human) to create transactions. The second type is called a contract account, which works autonomously based on its defined logic. It contains a contract code, which activates every time the account receives a message from either an externally owned account or another contract account. Execution of its contract code allows it to read and write to its internal storage, send other messages, or create contracts. As a tool for Ethereum's Turing-complete design philosophy, these accounts allow users to define autonomous systems running independently within the blockchain.

Scripts

Unlike Bitcoin, transactions in Ethereum use a Turing complete language that can be used to run any user-defined script. This opens the door for a new set of issues, mainly the execution of malicious infinitely running code. To prevent a denial of service attacks on the miners running the code, a new transaction fee system called gas was introduced [24]. Gas represents a 'fuel' used to run an instruction in a user-defined script. Every transaction contains two fields defined by the sender: gas price and gas limit. Gas price is used to define an amount of ether a user is willing to pay for one used up gas. This effectively acts as a transaction fee the user offers to pay to the miner for processing their transaction, with higher value acting like a bigger incentive for the miner. The gas limit is used to denote the maximum amount of gas that can be used in the script processing. If all gas is used up before the script is completed, the transaction will be marked as unsuccessful. However, the miner still keeps all the ether used to buy the gas for the processing of the transaction. This means that if a user defines an infinitely running loop in the script, the loop will stop running once all gas is used up, and the user is the one that has to pay for the useless code being run. Before miner initiates a transaction, they check if the user has a higher account balance than gasPrice*gasLimit, and only then starts the computation. Once the calculation is completed, all the unused gas is refunded to the sender.

Mining

Another difference between Ethereum and Bitcoin are their strategies in block processing. In Bitcoin, the amount of transactions included in a block is limited by their physical size, i.e., how many bytes they take up. Ethereum, on the other hand, has a block gas

limit, which defines the maximum amount of computational work that can be done in one block. Unlike Bitcoin, block gas limit is determined by the consensus of the miners and can change over time, which increases Ethereum's flexibility and scalability. Ethereum also has a much smaller average mining time per block, usually being around 10-20 seconds. While individual block is easier to compute, and thus has smaller confidence of the users, the faster build-up of the blockchain means that a block reaches high permanence probability faster than in Bitcoin [28]. A disadvantage of faster hash computations is a higher chance of multiple blocks being created simultaneously, causing more forks in the blockchain. Because this would create a big disadvantage for smaller miners and miner pools, a concept of 'uncle', or 'ommer' blocks was introduced [29]. These blocks are a part of a shorter blockchain fork and are considered non-canonical. However, miners responsible for their creation are still awarded 7/8ths of a reward for mining a block, which gives incentive to smaller miners who have a lower chance of producing canonical blocks. To not waste computational power used to mine these blocks, miners are also incentivized to include hashes of ommers in new block headers. For doing so, they are given a bounty of 1/32 of a block mining reward for every included ommer. This increases security, as a would-be attacker would have to recompute hashes of these uncle blocks as well. Blocks can only include ommers from last seven generations because of the difficulty in calculating the validity of older blocks.

Tokens

As mentioned in a previous section, Ethereum is actually a virtual machine capable of running any user-defined code. This capability has many implications, allowing the users to interact with and build upon the currency in many interesting ways. One of the most popular utilizations is the creation of tokens, or virtual currencies running on top of Ethereum. This empowers the users to create a currency with their own definition of usage, purpose, distribution, and supply. While the API and the underlying implementation can be unique for each currency, there is a motivation to create a set of rules that would standardize the interaction with each virtual token [13]. This would simplify their usability and allow them to interact with each other without much additional implementation. There exist two widespread standards: ERC-20 and ERC-721. Each of them defines a set of functions a token must have to be considered compliant. The standards do not conflict with each other, as they define different types of currencies, namely fungible and non-fungible tokens. New tokens are created with contract creation transactions, and any interaction with the token is made through its contract.

The first standard, ERC-20 [8], describes the requirements for fungible tokens. This means that the units of the currency are interchangeable, having no unique properties to them. ERC-20 tokens function much like real money, or the Ethereum itself. Any number of units can be transferred between the users, provided they have enough in their possession. ERC-20 standard defines the rules to create transactions, approve them, retrieve the balance of a user, and find out if the user can perform a transaction. Additionally, each token has a defined total supply of units in circulation. There are three optional properties of ERC-20 tokens. The first one is divisibility, which defines the smallest fraction of units that can be transferred. It functions similarly to cents in real-world currencies. The other two are name and symbol, which are self-explanatory. While ERC-20 suffers from a potential double spending vulnerability, it still remains the most widely adopted fungible token standard in Ethereum. Replacement standards such as ERC-223 and ERC-777 have been created, but they have a significantly smaller rate of adoption.

ERC-721 is a standard used to define non-fungible tokens. Each token has a unique ID, and is meant to represent a specific non-replaceable object. There are multiple use-cases for this kind of currency. It is most commonly used to track and represent virtual collectible items, with each of them having unique properties. The most well known example is called CryptoKitties, where the users can purchase their own virtual cat with its own unique appearance. In the future, ERC-721 could also be used to represent real world objects, such as houses. It would act as both a record keeping book, allowing the users to look for an owner or the transfer history, as well as a platform for selling and exchanging the real world objects. However, it is still debatable if the cryptocurrencies ever garner enough trust and attention for this to become a reality.

2.4.3 Ripple

Currently holding the second rank in market capitalization is the digital currency of a company Ripple, XRP [27]. Despite being frequently grouped with other cryptocurrencies, XRP's philosophy, usage, and implementation is so different, that it would be better described as digital fiat currency. First of all, it breaks the decentralization principle as it is managed by its creator company Ripple, which is also in possession of the majority of its capital. XRP ledger is public, which provides some level of transparency despite the fact that regular users are not allowed to participate in its creation. XRP does not use blockchain technology. Instead, it utilizes the Ripple protocol consensus algorithm (RPCA) to ensure the currency's trustworthiness. The algorithm creates consensus from multiple unaffiliated parties, all of which have differing interests, and thus are unlikely to participate in joint nefarious actions. Centralized system has its own advantages, specifically higher processing speed and determinism, as a new XRP block is created exactly every four seconds.

XRP is primarily not a cryptocurrency. Instead, it is intended to serve as 'currency of last resort' in Ripple's global transaction network, RippleNet. RippleNet is a system that globally connects banks and payment providers with the goal of creating a protocol for transactions and currency exchanges between world's various financial institutions. It aims to replace current, outdated systems (such as SWIFT), with a new, much faster and less costly system. As the world's various currencies and other objects of value (like gold) can be traded using the network, XRP was created to simplify the conversions between them [26]. In classic transaction systems, each conversion has a fee, and if no institution provides a direct conversion, a chain of intermediary conversions have to be created that further drive up the costs. XRP aims to act as a 'base currency' that all other currencies can be converted into, and additionally offers very small conversion fees, making the process cheaper and less complicated. However, XRP is currently not used in Ripple's most widely adopted product, XCurrent. It only exists in their other product, XRapid, and the question remains if XRapid will ever reach the same level of adoption as XCurrent. Another XRP's issue stemming from its centralized nature is its possibility of being shut down by government legislature, essentially losing all its value overnight.

2.4.4 Bitcoin derivatives

Differing philosophies and implementation concerns of Bitcoin have spawned numerous new cryptocurrencies that are largely based on the same protocol and use many of the same principles. These include both completely new currencies started from scratch and hard Bitcoin forks that contain all the old transactions that occurred before the fork.

Currently the most popular of these derived currencies is called Litecoin [25], which has launched the 7th of October 2011. It was created by former Google engineer Charles Lee and the motivation behind it was to increase the transaction processing speed and improve decentralization. New block creation time was reduced from 10 minutes to 2.5 minutes, and a new hashing algorithm called Scrypt was introduced. Original hashing algorithm SHA-256 used in Bitcoin allows for a higher degree of parallel processing, meaning that larger miners using specialized ASIC hardware have much better chances of solving the hash before the smaller miners. Scrypt, on the other hand, is a 'memory hard problem', so the hashing speed receives smaller gains from using specialized hardware, which creates a more inviting environment for smaller miners.

Another popular cryptocurrency is called Bitcoin Cash [30] and was created as a response to a disagreement about the future development of Bitcoin. Bitcoin has one of the lowest transaction processing speeds of all the cryptocurrencies, with only seven transactions per second being processed. Bitcoin Cash was created as a response to concerns regarding Bitcoin's scalability, with the aim of increasing the speed of the transactions. It was created on 1st of August 2017 as a hard fork of Bitcoin, and its biggest change was the increase of the block size from 1 MB to 8 MB. While it can support significantly more transactions per block, security concerns were raised about this alteration.

Another, relatively new cryptocurrency is called Bitcoin SV [32] or Satoshi Vision which has quickly climbed the ranks of market capitalization chart after its launch on 15th of November 2018. The philosophy behind it is to maintain the vision of Bitcoin's original creator Satoshi Nakamoto. It was hard forked from Bitcoin cash with the aim of increasing block size even further, to 128 MB.

There is also cryptocurrency called Dash [35], which was created with the motivation of having a government structure, that democratically decides any changes to the currency, making hard forks less likely to happen. It has smaller transaction fees and transaction wait times compared to Bitcoin, but its disadvantage is that miners do not get to keep all of their reward as parts of it go to master nodes and the development team.

Chapter 3

Insight API

This chapter describes an existing tool for exploring the Bitcoin blockchain, Insight API. The goal is to explore and analyze both the functionality and deeper implementation details of Insight API in order to get a better idea of an effective design that would be used in the Ethereum blockchain explorer created for this thesis.

3.1 Introduction

Insight API is a tool whose purpose is to provide REST API endpoints for accessing both simple and aggregated information from Bitcoin's blockchain in real time. It is a part of the Insight project along with the Insight UI tool. Insight was created by the American company BitPay as an open-source Bitcoin blockchain explorer [11] allowing for more advanced Bitcoin queries as well as visual blockchain exploration using a web browser. The project is written in JavaScript, with development being active since 2013. It is not a stand-alone tool, but exists as an additional service add-on for the Bitcore project. As such, it does not manage Bitcoin synchronization and appropriate data structure implementation by itself, instead, it utilizes API methods available from its parent project, Bitcore-node. Bitcore-node is a full Bitcoin node implementation also developed by BitPay, which extends the functionality of the default node, provides new methods, and defines a different data storage system allowing for faster queries, with the cost of higher storage requirements. Since version 5, Bitore supports codebase from various other node implementations, paving the way for a more modular and flexible system.

3.2 Functionality

This section will explore the functionality of Insight API, what endpoints it provides, and what the information given by each of them is. This information will help in the creation of Ethereum blockchain explorer design, and highlight possible differences in the data provided by these cryptocurrencies. Insight API provides the following endpoints [12]:

- `GET /insight-api/block/[:hash]`
- `GET /insight-api/block-index/[:height]`
- `GET /insight-api/rawblock/[:blockHash]`
- `GET /insight-api/rawblock/[:blockHeight]`

- `GET /insight-api/blocks?limit=[:intLimit]&blockDate=[:date]`
- `GET /insight-api/tx/[:txid]`
- `GET /insight-api/addr/[:addr]`
- `GET /insight-api/addr/[:addr][?noTxList=1][&from=&to=]`
- `GET /insight-api/addr/[:addr]/balance`
- `GET /insight-api/addr/[:addr]/totalReceived`
- `GET /insight-api/addr/[:addr]/totalSent`
- `GET /insight-api/addr/[:addr]/unconfirmedBalance`
- `GET /insight-api/addr/[:addr]/utxo`
- `GET /insight-api/addrs/[:addrs]/utxo`
- `POST /insight-api/addrs/utxo`
- `GET /insight-api/txs/?block=[:hash]`
- `GET /insight-api/txs/?address=[:addr]`
- `GET /insight-api/addrs/[:addrs]/txs[?from=&to=]`
- `POST /insight-api/addrs/txs`
- `POST /insight-api/tx/send`
- `GET /insight-api/sync`
- `GET /insight-api/peer`
- `GET /insight-api/status?q=xxx`

While most of these endpoints provide specific, or aggregated information about the various aspects of the blockchain, some endpoints offer meta information about the node itself, or allow the user to interact with the blockchain by creating new transactions. These endpoints are not considered relevant, as the purpose of this thesis is to create a blockchain explorer, not a fully interactive Ethereum node.

Information provided by the API can be sorted into three categories: blocks, addresses, and transactions. Block endpoints allow the user to acquire the information about a block specified by its hash, get block hash based on its index number, receive raw data of a block based on its hash or index, and get summary data of blocks that happened on a specified day with the possibility of limiting the result count. Responses to queries asking for a single block contain all information about the block including its transactions, while multi-block queries are only responded with block summaries omitting certain information, like the transactions specifics.

The second major category of information provided by Insight API is information about transactions, and their aggregations based on various criteria. The most basic query receives information about a transaction based on its ID. It includes summarizing information about the transferred currency, meta information about its position in the blockchain, list of input and output addresses participating in this transaction, and information about the spending of the output addresses. Insight API is also able to aggregate multiple transactions based on the specified block hash or address. Addrs endpoint provides the functionality to get the transaction history of an address, or multiple addresses, with the possibility to specify the date range, as well as request the omission of certain information like script signature and output spending information.

The last category provides the information about Bitcoin addresses, their unspent outputs, and the transactions they were used in. There are multiple endpoints, each of which provides specific data of the address, or fetches all the information at once.

3.3 Implementation specifics

As stated previously, Insight API does not handle saving the relevant blockchain information by itself, instead, it uses the methods provided by Bitcore node, which is responsible for saving and updating the blockchain information [2]. Exploration of Bitcore source code has revealed that a LevelDB database is used to store the information queried by Insight API. LevelDB [36] is a key-value type database developed by Google, which was created as a simpler alternative to a relational model, and offers smaller storage requirements and better speeds for reading data. The rationale behind its usage was likely the existence of only specific use cases in querying the information, allowing for the data to be saved in a way that results in faster responses for specific queries, as well as making the implementation simpler by not having to model the blockchain as an SQL database. For example, blocks are saved with their hash being the key, and the rest of the block information being the value. Of course, this solution also has its disadvantages, like the need to serialize data in order to perform reading and writing operations. This means that data has to be encoded/decoded when performing a query or adding a new record to the database. This is accomplished by concatenating the strings of some attributes of an entity to a unique key, and the rest of the attributes to the value. Another disadvantage is the lack of flexibility, and the inability to create more advanced SQL queries, however, this is not much of a problem as exploring the blockchain does not have that many use cases, and better performance of the predefined queries is more important. The records in the database are sorted by key, which allows for range-based bulk queries to be performed faster with only a single read. This is useful in certain use-cases offered by the Insight API, for example returning all transactions of an address from a specified date range.

Chapter 4

Design

This chapter describes the proposed design of Ethereum blockchain explorer that is to be created as a result of this thesis. The design, functionality, and data saving mechanisms are inspired by the functional solution of Insight API and Bitcore Node, explained in chapter 3. This chapter describes the intended functionality of the created tool, proposed endpoints, the differences to Insight API and the rationale behind them. Data structures holding the blockchain information are also proposed, created with the aim of providing fast responses to given queries while trying to avoid unnecessary redundancy. Programs and libraries intended to be used in the creation of the Ethereum explorer are also mentioned in this chapter.

4.1 Proposed functionality

The goal is to create a tool with functionality similar to Insight API, with respect to differences between Bitcoin and Ethereum principles and implementations described in section 2.4. The available endpoints in insight API can be seen in section 4.1 and can be divided into three categories: fetching blocks, fetching addresses, and fetching transactions. In this section, endpoints for each of the three categories will be proposed, and the possible differences between then and Insight API will be explained. The first set of endpoints will be used for querying information about specific blocks of the Ethereum blockchain. The proposed endpoints are:

- `GET /block/[:blockHash]` - return all information about a block specified by its unique hash
- `GET /block-index/[:height]` - return the hash of a block specified by its index
- `GET /blocks?limit=[:intLimit]blockStart=[:dateTime]&blockEnd=[:dateTime]` - return all information about a number of blocks specified by a date-time range with the possibility of limiting the number of returned results
- `GET /block-indexes?limit=[:intLimit]&indexStart=[:index]&indexEnd=[:index]` - return all information about a number of blocks specified by an index range

Every argument in the endpoints returning information about multiple blocks is optional. The start and end attributes have the implicit value of the first and last block of the blockchain. Endpoints for requesting raw block information were removed, as this information is not provided by the Ethereum nodes, and can be derived algorithmically from structured block information if desired.

The seconds set of endpoints will be used for querying information about past transactions. The proposed endpoints are:

- `GET /tx/[:txhash]` - return all information about a transaction specified by its hash
- `GET /txs/?block=[:hash]` - return information about all transactions of a block specified by the block's hash
- `GET /txs-index/?block-index=[:blockIndex]` - return information about all transactions of a block specified by the block's index
- `GET /txs-addr/?address=[:addr][?time_from=&time_to=&val_from=&val_to=&no_tx_list=]` - return all normal transactions of an address
- `GET /addrs/[:addrs]/txs[?time_from=&time_to=&val_from=&val_to=&no_tx_list=]` - return all normal transactions of multiple addresses
- `GET /int-txs-addr/?address=[:addr][?time_from=&time_to=&val_from=&val_to=&no_tx_list=]` - return all internal transactions of an address
- `GET /addrs/[:addrs]/int-txs[?time_from=&time_to=&val_from=&val_to=&no_tx_list=]` - return all internal transactions of multiple addresses
- `GET /token-txs-addr/?address=[:addr][?time_from=&time_to=&no_tx_list=]` - return all token transactions of an address
- `GET /addrs/[:addrs]/token-txs[?time_from=&time_to=&no_tx_list=]` - return all token transactions of multiple addresses

Endpoints, which gather various types of transactions of an address have the option of filtering them out by the optional parameters. The user can specify a date-time range of their occurrence, as well as the minimum and maximum amount of transferred units. Maximum limit of returned transactions can also be specified. All of these parameters are optional, and no restrictions are applied in case of their omission. Token transactions cannot be filtered by the transferred amount, since ERC-20 tokens use different decimal systems, and ERC-721 tokens are non-fungible.

The third set of endpoints will access addresses of both external accounts and contracts, their current state and their transaction history. The proposed endpoints are:

- `GET /addr/[:addr][?time_from=&time_to=&val_from=&val_to=&no_tx_list=]` - return all information about an address
- `GET /addr/[:addr]/balance` - return a balance of an address
- `GET /addrs/[:addrs][?time_from=&time_to=&val_from=&val_to=no_tx_list=]` - return information about multiple addresses

Address data contains the information like its balance, type, associated code and the list of transactions it was involved in. There is a possibility of filtering out some of the address's transactions by utilizing the provided optional parameters. The transactions can be filtered by the time of their occurrence or transferred value. Certain endpoints from Insight API were omitted, like the return of address's unconfirmed balance, as the explorer only works with confirmed, a permanent state of the blockchain. Other endpoints which were not included return unspent transaction outputs of an address, as this concept is specific only to Bitcoin, and is not a part of Ethereum architecture.

As the functionality of the created tool is somewhat extended compared to its original design, one more endpoint was added to accommodate these new features. This functionality involves tracking ERC-20 and ERC-721 tokens, and token transfers. The following endpoint was created:

- `GET /token/[:addr][?time_from=&time_to=&no_tx_list=]` - return all information about a token

This endpoint returns the data of a token, specifically its name, symbol, type, decimals, and total supply. The query also returns all transactions of this token. As this could be a very high number, the optional parameters can be used to filter them by time or limit the maximum number of returned results. The unique identifier of a token was chosen to be the address of its contract, as every token is associated with a contract address. The reason why a more user-friendly identifier like a name or a symbol was not chosen is that these fields can have duplicate occurrences. It should be noted that there is no endpoint used for requesting only token transfers. The reason is that these transfers do not contain a unique identifier and thus cannot be searched for by using the API.

4.2 Tracked information

This section describes what information is tracked and returned in each of the data structures in the Ethereum blockchain explorer. Queries for an Ethereum block will return the following information:

- `number` - a sequential index of the block
- `hash` - hash of the block
- `parentHash` - hash of the parent block
- `nonce` - hash of the generated proof-of-work
- `logsBloom` - bloom filter for the logs of the block
- `miner` - the address of the miner who receives the mining rewards
- `difficulty` - integer of the difficulty of this block
- `totalDifficulty` - integer of the total difficulty of the chain until this block
- `extraData` - extra data field of this block
- `size` - the size of the block in bytes
- `gasLimit` - maximum gas allowed in this block
- `gasUsed` - total used gas of all transactions in this block
- `timestamp` - unix timestamp of when the block was generated
- `sha3Uncles` - combined hash of parent block's uncles
- `transactions` - list of all transaction objects that were included in this block (described below)

Certain available information was omitted from the generated record, specifically roots of state tree, transaction tree, and receipt tree.

Each transaction holds the following information:

- `blockHash` - hash of the block the transaction resides in
- `blockNumber` - index of the transaction's block
- `from` - address of the sender
- `to` - address of the receiver, or new contract address, if the transaction is a contract creation
- `gas` - gas provided by the sender
- `gasPrice` - gas price provided by the sender in Wei
- `hash` - hash of the transaction
- `input` - data sent along with the transaction
- `nonce` - the number of transactions made by sender prior to this one
- `transactionIndex` - integer of transaction's index position in the block
- `value` - value transferred in Wei
- `cumulativeGasUsed` - the total amount of gas used when this transaction was executed
- `gasUsed` - the amount of gas used by this specific transaction alone
- `logs` - list of log objects, which the transaction generated
  - `data` - data of a log
  - `topics` - dist of topics of a log
- `logsBloom` - bloom filter for the generated logs
- `timestamp` - timestamp of the block this transaction is part of
- `internalTransactions` - List of internal transactions that were a part of this normal transaction

Some information is omitted in transaction records as well, specifically ECDSA recovery ID, ECDSA signature r, and ECDSA signature s. Transactions and their receipts are recorded separately in Ethereum implementation, but since the Ethereum blockchain explorer records only confirmed transactions, every transaction's execution information will always be present, and thus is included in the transaction object. A timestamp is a piece of extra information allowing for faster information querying.

Returned addresses contain the following information:

- `address` - unique address identifying the account
- `code` - code of the contract, or 0 if the address is not a contract
- `balance` - the current balance of the address
- `inputTransactions` - list of transactions where the address is the input
- `outputTransactions` - list of transactions where the address is the output
- `mined` - list of block hashes mined by this address
- `tokenContract` - whether the address is a ERC-20 or ERC-721 contract (or neither)
- `inputTokenTransactions` - list of token ERC-20 or ERC-721 token transactions, where this address was an input

- `outputTokenTransactions` - list of token ERC-20 or ERC-721 token transactions, where this address was an output
- `inputInternalTransactions` - list of internal transactions, where this address was an input
- `outputInternalTransactions` - list of internal transactions, where this address was an output

As Ethereum maintains a state tree for each block, only current information is available through JSON RPC with no way to query for transaction history of an address. This means that it has to be built manually by parsing the blockchain from the beginning.

The token transfers that can be queried through a participant's address, or the token itself contain the following information:

- `addressFrom` - Input address of the token transaction
- `addressTo` - Output address of the token transaction
- `tokenAddress` - Contract address associated with the token. Can be used to look up more information about the token
- `timestamp` - Timestamp of the token transfer
- `transactionHash` - Hash of the transaction this token was a part of
- `value` - Value transferred. In case of ERC-20, the amount of transferred tokens, in case of ERC-721 the ID of the unique transferred token

The following information is tracked about the ERC-20 and ERC-721 tokens:

- `symbol` - Symbol of the token
- `name` - Name of the token
- `decimals` - Decimal places, or how much a token can be divided. Only relevnt to ERC-20 tokens.
- `totalSupply` - Total supply of a token
- `type` - Whether the token is ERC-20 or ERC-721
- `contract` - Contract address associated with the token

Last but not least, data about the internal transactions contain the following fields:

- `from` - Address of the sender of the internal transaction
- `to` - Address of the receiver of the internal transaction
- `value` - Value transferred in Wai
- `input` - Data sent with the message call
- `output` - Output of the message call
- `traceType` - Type of the trace. Possible values: call, create, suicide, reward, genesis, daofork
- `callType` - Call of the trace. Possible values: call, callcode, delegatecall, staticcall
- `rewardType` - Type of the reward. Possible values: block, uncle.
- `gas` - Gas provided with the internal transaction

- `gasUsed` - Amount of gas used during the execution of the internal transaction
- `transactionHash` - Hash of the transaction this internal transaction was a part of
- `timestamp` - Timestamp of the internal transaction
- `error` - Error message in case of failure

4.3 Libraries and tools

This section will detail the programs, tools, and libraries intended to be used in the creation of the Ethereum blockchain explorer. Some of the original approaches were changed during development due to major flaws or unsatisfactory performance.

4.3.1 Geth - Ethereum full node

A node is a program run on a client's computer, whose purpose is to connect to other peers, download, verify, build, and update the blockchain, and respond to user's queries in accordance to its API. In order to fill its database and keep it updated, blockchain explorer has to rely on communicating with the node to receive the constantly updating information. Multiple Ethereum nodes exist, differing in the programming language they were written in, as well as their overall functionality and stability. The most popular one and the defacto standard is called Go-ethereum [10], or geth, written in Golang and actively developed by the official Ethereum team. The node can be communicated with locally or remotely, using HTTP requests, WebSocket, or an IPC socket. As Ethereum explorer will gradually query the entire blockchain, local node access is preferable, as network communication may become the bottleneck in the database building phase. Ethereum nodes have defined a unified JSON RPC API, used for getting the information about blocks, transactions, receipts, and account states. While the API is universal across several node implementations, the returned information differs somewhat in its format, which could cause an issue if the blockchain explorer was run on a node implementation which was not accounted for. As geth version is the most popular, blockchain explorer will consider its format to be the default. Parsing the blockchain directly is not feasible, because there is no detailed documentation about its inner structure, and it is unlikely that a third party implementation would reach a higher speed in fetching and deciphering the blockchain compared to the official implementation.

4.3.2 Python and supporting libraries

The language chosen for the implementation is Python, as most of the tools created for project Tarzan are written in Python or PHP. Another reason is a wide variety of available libraries and supported technologies, leaving the doors open for alternative implementation strategies. In the initial design, Web3.py [21] library was used to gather the data from the blockchain. However, due to issues which were explained in chapter 5 a new data gathering strategy had to be found. The finished program uses a tool called Ethereum ETL, whose functionality and usage is explained in more detail in section 5.2.

As the goal of the thesis is not to create a python library, but a RESTful API, a module for REST endpoints definition will also be required. Many options exist in Python, with the most popular of them being Django, Flask, and Restless. Last but not least, gathered data would reside in a database, which Python will need to interact with. The used type of

database ended up being RocksDB [15], due to the issues found in the originally planned LevelDB database. More information about the problems can be found in section 5.11. The RocksDB database can be used directly by Python, and its bindings are provided by the python-rocksdb library [22].

4.3.3 Database

One of the most important tasks in the creation of the Ethereum blockchain explorer is the definition of an appropriate structure of the database, which would allow the application to give answers to queries in real time. The goal is to find the balance between a small number of disk reads and having a low redundancy of the saved data. Also, with the analysis of proposed query types, it is appropriate to structure the data in a way that would allow for faster sequential reads instead of numerous random single reads.

For this project, a key-value type database was chosen, as it is also used in Bitcore node and Insight API, and is a proven concept. Compared to the traditional relational databases it has certain advantages and disadvantages, the biggest downside being its lack of flexibility since it has no support for SQL queries. However, as the possible use-case set is not significantly larger than the functionality defined in section 4.1, support for more variant and complex queries is not necessary. On the other hand, RocksDB's strength is its speed in answering certain queries which take advantage of the defined data structure. This means that the data storage has to be designed in a way that best serves the endpoints defined in section 4.1. Another advantage of RocksDB is its simplicity, which however has a drawback of being less intuitive and having to serialize structured data before storing them. Throughout the development, changes were made to the original design, mainly due to performance issues.

As mentioned in the previous paragraph, an appropriate storage structure has to be created in order to ensure fast responses to given queries. The following key-value pairs have been defined for the block endpoints:

- Key: `blockIndex`
- Value: `blockHash|parentHash|nonce|logsBloom|miner|difficulty|totalDifficulty|extraData|size|gasLimit|gasUsed|timestamp|sha3Uncles|transactionHashes`
- Key: `blockHash`
- Value: `blockIndex`
- Key: `timestamp`
- Value: `blockIndex`

The first key-value type stores all information about the block, with the key being the blocks unique index. Both block index and block hash are viable candidates for a key, however as RocksDB is a sorted key-value database, using the index will cause the blocks to be chronologically sorted, allowing for faster, sequential reads for queries requesting multiple blocks in a certain time range. The other two types are defined to translate the block's hash or timestamp to its index. While additional read is required for these queries, the goal was to reduce data redundancy by only including block index in the value field of these pairs. In order to save space, full transactions information is not kept within the

block but is saved separately, which means that multiple additional reads need to be made in order to get this information. Individual transactions are saved in the following key-value structure:

- Key: `transactionHash`
- Value: `blockHash|blockNumber|from|to|gas|gasPrice|input|nonce|value|cumulativeGasUsed|gasUsed|logs|contractAdress|timestamp|internalTxIndex`

The transaction information is saved with its key being its hash, as it is a unique property. The 'internalTxIndex' is a piece of data that is not a part of the transaction's original information. It is used for gathering a list of child objects belonging to its parent, and can also be seen in the address and token data. It is explained in greater detail in section 5.6 since it was an alteration of the original design due to performance issues. Following key-value structure is used to hold information about addresses:

- Key: `address`
- Value: `balance|code|inputTxIndex|outputTxIndex|minedIndex|tokenContract|inputTokenTxIndex|outputTokenTxIndex|inputIntTxIndex|outputIntTxIndex`

Address code was chosen as a key because it is unique and is used as an identifier in queries. Each address holds information about its transactions, token transfers, and internal transactions. Similarly to transactions, it uses local indexes to represent the association to its child objects. This data was designed in a way that allows for filtering out the transactions based on their timestamp or transferred value without requiring to perform a full lookup. As information about specific transactions the address was involved in is generally scattered around the database, multiple random reads will need to be performed to collect all this information. The following key-value structure is used to hold information about tokens:

- Key: `contractAddress`
- Value: `symbol|name|decimals|totalSupply|type|txIndex`

Contract address was chosen as a key of the individual tokens, as it is the most useful unique identifier associated with a given token. The next two key-value structures show how token transfers and internal transactions are saved within the database:

- Key: `tokenTxIndex`
- Value: `tokenAddress|addressFrom|addressTo|value|transactionHash|timestamp`

- Key: `intTxIndex`
- Value: `from|to|value|input|output|traceType|callType|rewardType|gas|gasUsed|transactionHash|timestamp|error`

These two types of data are unique, as they cannot be searched directly by the user. The reason is that they lack any globally used unique identifier. Instead, their keys are simply indexes that are tracked only by the database. The data can be accessed by the structures they are associated with, like the addresses of their participants, the hash of its transaction, or an address of its token.

Chapter 5

Implementation

This chapter describes the implementation details of the developed tool, shows the internal architecture, and argues the alterations that have happened from the proposed design. As mentioned in the previous chapter, the program was created in Python 3 with blockchain data being saved in a key-value database. However, some changes of its inner workings have occurred, mainly due to unsatisfactory performance, or significant bugs present in the utilized software. The most critical bug that has been encountered could have had a major negative impact on the usability of the developed program.

5.1 Overview

The developed program actually has to perform two distinct functions at once: update the database with blockchain data, and maintain a REST API interface that gathers data from said database. As it would be highly undesirable to momentarily turn off the REST API in order to add new information into the database, the program functions as two separate processes running concurrently, with each of them fulfilling their own separate functionality. However, before the fork occurs, the program only updates the database until it has caught up to its current state. Only when this action has been completed is the REST API server initialized. The database updating process periodically wakes up and adds new entries into the database. This was an arbitrary design decision that was not influenced by any implementational constraint, meaning that, if desired, the database could start handling queries even if it has not synced up to the present. However, as the synchronization process performs a lot of database I/O, the program may fetch queries more slowly in this state. Blockchain information is added to the database chronologically, meaning that partially synced database will return consistent historical data, up to a certain point in time, its current syncing progress. The exception is the constantly updated data it contains, namely the balances of Ethereum's addresses. In a partially synchronized database, this information will either not be present, or will be out of date.

5.2 Data gathering

This section describes the inner workings of the part of the program whose goal is to gather data from the blockchain and save them to the database. This functionality is contained in three classes, `DatabaseUpdater`, `DataRetriever`, and `BalanceUpdater`. The main class is the `DatabaseUpdater`, whose goal is to orchestrate the main gathering loop, save gathered

data to relevant structures, process this data, and write it to the database. The main loop runs until the entire blockchain has been processed, with one iteration representing a batch of data being processed at once. Each batch is actually a certain, usually constant number of blocks taken chronologically, starting from where it left off in the previous iteration. This number can be explicitly set by the user. Larger batches typically decrease the total time spent filling the database, as they allow for greater parallelism, and lower the required number of database reads needed to update the existing data. Specifically, it is the process of adding new transactions and token transfers to addresses, as existing information has to be read before the new information can be added. As new information is actually written to disk at the end of the iteration, it is kept entirely in memory when the data is being processed. This means, that choosing too large of a batch size can deplete all the available RAM and either decrease performance by causing thrashing, or cause the program to be killed entirely by the OS. The problem is, that despite the fact that a constant number of blocks is being processed at once, the amount of processed data can vary widely. Since Ethereum is designed so that a new block is created at roughly the same intervals, the number of transactions within individual blocks can vary depending on how much the currency is being used. This means, that the blocks near the beginning of the blockchain will be processed more quickly and will be less memory intensive. For this reason, there exists a mechanism that can semi-dynamically lower the number of blocks processed at once. It is described in section 5.12.

5.3 Data retrieval method

The initial plan of how to gather the data from the blockchain was to simply use the JSON RPC it provides, called web3. However, during the testing, it was discovered that this method was unacceptably slow, with roughly only 1 GB of data being added to the database after ten hours of running. Ethereum data structures were designed for the purpose of serving as an immutable archive of the currency's transactions, so they are poorly optimized for fetching large amounts of data. In addition, these requests can only be performed synchronously, meaning there is no possibility to perform them in a parallel, asynchronous manner. Thankfully, there exists an unofficial tool that implements these asynchronous calls and is able to gather the data significantly faster. The tool is called Ethereum ETL and was developed by Evgeny Medvedev [9]. It contains the functionality of gathering data from specified blocks and saving it into a CSV file. Ethereum ETL is used extensively in the created program, as it can gather many different types of information from the blockchain. It functions as a command line tool and is invoked directly by the Python code. While it can be argued that writing the information into a CSV file, and subsequently reading it results in an unnecessary file I/O operations, the extra time turned out to be insignificant enough for it not to matter. Gathering of account balances, which is required for the developed program is not provided by Ethereum ETL and had to be implemented. The functional principles were taken from the Ethereum ETL's code so that the gathering speed could be extended to this operation as well.

The following information is gathered from Ethereum's blockchain:

- Blocks
- Transactions
- Transaction receipts

- Transaction logs
- Contract addresses (from receipts)
- Contracts
- Token creations (from contracts)
- Token transfers (from logs)
- Internal transactions

Depending on the passed arguments, only a subset of these items is being collected. Information that is always being gathered is the block, transaction, transaction receipt, contract addresses, and contract data. If the user wishes to track ERC-20 and ERC-721 information, token creation contracts, and token transfers are gathered as well. As some of these operations are CPU intensive, they can noticeably increase the total synchronization time. Another data that the user can opt out of gathering is the internal transactions. The reasons why they are not included by default are the slower synchronization and higher storage requirements, as explained in more detail in section 5.9.

5.4 Balance gathering

As mentioned previously, there is a piece of collected data that has to be handled in a different manner from the rest. That data is the account balances, whose main difference is that they are not immutable and are expected to change over time. Unlike the rest of the gathered information, which is historical and unchanging, gathered balances could change by the time the database filling has been completed. For this reason, account balances are not actually collected during the synchronization. Instead, every encountered address is saved to a temporary file `addresses.txt`. The class`BalanceUpdater` is where most of this logic resides. In order to keep the file size manageable, periodic duplicate removals are performed (currently set to every 5 batches). After the database initialization has caught up to the end of the blockchain, the next stage is started where a balance of every encountered account is gathered. Since there are currently more than 60 million distinct addresses in the Ethereum mainnet blockchain, this stage is expected to take a considerable amount of time. It is therefore expected that the database (and many balances) will become out of date by the time it is completed. For this reason, there exists a check that find out how far behind the database has fallen. If the number of blocks is over a certain threshold, a new synchronization is started in order to catch up to the blockchain. Once again, the encountered addresses are gathered and their balance is queried after the sync has been completed. It is apparent that the number of newly encountered addresses will be significantly smaller, however, the sync can recommence once again in case it fell behind.

The balance gathering process is implemented asynchronously, taking some methods and classes from Ethereum ETL. Code that makes the asynchronous geth requests, and extends the existing functionality can be found in the `requests` folder. Files `auto.py`, `ipc.py` `rpc.py` and `thread_local_proxy.py` have been taken directly from Ethereum ETL, and their purpose is to handle the asynchronous interface calls and multithreading. The file `balances.py` contains the newly implemented API call for gathering account balances.

Depending on the used arguments, some balance information may not be correct in the created database. As previously stated, only the encountered addresses are saved to the temporary file, and their balance is later queried. If the user has chosen not to track the token transfers and internal transactions, their addresses would not be saved (unless they

were also a part of a normal transaction). This means that only a subset of active addresses would be queried, resulting in outdated balance information for the remaining ones. A possible solution is to simply query the balance of all recorded addresses. This is obviously not viable as it would take a significant amount of time, especially considering the fact that the database is supposed to synchronize every 15 seconds. It would also not resolve the problem, as some addresses could only have taken part in internal or token transactions, not being tracked by the program at all. Thus, only full database sync including the internal transactions and tokens will ensure the consistency of balance information.

5.5 Encoding and decoding

One of the properties of key-value databases is that it can only save the data as a sequence of bytes. This means, that any semantics the data has, like its dictionary structure will be lost when it is inserted to the database. This creates a requirement of saving the structured data into a byte stream of predictable format. Subsequently, the data must have the ability to be structured back into a dictionary for when REST queries are performed. The file `coder.py`, which is a collection of encoding and decoding functions has been created for this very purpose. Each encoding-decoding function pair is used for a different type of data structures, such as blocks, transactions, or addresses. The principle of encoding is to append all the structure fields in a specific order, separated with a chosen delimiter character. The delimiter was chosen to be the first ASCII character, or '\0', as it is unprintable and should not normally occur in any of the processed data. Decoding is a reverse process, where the byte stream is separated by this delimiter, and added to the corresponding field. In order for this operation to be performed correctly, the data is expected to be in the same order during both operations. A question may arise, as to why the data was appended manually instead of utilizing python's capabilities, like its pickle module. The answer is, simply, that it was implemented this way in the Insight API, and there might be some non-obvious performance or storage related reasons for why it was chosen.

Not all data in the dictionary structures have simple data types such as strings or numbers. They can also be arrays of unbounded length, which means that there is a requirement to have a nested delimitation for their elements. An example is a list of logs of a transaction or a list of topics contained in the log. The symbol for a second level delimitation was chosen to be '|' since the data inside the arrays only contain alphanumerical characters, and this symbol should not have a natural occurrence. Because each log contains data and a list of topics, these fields had to be separated with a third level delimiter, which was chosen to be '+'. Finally, topics of a transaction log is a list as well, requiring a fourth level delimiter, which was chosen to be '-'. The following example demonstrates how the data is saved in the database:

```
contractAddres\0timestamp\0data+topic1-topic2|data+topic1-topic2\0
```

Figure 5.1: Example of how transaction information is encoded

5.6 Architectural changes

Due to significant speed issues and memory requirements, certain changes had to be made in terms of how the data is being saved in the database. Specifically, the changes are related to how the associations between an address and its transactions, token transfers,

internal transactions, and mined blocks are represented. The initial design proposed that the transaction key would be saved with the address so that it can be used to look up the full information about the transaction. On first glance, this design seems reasonable, however, full blockchain synchronization has revealed serious flaws of this method.

While this approach works well on a small scale, Ethereum has many 'power users', whose accounts took part in hundreds of thousands or even millions of transactions. This means, that even a small piece of information (like a transaction hash) can significantly increase the size of an address when enough records are present. These large sizes are not inherently undesirable, they only cause problems if the data has to be fetched repeatedly during the synchronization (which is the case with addresses). This gradual size enlargement has several negative implications. Firstly, new transactions are added to an address during the synchronization process, which means that address data has to be read from the database first. With thousands of blocks being processed at once, hundreds, or even thousands of megabytes have to be read for each iteration of the synchronization process (not to mention that it is mostly the same data being read repeatedly). These excessive reading operations significantly slow down the synchronization. Secondly, all this data has to be held in memory, depleting the available resources and slowing down the program by causing thrashing. Thirdly, the data has to be saved back to the database which is a very memory intensive action even without the large sizes. Significant memory spikes were observed during this stage, with memory allocation errors being quite common. Lastly, the program could not keep up with the growing blockchain, since reading and writing the address information for a single block would routinely take more than 15 seconds to complete. Since a new block is added to the blockchain roughly every 15 seconds, this would impact the program's ability to update the database in real time.

Thus, a new strategy in associating pieces of data had to be created, with the aim of suppressing the size increase of the address data. The strategy introduces a concept of local indexes of an address, replacing the lists of hashes, or other key data. Transactions, token transfers, internal transactions, and mined blocks of an address are each represented by their own index that is local to the given address. For example, the transaction index of a given address is initially set to 0. When a new address transaction is discovered, the index is incremented to 1, and then gets incremented for all other discovered transactions. This index is used to save the associating data in a way that can be recreated by the information available to the address, specifically its own hash. The index gets incremented to ensure the uniqueness of the new keys. The following key-value pairs were introduced to the database:

- `[addressHash-i-inputTxIndex]: [txHash-value-timestamp]`
- `[addressHash-o-outputTxIndex]: [txHash-value-timestamp]`
- `[addressHash-b-minedBlockIndex]: [minedBlockHash]`
- `[addressHash-ii-inputIntTxIndex]: [IntTxIndex-value-timestamp]`
- `[addressHash-io-outputIntTxIndex]: [IntTxIndex-value-timestamp]`
- `[addressHash-ti-inputTokenTxIndex]: [tokenTxIndex-timestamp]`
- `[addressHash-to-outputTokenTxIndex]: [tokenTxIndex-timestamp]`
- `[tokenHash-tt-localTokenTxIndex]: [tokenTxIndex-timestamp]`
- `[txHash-tit-transactionIntTxIndex]: [IntTxIndex]`

A prefix of 'associated-data-' is added to each of the keys in order to define the meaning of these new records. All of the keys follow a common pattern, which can be generalized

as `localData-recordType-localIndex`. Three pieces of data make up the structured key, all of which are available to the master structure. The first is a globally unique identifier of the record itself, specifically the hash of an address, transaction, or token. The second element denotes a type of data that is to be looked up. For example, 'i' means an internal transaction, 'io' means output internal transaction and so on. The third element is a unique local index of the child record, which is generated by incrementing the previous index saved in the master record. The old index is then overwritten by its new value. The values of these new records contain the database keys of the full records, along with optional extra data. This data can be used to filter out the undesirable items before a full record lookup is performed based on the parameters of a REST query.

It is obvious that an index representation takes up significantly less data than a list of database keys since even millions of records require only a few bytes of data. The required number of bytes also increases logarithmically with the linear increase of saved records. Switching to this system has significantly decreased the reading and writing times of addresses, as well as lowered the memory requirements for the synchronization. Unfortunately, this design has its own disadvantages as well, which are mainly related to the speed of certain REST queries. As a new record was added for associating an address with its transaction, an extra read has to be performed in the process of gathering the transaction information. This effectively doubles the number of random reads needed to gather all address transactions, having a negative effect on the speed of queries requesting address information. In order to compensate for these changes and minimize their impact, a sequential reading strategy was employed. The keys of the same type of record share a common prefix, which means that they are saved sequentially in the database (since keys are ordered in RocksDB). This property allows them to be looked up sequentially, using an iterator. Sequential read times are lower than random reads, and they stay constant even with the increasing size of the database. This strategy could not completely nullify the higher fetching times, as address queries still take ~25% longer to complete (compared to the old saving strategy).

5.7 Saving the data

After the data has been collected, processed and encoded, the only step that remains is to actually save it to the database. Key-value databases offer a feature called write batches, which allow for insertion of multiple entries at once. This way, the writing operation is performed significantly faster. Moreover, write batch is an atomic operation, meaning that either all entries are written, or none of them are. However, write batches are quite memory intensive, and if large amounts of data is being written at once they can cause memory spikes and even memory allocation errors. Due to an architectural oversight described in the previous section, write batches could not be initially used, as too much data was being written at the same time. Single writes were used instead, which worsened the performance as well as the integrity of the database. However, after the changes in data structures, the batch writing operations are much more stable, with no memory errors being observed. So the data of each synchronization iteration is written using a single write batch.

5.8 ERC-20 and ERC-721 tokens

Extended functionality, which was not a part of the original design was the inclusion of ERC-20 and ERC-721 tokens. The developed application has the ability to track the creation of new tokens, as well as the transactions of existing tokens. These standards define different functionality and serve different purposes, which reflects on how they are represented in the database. ERC-20 tokens function as any other cryptocurrency, they have a defined total supply and maximum divisibility of a single unit. Their transactions simply contain the number of tokens transferred from one address to another. On the other hand, each ERC-721 token represents a unique object with its own identification number. Their transactions explicitly track the change of ownership of these individual objects.

Contract creation transactions are extracted from all the transactions of the processed batch, and Ethereum ETL examines whether they implement ERC-20 or ERC-721 operations defined by their standard. If the signature hashes of the functions match, new information is added to the database denoting the name, symbol, total supply, and unit divisibility of the new token. Token transactions are identified as well and are associated with their corresponding addresses and the token itself. They contain information like source and destination addresses, amount transferred (or object ID in case of ERC-721), and a contract address of the token. Contract address is used as a unique identifier of the token, as there is no prevention in the creation of duplicate token names. ERC-20 balances are not tracked however, as the only official way of getting the balance is to query the contract through Ethereum. Simple addition and subtraction of tokens based on the captured transfers are not feasible either, as many of these currencies implement extensions to the defined standard that allow users to lose and gain the tokens through different means.

There are caveats about how these token currencies are tracked compared to other Ethereum tracking services, like Etherscan. Tokens are considered ERC-20/ERC-721 compliant when they implement all the operations defined in the standard. For example, ERC-20 tokens must implement the following operations:

- `totalSupply() public view returns (uint256 totalSupply)`
- `balanceOf(address _owner) public view returns (uint256 balance)`
- `transfer(address _to, uint256 _value) public returns (bool success)`
- `transferFrom(address _from, address _to, uint256 _value) public returns (bool success)`
- `approve(address _spender, uint256 _value) public returns (bool success)`
- `allowance(address _owner, address _spender) public view returns (uint256 remaining)`

Some, even popular tokens like GNT implement only some of these functions, and thus are only partially compliant [8]. Ethereum tracking services can use a human element to personally determine whether to include these tokens among the tracked ERC-20/ERC-721 currencies. On the other hand, they do not track some tokens that are considered too irrelevant, despite their full compliance. As the developed program is only an automated tool, not a service with dedicated staff, the decision of whether a contract is to be tracked as ERC-20/ERC-721 has to be reached algorithmically. Since the borders of partial compliance are nebulous and debatable, only fully compliant tokens are included in the database.

The token transfers are saved in their own separate records in the database. Since they do not contain any uniquely identifying information, their key is simply an incrementing

index found only in the created application. Token transfers cannot be searched for individually (they do not contain any unique identifiers), but instead are looked up by their input/output addresses, or the contract address of the token.

5.9 Internal transactions

Another type of data which was not included in the original design but was added to the created application are the internal transactions. As these transactions allow for the transfer of Ether, their inclusion is important if the user wishes to have the full canonical history of Ethereum. Some addresses only took part in the internal transactions, which means that processing only the normal transactions will result in some addresses not being included in the database at all. These transactions make up a significant portion of the blockchain, being more numerous than the normal ones. However, there are several good reasons why they were not included in the original design, and why their inclusion is only an optional parameter.

First of all, the geth node has to be synchronized in the 'full' mode, which takes up more than 1 TB of data at its completion. This is already an issue since it should reside in an SSD disk if the synchronization is expected to complete in a reasonable time-frame. It is hard to estimate the size of the application's database, but it is safe to assume that it would be several times larger with the inclusion of internal transactions. The large size should not be an issue for the database responsiveness, as RocksDB was designed to store up to several terabytes of data [23]. The second issue is the speed of the database synchronization. The synchronization slows down significantly, taking about five times as long. Unfortunately, due to storage and time constraints, the inclusion of internal transactions could not be fully tested in the created program. But since even light Geth node keeps the full records of the last 256 blocks, these blocks were used to test the implemented functionality. Apart from the slower synchronization, no issues were detected in the processing of this extra data.

5.10 Gathering data from the database

After all the data has been collected from the Ethereum blockchain, a REST API service is started, which can retrieve the data requested by a given query. The exhaustive list and explanation of supported queries can be read in the design chapter. The logic for fetching, processing, and formatting the data is situated in the `DatabaseGatherer` class. Some queries have the ability to filter the returned results by their properties, like timestamp and the number of transferred units. This functionality is included in the said class as well. The operations are implemented in a way that database queries for irrelevant data are not performed. This is achieved by including a small amount of data with the associated entry value. This data can be used to filter out the entry before the query for full information is even executed. Extracted data is decoded, inserted into relevant data structures, and returned to its calling function.

Functions, that are actually triggered by incoming REST queries are included in files `addresses.py`, `blocks.py` and `transactions.py`. Their purpose is to validate the query input, invoke the relevant `DatabaseGatherer` method, and return the gathered data and HTTP status code. If the request is nonsensical or poorly formed, such as non-chronological time range, the server returns a 400 'bad request' message with an appropriate description. Similarly, if the database does not contain the requested entry, 404 'not found' message is

returned. Finally, if the requested query was successfully retrieved from the database, a 200 code along with a JSON structure containing all the information is returned.

Flask package was chosen as a web framework for the developed tool, mainly due to its simplicity and popularity. Instead of pure Flask, a wrapper module called Connexion was used as it simplifies Flask's usage and provides it with additional functionality. It has the ability to automatically hook the functions to their corresponding endpoints, based on the definitions in the configuration file. More importantly, it can automatically generate Swagger UI documentation, which lists all the available endpoints and parameters, the format of returned data, and provides a description for all the returned fields. The documentation can be accessed at `http://{base-url}/api/ui`. This configuration is defined in the `swagger.yaml` file.

5.11 Database implementation and issues

During the design phase of this thesis, a key-value type database was chosen for data storage, with main reasons being simplicity, good scalability and data fetching speed. Its disadvantages like the lack of SQL query functionality was deemed unimportant, as only a small, predictable number of operations would be performed. The specific implementation was chosen to be LevelDB, mainly due to being used in both Insight API and Geth, as well as its overall popularity. In the previous section, it was mentioned that the program functions as two separate processes, with one used to update the database with new blockchain entries, and the other to handle the REST API. The issue with LevelDB is, that only one database instance can exist at any given time. As sharing one instance between two processes was leading to deadlocks and various other issues, exclusive database access had to be assured in the created program. A mutex logic was implemented, which ensured that only one process at a time could access the database. This created an undesirable side effect, where the REST queries could have a delayed response due to waiting for the completion of the database update. Worse yet, a critical bug has been discovered during testing, where the database instance would not properly close. The bug would happen randomly, and it is still unclear whether it was caused by exclusive access logic error in the application, or LevelDB implementation itself. When it occurred, it would essentially deadlock the application, and so its existence was not acceptable in the finished program.

To resolve the bug, a different database was chosen, called RocksDB [15]. This database is actually a LevelDB fork maintained by Facebook, so its functionality and API is very similar. Compared to the LevelDB it has one big advantage, which allows its user to create multiple instances, as long as all but one of them are read-only. This is sufficient for the created program, as the process that handles REST API only performs reading operations. It also solves the issue of delayed API responses due to database unavailability. RocksDB is likely to be a more future proof database, as it has become a de-facto key-value database standard since LevelDB has all but ceased development. RocksDB offers extensive performance tuning options, that allow for the optimization of reading and writing speeds in the chosen deployment environment. Performance tuning was not attempted in this thesis, as it is rather complex, and even developers themselves do not fully understand the effects of various changes [16].

5.12 Recovery of memory errors

Testing has revealed, that the developed application may sometimes receive out of memory errors. These errors are quite inconvenient, as they cause the program to be killed during its lengthy synchronization process, further increasing the time of the synchronization and requiring the user's constant attention to perform a restart. The main cause of this error is setting the batch size too high for the computer's resources to handle. Deciding on the optimal batch size is not a simple or straightforward process, as it depends on the available resources of the platform, which can vary widely. It also changes over time, as batches start containing more transactions with the growing popularity of Ethereum. Random spikes of batch sizes were observed as well, introducing an unpredictability to the synchronization process.

As a solution, a wrapper script was created, which can detect and recover from these memory errors. When an out of memory error occurs, the program is restarted with the batch size decreased to half of its previous value. With this mechanism, the batch size eventually reaches the point where it can be safely handled by the system's resources. Moreover, the size of the batch dynamically decreases with the increased number of transactions in a block, compensating for the higher batch sizes. This way, more blocks with smaller sizes can be processed simultaneously, decreasing the overall synchronization time.

One more error type was observed during the synchronization, namely the socket receive timeout. This error occurred infrequently, and with a high degree of randomness, so its root cause is not easily determined. An educated guess is that it is likely related to long fetching times of geth data, which is not something that can be easily resolved. In any case, the program restarts upon encountering this error, but it does not decrease the processed batch size. This is because the error is not really indicative of high memory usage, and because decreasing the batch size too eagerly will have a negative impact on the database synchronization speed.

5.13 Container environment

Part of the implementation was also the creation of a Docker [6] container, which would serve as a running environment for the created application. This would solve the issue of compatibility to the deployed platform and automatically deal with the installation of dependencies. Ubuntu was chosen as the base image, mainly because it allows for the installation of all of the program's dependencies through its package manager. This is useful mainly for the RocksDB database, which does not support automatic installation in every distribution of Linux, having to be compiled from source. It also solves the issue of using a virtual environment for installing the Python modules. Docker container is orchestrated through docker-compose [7] so that REST API endpoints are exposed and required data is imported.

As it would not be a good idea to store large, permanent database inside of a container, the database is actually stored in a folder outside of the container's environment. This way, in case the container was deleted, all of its gathered data would not be destroyed as well. In total, three external folders are included in docker-compose: 1) database; 2) the data folder; 3) optional folder is where geth's data is stored. The data folder is important, as it holds information about the state of the synchronization, and is required for it to run correctly. It holds the information of the last successfully gathered block and the highest index for token and internal transactions. It also remembers all encountered addresses for

the purposes of balance gathering. In the case of database backup or duplication, these two folders need to be carried over. The third folder contains geth's data information, and it is optional since it is only required for the synchronization done through the IPC interface. As the IPC file is situated in this folder, the Docker container must have direct access to it.

5.14 Application arguments and final remarks

This section describes the application arguments and explains their meaning. The program can be run with the following arguments:

- `--interface` - Geth IPC, WS, or HTTP interface. Required for data gathering and database creation.
- `--dbpath` - Path to the folder in which the database is to be stored. In the container application, this folder defined by docker-compose is used by the Entrypoint.
- `--datapath` - Similar to `--dbpath`, used to define the location of the data folder. Warning: During the synchronization, the folder will reach the size of a few Gigabytes, so the disk it is situated on should have enough free space.
- `--confirmations` - The minimum amount of confirmations a block has to have before it will be added to the database. This argument exists because the canonicality of the most recent blocks may change, and the database might be filled with irrelevant, false data. The default value is 12, which guarantees more than 99.9% probability of permanence while being only about five minutes behind the blockchain.
- `--refresh` - How often should the database be updated with new data. As new blocks are created roughly every 15 seconds, 20 seconds was chosen as a default value to account for some expected time variance.
- `--bulk_size` - How many blocks should be processed simultaneously during data gathering. This option could have significant effects on the speed and stability of the application. The default value of 10,000 was chosen, as it was both stable and fast during the testing.
- `--internal_txs` - A boolean parameter of whether to also gather internal transactions from the blockchain. Fully synced geth node is required. Database creation will be significantly slower.
- `--gather_tokens` - Another boolean parameter for defining whether tokens and token transactions should be gathered as well. Synchronization will be somewhat slower, but not by a debilitating margin.
- `--max_workers` - Maximum number of workers to be used in Ethereum ETL. Can be used to fine-tune data gathering performance. Caution: choosing more workers is not inherently faster. The default value is five since it is also default in Ethereum ETL.

In order to maintain the readability, maintainability, and potential reusability of the developed code, styling and typing checks have been introduced. The code is compliant to the PEP 8 styling standard, enforced by the linting tool called Flake8. Each file, class, method, and function contain docstrings written in Google style, explaining their purpose. As Python does not natively contain explicit type definitions, the tool called mypy was used to allow and enforce this functionality. This makes the code more easily understandable and can detect some non-obvious errors. Tests verifying the validity of returned data were

also created. They compare the information returned from the database with the data available in various online Ethereum blockchain explorers. Styling and typing checks, as well as the tests can be run by using the `tox` command in the project's folder. The program was developed on Ubuntu 18.10.

Chapter 6

Testing

This chapter explores the performance and validity of the developed tool. It looks at the process of database synchronization and its speed and attempts to showcase the viability of the implemented solution. Then it focuses on the analysis of REST API queries, identifies their validity and performance, and compares them to existing tools.

6.1 Database synchronization environment

For testing purposes, a virtual machine has been provided, located on a school server. The machine had 16 GB of RAM and 1 TB of free disk space. The server was equipped with mechanical disks, which was not ideal for the testing of the developed application, as both geth and the created database rely heavily on the effectiveness of numerous random reads. As access speed is the one area where SSDs significantly outperform classic mechanical disks, a degraded performance was expected to be observed, despite the server using RAID technology.

While the developed tool supports all three interfaces the geth provides, IPC was likely to be the fastest, as data is being transferred locally, and no network bottleneck is introduced. In order for this setup to be used, the Ethereum node has to be available locally on the same disk. For this reason, a synchronization of the geth light mainnet node was initialized on the provided server. However, a significant slowdown has occurred, stalling the synchronization and making it unable to complete after more than a week of waiting. Due to fear of the inability to test out the application at all, a different strategy of testing on a personal SSD disk was chosen. As the light mainnet node has a size exceeding 200 GB, and the created database was going to be at least just as large (this was actually just a best case estimation), an SSD with the size of 1 TB was required for the testing. Only the SSD with the size of 256 GB was available, which was the reason for deciding to test out the application on a testnet network. The slowness of mainnet geth node synchronization was actually only a part of the reason why a testnet node was used to test out the program. For the majority of the application development and testing, the database creation process has been very slow and very taxing on the system resources (mainly RAM). The database synchronization of a testnet node (which is about 50 GB) took about a week to complete when both the node and the database were situated on an SSD disk. The testnet synchronization on the school server was attempted but was abandoned since it was progressing even more slowly. As creating the database with mainnet data on a mechanical disk would have taken more than a month to complete, it was never attempted, as there would not

be enough time to fix any possible bugs. Fortunately, the reason behind the slowness was identified and a fix was created, as is described in section 5.6. The cause of the issue was found very late in the development though, which is why there was only enough time to test it out on a testnet node.

After the deployment of the fix, the synchronization time and stability have improved significantly. The process remained stable all throughout, having no issues with the increasing size of the database, or higher densities of newer blocks. The processing of gathered data and writing it to the database took only a small fraction of total runtime, and this fraction was not increasing throughout the synchronization process. For these reasons, I believe that mainnet database creation can also be performed with no issues, with a linearly scaled processing time (so about four times as long as testnet). With the significantly improved synchronization speed, mechanical disks (with RAID) could also be a viable environment for the application.

6.2 Synchronization process

This section examines and describes the process of creating the database from the data gathered from the Ethereum node. Various batch sizes were tested out, but the size of 10,000 blocks was eventually chosen as it has both good speed and high stability. At the completion of each batch, a synchronization percentage is displayed, denoting the fraction of processed blocks to all available blocks. A batch of 10,000 blocks corresponds to roughly 0.2% of total progress. The percentage shown only denotes timeline based progress, so it is only partially indicative of how much data is still to be processed. Even in testnet, the amount of data included in one block rises significantly over time, increasing how much data is processed in one batch. As an example, at the synchronization point of 50%, the created database only had about 20 GB, less than a third of its final size.

The testnet synchronization on an SSD disk took 30 hours to complete. The process was stable, with no errors or system slowdowns being observed. While the CPU usage spiked during some operations, it had no adverse effects on the responsiveness of the OS or the usability of the system. The process has high memory requirements, with about 10 GB of RAM needed for a batch size of 10,000 blocks. The memory usage is divided between the geth node itself, the process of gathering its data, keeping it in memory, and writing it to the database. This usage remained stable throughout the synchronization, experiencing no spikes or fluctuations. Available RAM should be the primary factor in deciding the optimal batch size on a given system.

At the time of the synchronization completion, the database size was 72 GB. While the size has increased from the original 50 GB, the increase is not significant enough to be a cause for concern. The most likely reasons for the size increase are the data redundancies which were introduced for the purpose of faster data filtering, and the creation of address-transaction associations. Inclusion of ERC-20 and ERC-721 tokens is likely an another contributing factor, since part of the data is saved redundantly. The first time it is saved as raw data of a transaction, and the second time as parsed token information. As the fraction of redundant data is likely to stay similar to current values, it can be assumed that the database size will increase linearly over time. Thus, if the size increase percentage remains the same in the future, the larger database size should not pose any problems.

Processing times and memory usage have increased in the latter parts of the blockchain. The first 50 % of blocks were actually processed in about 4 hours, as they contained significantly fewer transactions. The slower processing rate of the second half of the blockchain

was caused by the higher gathering times of Ethereum ETL, which occurred due to more data needing to be fetched. In order to analyze the efficiency of the synchronization, and identify possible bottlenecks, profiling of the running code was performed. A graph containing the percentages of time spent in the program's functions can be seen below:

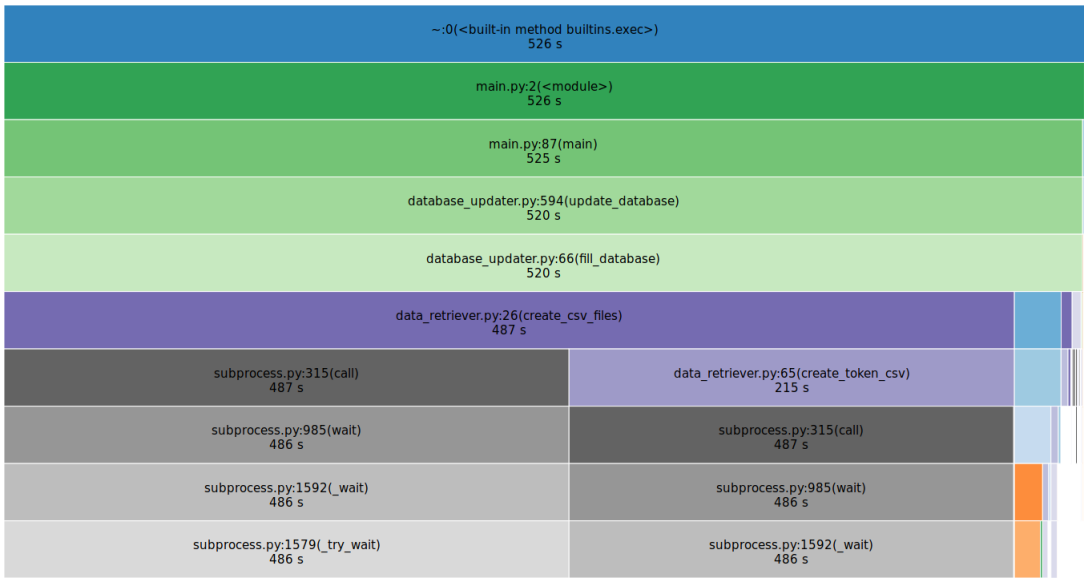

Figure 6.1: Profiling of the database synchronization phase

The graph was created by a tool called SnakeViz [18]. As it can be observed, the majority of the program's runtime was spent gathering the data from the blockchain. The file I/O, data processing, and interacting with the database make up only a small fraction of the total runtime. There is little to be done in optimizing the data gathering process, as the operations are already running asynchronously, and their efficiency can only be improved by the developers themselves. Thus, it can be concluded that the program itself contains no significant bottlenecks, and any synchronization speed issues are not caused by its implementation or the design of its structures.

One of the goals of this project was for the application to not only provide historical data but also have the ability to add newly created data in real time. The program was designed as such, having a process that periodically wakes up and processes all newly added transactions. After the synchronization has been completed, the program has no issues keeping up with new blocks added to the end of the blockchain. The process of adding the new data takes one to two seconds, which is sufficient since a block is added to the blockchain roughly every 15 seconds. The resource requirements are low, which is important since the application is expected to look up the user's queries. No slowdown in fetching speeds was recorded as a result of data being written to the database. While the internal transactions could not be fully tested due to reasons explained in section 5.9, they were tested near the end of the blockchain. For this reason, the ability of the full gathering process to keep up with the blockchain could have been observed. Even with the inherent slowness of collecting the internal transactions, the program had no issues keeping up with the growing data. Thus, it can be concluded that the application has fulfilled the design goal of updating the database in real time.

6.3 Validity testing

This section attempts to validate the data returned from the application's API calls. The only practical way to evaluate them is by comparing the returned results to the data available from an authoritative source. These sources can either be various Ethereum blockchain explorers available on the internet, or the Ethereum node itself. Both of these sources have their own problems. The Ethereum node does not readily provide some vital pieces of information, like the transactions of a given address without requiring significant processing time (most likely days for a single address). Ethereum blockchain explorers, such as etherscan provide a limited API, which returns only some of the tracked information, has a limit on the amount of performed requests, and can only return up to 10,000 items per result. For these reasons, a full database validation is not computationally feasible in the limited time-frame of this work. Instead, only a few examples of each returned type of data will be compared with an authoritative source, since the correctness of one piece of data can be at least somewhat indicative of the validity of other data of the same type. The most extensive, and readily available data is on the Etherscan website itself, which is why it was chosen as a reliable source to compare the gathered information against.

As the returned information takes up a lot of space, only a small example will be included in the text of this work. The following example shows the information about a chosen transaction returned from the blockchain explorer:
Query:

```
curl 'http://localhost:5000/api/tx/0x01ce2545aa93225dc16368d8fd0c08de06
      15ff16947849fb8539049da82bb0c0'
```

Returned data:


```
{
  "blockHash": "0
     x5718e3212cfbd3d472bd181ad10a0c10cffd45ab4f34d7a92ac2bcbc256bbe17",
  "blockNumber": "15011",
  "contractAddress": "",
  "cumulativeGasUsed": "394284",
  "from": "0x13c0127b66b336a644cc36c2e44eadc5fcd8b79d",
  "gas": "250000",
  "gasPrice": "20000000000",
  "gasUsed": "98571",
  "hash": "0
     x01ce2545aa93225dc16368d8fd0c08de0615ff16947849fb8539049da82bb0c0",
  "input": "0x51a34eb8000000000000000000000000000000000000000000000000
           00000850fc6266aec4000",
  "internalTransactions": [],
  "logs": [
    {
      "data": "0x0000000000000000000000000000000000000000000000000000
               00850fc6266aec4000",
      "topics": [
        "0xa609f6bd4ad0b4f419ddad4ac9f0d02c2b9295c5e68914
         69055cf73c2b568fff",
        "0x000000000000000000000000ab0cfa8d81c051dc883127e898a2716b712..."
```

```
      ]
    }
  ],
  "nonce": "161",
  "timestamp": "1479763707",
  "to": "0xab0cfa8d81c051dc883127e898a2716b712c6e55",
  "value": "0"
}
```


0x01ce2545aa93225dc16368d8fd0c08de0615ff16947849fb8539049da82bb0c0 is the hash of the transaction, and its validity can be verified on the website https://ropsten.etherscan.io/.

The second example is the information about an ERC-20 token:
Query:

```
curl 'http://localhost:5000/api/token/0x9724a061eaff6e34f4127272bf84ca
      b690f98339'
```

Returned data:


```
{
  "address": "0x9724a061eaff6e34f4127272bf84cab690f98339",
  "decimals": "18",
  "name": "Tetris",
  "symbol": "PCC",
  "tokenTransactions": [
    {
      "addressFrom": "0x4d3d0f631e4c3d2bff847c3f692b1d99eb69ce47",
      "addressTo": "0x4ee9600513cfbb07b7c127e68af8fdf98bb64dbd",
      "timestamp": "1519245379",
      "tokenAddress": "0x9724a061eaff6e34f4127272bf84cab690f98339",
      "transactionHash": "0
          x1d804cb19bb3447b407ee99197d270037635de4dfa024adc25c7232504b6548f
          ",
      "value": "100000000000000000000000"
    },
    {
      "addressFrom": "0x4d3d0f631e4c3d2bff847c3f692b1d99eb69ce47",
      "addressTo": "0x4ee9600513cfbb07b7c127e68af8fdf98bb64dbd",
      "timestamp": "1519245459",
      "tokenAddress": "0x9724a061eaff6e34f4127272bf84cab690f98339",
      "transactionHash": "0
          x8de8b4dbadfffaa918b959b7499444bffcc227b7e4c47167806947f4664d93b8
          ",
      "value": "999000000000000000000000000"
    }
  ],
  "totalSupply": "1000000000000000000000000000",
  "type": "ERC-20"
}
```

Once again, it can be verified through the aforementioned website.

As the returned data can be quite long (shorter examples were chosen on purpose), other examples can be examined in the Appendix B. Tests, which compare the data returned by the program with the publicly available information were created. They can be found with the program's source code and they validate every type of returned data. A path to a database containing testnet information must be provided, in order for these tests to function properly.

6.4 Performance testing

The next section details the performance of the created database, and the speed of its information retrieval. The database was saved on a personal SSD drive, meaning that the measured times are likely to be much faster than on a mechanical hard-drive. Single random read operations, such as the data of a transaction or token information happen instantaneously. All the measured retrieval time differences could most likely be attributed to processes unrelated to the created application, like the activity of the OS or other running programs. Due to the short running times of these simple queries (usually less than 1/10th of a second), the noise of the environment becomes a significant factor in the speed of these operations. As a result, they are not very informative as a performance benchmark of the database.

Instead, larger queries requiring many random read operations were used highlight the retrieval times, and showcase the observed trends. The first graph shows the number of seconds it took to retrieve the data about multiple blocks from the database:

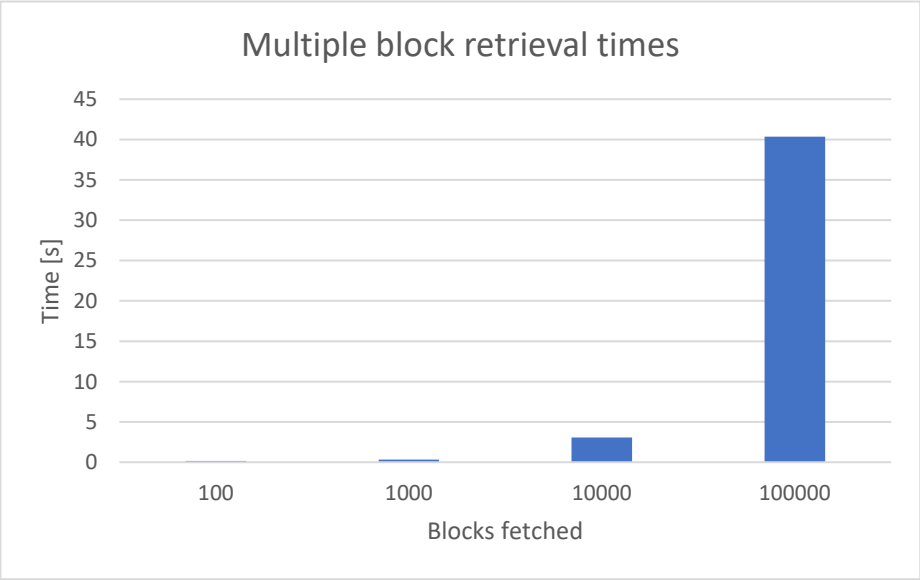

Figure 6.2: Block retrieval times

Specifically, the graph denotes retrieval times of 100, 1,000, 10,000 and 100,000 blocks. The measured times are 0.153, 0.326, 3.069 and 40.358 seconds respectively. It is important to note that along with the block data, information about each block's transactions was retrieved as well. To limit the influence of these extra operations, blocks from the beginning of the blockchain were gathered, since most of them contain zero or very few

transactions. However, especially in the case of the 100,000 blocks, these extra fetches had a non-insignificant effect on the overall time.

The second example shows the retrieval times of information about an account that was involved in large amount of transactions. The tested account has the address `0x70c9217d814985faef62b124420f8dfbddd96433`, and it took part in 98335 transactions. The collected data can be seen in the graph bellow:

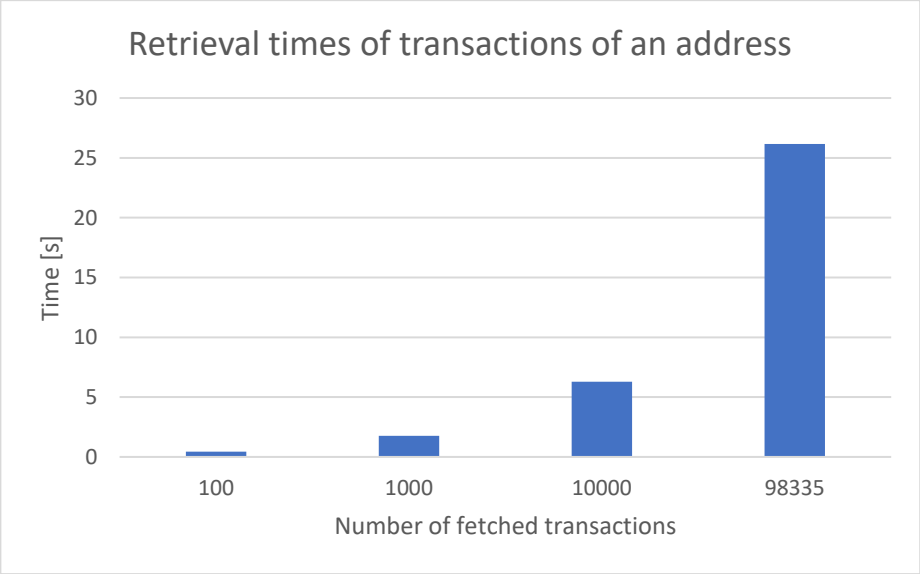

Figure 6.3: Address transactions retrieval times

To measure the retrieval time differences based on the amount of gathered data, the desired number of transactions was specified in the query. The graph shows the time it took to collect 100, 1,000, 10,000 and all 98,335 transactions. This query shows more 'pure' results, as the amount of requested transactions directly corresponds to the number of performed database queries. Database implementation specifics had to be considered when performing these measurements, as RocksDB maintains a separate cache that helps with faster execution of previously performed queries. This mechanism results in significantly lower retrieval times of previously executed requests, invalidating the gathered time measurements. In order to keep the measuring environment constant, the database was rebooted before each query.

The performed queries were completed in 0.43, 1.76, 6.27, and 26.16 seconds. Each query requests 10 times as much information as the previous one, meaning that the expectation was for the times to rise exponentially. The measured time trend is actually sub-exponential, indicating that the time of one random read decreases with the rising amount of queries needing to be performed. The measured times and time trends suggest that the database will be both a viable tool for gathering random requested data, as well as for larger bulk requests.

6.5 Comparison with existing tools

The last section describes how the developed program compares with the existing tools. Various Ethereum blockchain explorers can be found on the internet, and allow their users to look up the desired information through a web interface. Some of them also offer APIs that

gather information in a more structured format. The following services were found in the research of this thesis: https://etherscan.io, https://ethplorer.io/, https://blockscout.com, and https://blockchair.com. Only the services which offer APIs were considered so that meaningful comparisons could be made between them and the created program. It should be noted that all of the available APIs offer a smaller amount of endpoints than the one created in this thesis, and also provide the user with less information. Comparing simple queries, such as data of a transaction is not very useful since the low fetching times will ensure that the most significant speed factor will be the transfer over the internet.

Thus, larger data requests would serve as a better comparison. Unfortunately, all of these APIs have significant restrictions on how many requests can be made in a given time-frame. This means that the speeds cannot be measured by creating many simple API queries at the same time. Instead, the only feasible way of measuring performance is by executing one query, which fetches a large amount of data. The only query present in all of these APIs is the gathering of all transactions of an account. Limitations exist here as well since only up to 10,000 transactions can be gathered from one account. Blockchain explorers https://ethplorer.io/ and https://blockchair.com are even more limited, providing only up to 1,000 and 100 transactions respectively. As these numbers are not significant enough to draw any conclusions, only https://etherscan.io and https://blockscout.com will be compared. The following table shows how long it took to execute the aforementioned query in each service.

Developed tool	etherscan.io	blockscout.com
6.275 s	14.838 s	17.855 s

Table 6.1: Retrieval times of 10,000 transactions

As a side note, https://blockscout.com returned a time-out error multiple times before the data could finally be gathered. A large time difference can be seen between the developed program and online blockchain explorers. Of course, the comparison is not completely fair, since the transfer over the internet is almost certain to negatively impact the measured speeds. However, with the gathered file sizes of only a few MB, internet transfer should not have such a pronounced effect. Also, the created application provides significantly more data than the API queries, so the additional disk I/O should put it in the disadvantage compared to these online tools. From the measurements, we can conclude that the developed tool can provide the data more quickly than existing the existing sources (even without considering the aforementioned limitations of these websites).

Chapter 7

Conclusion

The goal of this thesis was to create an application which has the ability to provide various information from the Ethereum's blockchain. The reason for creating this program was that the Ethereum's RPC interface is somewhat limited, and certain types of information are difficult to obtain. This stems from Ethereum's underlying architecture, so the focus of this thesis was to create an interface whose main parameters were speed and the exhaustiveness of the provided information.

The research portion of this thesis focused mainly on the underlying principles of blockchain technology, and the specific implementational details of Ethereum. After establishing the Ethereum's features, existing solutions were examined to act as a possible reference point for the created application. In the design phase, used technologies, database, and underlying data representation was proposed with the focus on the speed of the data gathering process. Communication interface in the form of REST API endpoints was described as well, with the motivation to cover the majority of the possible use-cases.

The next section describes the details, decisions, and issues related to the implementation. The main issues that arose were related to the speed of gathering data from the Ethereum's blockchain, and to the functionality of the database. To fix the database issues, database implementation was switched from LevelDB to RocksDB. Slow data gathering speeds were solved by using the unofficial implementation of asynchronous RPC requests. Architectural changes had to be made as well due to the low speed of the database synchronization.

The next chapter describes the testing of the created application. The application shows satisfactory performance, executing simple queries almost instantaneously, and finishing even the larger, bulk queries in relatively short times. The database synchronization process is relatively quick, and the most time-consuming operation is waiting for the Ethereum RPC operations to be completed. As these functions are a part of Ethereum itself, so not much can be done to improve their performance.The application has the ability to track ERC-20 and ERC-721 tokens, as well as the internal transactions, both of which were an extension to the original design. As new blocks are added to the blockchain, the program is able to process them in real time and keep the database up to date.

One of the shortcomings of this work was a lack of large scale testing, as only a testnet node without internal transactions was fully processed. This was caused by time constrains, as well insufficient hardware resources. The synchronization process is quite memory intensive, so the reduction of RAM requirements could be improved in future works. As data gathering is still a difficult task in many cryptocurrencies, perhaps the design and the technologies used in this work can be utilized in the creation of similar projects.

Bibliography

[1] *Bitcoin Developer Guide*. [Online; navštíveno 17.11.2018].
Retrieved from: `https://bitcoin.org/en/developer-guide`

[2] *Bitcore Node*. [Online; navštíveno 30.12.2018].
Retrieved from: `https://github.com/bitpay/bitcore-node`

[3] *Block*. [Online; navštíveno 17.11.2018].
Retrieved from: `https://en.bitcoin.it/wiki/Block`

[4] *Blockchain Architecture*. [Online; navštíveno 17.11.2018].
Retrieved from: `https://www.pluralsight.com/guides/blockchain-architecture`

[5] *Controlled supply*. [Online; navštíveno 25.12.2018].
Retrieved from: `https://en.bitcoin.it/wiki/Controlled_supply`

[6] *Docker*. [Online; navštíveno 11.4.2019].
Retrieved from: `https://www.docker.com/`

[7] *Docker Compose*. [Online; navštíveno 11.4.2019].
Retrieved from: `https://docs.docker.com/compose/`

[8] *ERC20 Token Standard*. [Online; navštíveno 30.3.2019].
Retrieved from: `https://theethereum.wiki/w/index.php/ERC20_Token_Standard`

[9] *Ethereum ETL*. [Online; navštíveno 7.3.2019].
Retrieved from: `https://github.com/blockchain-etl/ethereum-etl`

[10] *go-ethereum*. [Online; navštíveno 31.12.2018].
Retrieved from: `https://github.com/ethereum/go-ethereum`

[11] *Insight*. [Online; navštíveno 30.12.2018].
Retrieved from: `https://insight.bitpay.com/`

[12] *Insight-api*. [Online; navštíveno 30.12.2018].
Retrieved from: `https://github.com/bitpay/insight-api`

[13] *Learn Everything About ERC20 Tokens: The Most Comprehensive Guide*. [Online; navštíveno 7.2.2019].
Retrieved from: `https://blockgeeks.com/guides/erc20-tokens/`

[14] *Proof of work*. [Online; navštíveno 17.11.2018].
Retrieved from: `https://en.bitcoin.it/wiki/Proof_of_work`

[15] *RocksDB*. [Online; navštíveno 21.3.2019].
Retrieved from: https://rocksdb.org/

[16] *RocksDB Tuning Guide*. [Online; navštíveno 24.3.2019].
Retrieved from:
https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

[17] *Smart Contracts: The Blockchain Technology That Will Replace Lawyers*. [Online; navštíveno 23.11.2018].
Retrieved from: https://blockgeeks.com/guides/smart-contracts/

[18] *SnakeViz*. [Online; navštíveno 22.4.2019].
Retrieved from: https://jiffyclub.github.io/snakeviz/

[19] *Standard Transactions*. [Online; navštíveno 25.12.2018].
Retrieved from:
https://bitcoin.org/en/developer-guide#standard-transactions

[20] *Transaction*. [Online; navštíveno 23.11.2018].
Retrieved from: https://en.bitcoin.it/wiki/Transaction

[21] *Web3.py*. [Online; navštíveno 31.12.2018].
Retrieved from: https://web3py.readthedocs.io/en/stable/

[22] *Welcome to python-rocksdb's documentation!* [Online; navštíveno 21.3.2019].
Retrieved from: https://python-rocksdb.readthedocs.io/en/latest/

[23] *Welcome to RocksDB*. [Online; navštíveno 28.4.2019].
Retrieved from: https://github.com/facebook/rocksdb/wiki

[24] *What is Ethereum Gas: Step-By-Step Guide*. [Online; navštíveno 26.12.2018].
Retrieved from:
https://blockgeeks.com/guides/ethereum-gas-step-by-step-guide/

[25] *What is Litecoin?* [Online; navštíveno 28.12.2018].
Retrieved from: https://blockgeeks.com/guides/litecoin/

[26] *What is Ripple and XRP*. [Online; navštíveno 27.12.2018].
Retrieved from: https://www.boxmining.com/what-is-ripple-and-xrp/

[27] *XRP Ledger Overview*. [Online; navštíveno 27.12.2018].
Retrieved from: https://developers.ripple.com/xrp-ledger-overview.html

[28] Buterin, V.: *On Slow and Fast Block Times*. [Online; navštíveno 26.12.2018].
Retrieved from:
https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/

[29] Buterin, V.: *Uncle incentivization*. [Online; navštíveno 26.12.2018].
Retrieved from: https:
//github.com/ethereum/wiki/wiki/Design-Rationale#uncle-incentivization

[30] Frankenfield, J.: *Bitcoin Cash*. [Online; navštíveno 28.12.2018].
Retrieved from: https://www.investopedia.com/terms/b/bitcoin-cash.asp

[31] Nakamoto, S.: *Bitcoin: A Peer-to-Peer Electronic Cash System.* [Online; navštíveno 17.11.2018].
Retrieved from: https://bitcoin.org/bitcoin.pdf

[32] Nezvisky, M.: *What You Need To Know About Bitcoin SV - The New Top 10 Cryptocurrency.* [Online; navštíveno 28.12.2018].
Retrieved from: https://cryptodaily.co.uk/2018/11/what-you-need-to-know-about-bitcoin-sv-the-new-top-10-cryptocurrency

[33] Nuvreni, J.: *The Bitcoin Network.* [Online; navštíveno 25.12.2018].
Retrieved from:
https://medium.com/coinmonks/the-bitcoin-network-6713cb8713d

[34] Ray, J.: *A Next-Generation Smart Contract and Decentralized Application Platform.* [Online; navštíveno 26.12.2018].
Retrieved from: https://github.com/ethereum/wiki/wiki/White-Paper

[35] Rutnik, M.: *What is Dash? — a short guide.* [Online; navštíveno 28.12.2018].
Retrieved from: https://www.androidauthority.com/what-is-dash-820943/

[36] Sanjay Ghemawat, J. D.: *LevelDB.* [Online; navštíveno 28.12.2018].
Retrieved from: https://github.com/google/leveldb

[37] Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151 (2014). 2014.

[38] Zheng, Z.; Xie, S.; Dai, H.; et al.: *An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends.* [Online; navštíveno 17.11.2018].
Retrieved from: https://ieeexplore.ieee.org/abstract/document/8029379

Appendix A

Contents of the CD

The enclosed CD contains the following files:

- doc/ - source files of this thesis
- app/ - source code of the created application
  - cfg/ - configuration of the application
  - src/ - Python source code
  - tests/ - unit tests and their resources
  - Dockerfile - Dockerfile for the containerized program
  - README.md - readme describing the installation process and usage
  - docker-compose.yml - file for orchestrating the Docker initialization
  - docker-run.sh - entrypoint for the Docker container
  - main.py - file for launching the application
  - requirements.txt - list of python requirements
  - run.sh - wrapper file for the application
  - tox.ini - file for running the style check, typing check, and tests
- thesis.pdf - PDF of this thesis

Appendix B

Examples of retrieved data

B.1 Block information

Information about a retrieved block. Only one transaction has been included for the sake of brevity.
Query:

```
curl 'http://localhost:5000/api/block/0x5718e3212cfbd3d472bd181ad10a0c
      10cffd45ab4f34d7a92ac2bcbc256bbe17'
```

Returned data:


```
{
  "difficulty": "90429758",
  "extraData": "0xd583010500865061726974798631 2e31332e30826c69",
  "gasLimit": "4707788",
  "gasUsed": "394284",
  "hash": "0
     x5718e3212cfbd3d472bd181ad10a0c10cffd45ab4f34d7a92ac2bcbc256bbe17",
  "logsBloom": "0
     x0400200020000010000000000000000000070000100000000000000000000
  0000000000020000000000000000000000000040000000000840010000000004
  8000000000000000000000000000000000000000000000000400000400000000 1
  0020000020440000000008000000800400000000000000000008000000000000000
  0008800000000000000000000800000008000000000000000000000008000800
  0000000000000001000100000000000000000000000000000040000000000000
  1000000200000000000000004000000000000000000000000002020000000000
  0000002000000000000000081000000800000000000000000000",
  "miner": "0xbe9f6b7b7f66bbe07ebd50354d2fb7bf6b320093",
  "nonce": "0x5fab0bb1083c7b0b",
  "number": "15011",
  "parentHash": "0
     xc76e49d2948349372cf9798a97205d415810b5c2dd4eb737b4ead7f679b9fedb",
  "sha3Uncles": "0
     x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
  "size": "1101",
  "timestamp": "1479763707",
  "totalDifficulty": "289784270153",
```

```
  "transactions": [
    {
      "blockHash": "0
         x5718e3212cfbd3d472bd181ad10a0c10cffd45ab4f34d7a92ac2bcbc256bbe17
         ",
      "blockNumber": "15011",
      "contractAddress": "",
      "cumulativeGasUsed": "98571",
      "from": "0x13c0127b66b336a644cc36c2e44eadc5fcd8b79d",
      "gas": "250000",
      "gasPrice": "20000000000",
      "gasUsed": "98571",
      "hash": "0
         xe7bb773d2a07df142ae91d7ee9e49021e4a417e0707fa8adf644a6e6fe113dfa
         ",
      "input": "0x51a34eb8000000000000000000000000000000000000000000
                00000000000850fc6266aec4000",
      "internalTransactions": [],
      "logs": [
        {
          "data": "0x0000000000000000000000000000000000000000000000000
                   0000850fc6266aec4000",
          "topics": [
            "0xa609f6bd4ad0b4f419ddad4ac9f0d02c2b9295c5e6891469055cf73c2b5
             68fff",
            "0x000000000000000000000000581318619cd8e37fa7dc5808e88e3b
             396a97fe8d0x000000000000000000000000581318619cd8e37fa7dc5
             808e88e3b396a97fe8d"
          ]
        }
      ],
      "nonce": "158",
      "timestamp": "1479763707",
      "to": "0x581318619cd8e37fa7dc5808e88e3b396a97fe8d",
      "value": "0"
    },
    .
    .
    .
  ]
}
```


B.2 Address information

The second example shows the retrieved information of an address. The address was involved both in regular and token transactions. Only some transactions are included in order to save space. As the database was build using a light geth node, internal transaction data

is not present.
Query:

```
curl 'http://localhost:5000/api/addr/0x4d3d0f631e4c3d2bff847c3f692
      b1d99eb69ce47'
```

Returned data:


```
{
  "balance": "1868697374000000000",
  "code": "0x",
  "inputInternalTransactions": [],
  "inputTokenTransactions": [],
  "inputTransactions": [
    {
      "blockHash": "0
         x088823148e2d7b463252b5f50533c512300d7e812c02a8f167d486bf34eff538
         ",
      "blockNumber": "2687498",
      "contractAddress": "",
      "cumulativeGasUsed": "1107479",
      "from": "0x687422eea2cb73b5d3e242ba5456b782919afc85",
      "gas": "314150",
      "gasPrice": "1000000000",
      "gasUsed": "21000",
      "hash": "0
         x4594bc7c26c2f67d389b05c0e9d21e805102034941882ea98e82a75762498866
         ",
      "input": "0x",
      "logs": [
        {
          "data": "",
          "topics": []
        }
      ],
      "nonce": "415950",
      "timestamp": "1519099066",
      "to": "0x4d3d0f631e4c3d2bff847c3f692b1d99eb69ce47",
      "value": "1000000000000000000"
    },
    .
    .
    .
  ],
  "mined": [],
  "outputInternalTransactions": [],
  "outputTokenTransactions": [
    {
      "addressFrom": "0x4d3d0f631e4c3d2bff847c3f692b1d99eb69ce47",
      "addressTo": "0x4ee9600513cfbb07b7c127e68af8fdf98bb64dbd",
```

```
    "timestamp": "1519245379",
    "tokenAddress": "0x9724a061eaff6e34f4127272bf84cab690f98339",
    "transactionHash": "0
       x1d804cb19bb3447b407ee99197d270037635de4dfa024adc25c7232504b6548f
       ",
    "value": "100000000000000000000000"
  },
  {
    "addressFrom": "0x4d3d0f631e4c3d2bff847c3f692b1d99eb69ce47",
    "addressTo": "0x899c688bf48f9fc8664d1599c6fd269fc0fdc972",
    "timestamp": "1519247508",
    "tokenAddress": "0xbcf4c93b791af3953a1b014b7518255ed0a22c6c",
    "transactionHash": "0
       xde08e0965df2a45e2eb78be5acabcf105b39326fddcafb4094cc31fe9380f7b4
       ",
    "value": "1000000000000000000000000000"
  },
  {
    "addressFrom": "0x4d3d0f631e4c3d2bff847c3f692b1d99eb69ce47",
    "addressTo": "0x4ee9600513cfbb07b7c127e68af8fdf98bb64dbd",
    "timestamp": "1519245459",
    "tokenAddress": "0x9724a061eaff6e34f4127272bf84cab690f98339",
    "transactionHash": "0
       x8de8b4dbadfffaa918b959b7499444bffcc227b7e4c47167806947f4664d93b8
       ",
    "value": "999000000000000000000000000"
  },
  {
    "addressFrom": "0x4d3d0f631e4c3d2bff847c3f692b1d99eb69ce47",
    "addressTo": "0x30e7cb3af3e74c1f9f838905a3ec578d9cdf993f",
    "timestamp": "1519253603",
    "tokenAddress": "0xbcf4c93b791af3953a1b014b7518255ed0a22c6c",
    "transactionHash": "0
       x4d39accc1475b9c0dfe9e97e3d40a960f65e3b46fd954a101ab3e2ee85d8346c
       ",
    "value": "250000000000000000000000000"
  }
],
"outputTransactions": [
  {
    "blockHash": "0
       xd7b9f3dfc23dbe07464ac3072e3e0783468b09c44084fd4e67822ab23df6a024
       ",
    "blockNumber": "2934147",
    "contractAddress": "",
    "cumulativeGasUsed": "4681429",
    "from": "0x4d3d0f631e4c3d2bff847c3f692b1d99eb69ce47",
    "gas": "21000",
```

```
      "gasPrice": "32000000000",
      "gasUsed": "21000",
      "hash": "0
         x1e1db30d641ad843a8fb24fbc3e697a9d4295845dc628d0cdb90602453484565
         ",
      "input": "0x",
      "logs": [
        {
          "data": "",
          "topics": []
        }
      ],
      "nonce": "15",
      "timestamp": "1522365171",
      "to": "0x1ab84fdbb25b8726f17d5c58ef9c1a0062abf77a",
      "value": "100000000000000000"
    },
    .
    .
    .
  ],
  "tokenContract": "False"
}
```