University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Processing Bitcoin Blockchain Data using a Big Data-specific Framework

Dominik Sommer
Zürich, Switzerland
Student ID: 13-927-454

Supervisor: Eder Scheid, Prof. Dr. Claudio Tessone
Date of Submission: May 2, 2019

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Die Analyse von Bitcoin Blockchain-Daten, wie zum Beispiel die Aggregation von Bitcoin-Adressen, ist interessant für Blockchain- sowie Wirtschafts- Forscher. Mit wachsender Grösse der Bitcoin Blockchain sind solche Analysen jedoch schwieriger geworden, weil die Laufzeiten der dafür benötigten Programme sehr lang sind. Im Gegensatz zu Lösungen, die in früheren Arbeiten erwähnt werden, zielt der in dieser Arbeit präsentierte Ansatz nicht nur darauf ab, die Laufzeit zu reduzieren, sondern *(i)* dies auf handelsüblicher Hardware zu bewerkstelligen, als auch *(ii)* zukünftigen Nutzern die Möglichkeit zu bieten, neue Heuristiken in einer einfachen Art und Weise zu implementieren. Dieser Ansatz beinhaltet die Bildung eines Transaktionsgraphen in der Graphendatenbank Neo4j aus den rohen Bitcoin-Blöcken. Auf diesem Transaktionsgraphen können dann in der Abfragesprache Cypher verfasste Heursitiken in einem Spark Cluster mit Hilfe von Cypher for Apache Spark (CAPS) angewendet werden. Die Nutzung von Cypher für die Implementierung der Heuristiken erlaubt es zukünftigen Nutzern eigene Heuristiken zu entwickeln, die die Graphstruktur der Blockchain-Daten nutzen. Auswertungen haben gezeigt, dass es praktikabel ist, die Bitcoin Blockchain-Daten zu parsen, einen Neo4j Transaktionsgraphen zu erstellen und diesen in die Neo4j Graphendatenbank zu importieren. Wegen eines Programmfehlers in CAPS konnte die in Cypher implementierte Multi-Input-Heuristik jedoch noch nicht in einem Spark Cluster angewandt werden. Ausserdem zeigt die vorliegende Untersuchung auf, dass die Anwendung der Heuristik direkt in Neo4j auf einem lokalen Rechner wegen zu langer Laufzeiten nicht praxistauglich ist. Zudem lieferte dieser Versuch den Nachweis, dass die Multi-Input-Heuristik viele Duplikate erzeugt, die wieder entfernt werden müssen. Dieser Umstand führte zu der Erkenntnis, dass, falls beim Entfernen von Duplikaten in einem verteilten System wie Apache Spark das Rechner Cluster nicht richtig genutzt werden kann, eine hochgradig optimierte lokale Lösung in Erwägung zu ziehen wäre. Gleichwohl ist der in dieser Arbeit präsentierte Ansatz, der Heuristiken in CAPS anwendet, vielversprechend, um Bitcoin Blockchain-Daten zu verarbeiten.

The analysis of Bitcoin blockchain data using heuristics, such as address clustering, is attractive for blockchain researchers and economics researchers because of the economical insights it can provide. Such analysis has become more challenging with increasing size of the Bitcoin blockchain because of its long processing times. In contrast to solutions mentioned in previous work, the approach presented in this thesis aims to reduce not only processing times but also *(i)* perform it on commodity hardware, and *(ii)* provide the possibility for users to implement new heuristics quickly. This approach includes the creation of a Transaction Graph in the graph database Neo4j from raw Bitcoin blocks, on which clustering heuristics written in the query language Cypher can be applied on a Spark cluster using Cypher for Apache Spark (CAPS). Employing Cypher for the implementation

of the heuristics allows users to write their heuristics that exploit the graph nature of the blockchain data. Evaluations have shown, that it is practicable to parse the Bitcoin blockchain, create a Neo4j Transaction Graph, and import it into Neo4j. However, due to software errors encountered during the employment of CAPS, the implemented Multi-Input heuristic in Cypher could not yet be applied on a Spark Cluster. Moreover, the application of the heuristic on a local machine directly in Neo4j has been proven to be not feasible for the complete blockchain because of long processing time. It could also be shown that the Multi-Input heuristic produces a considerable amount of duplicates which need to be removed. Thus, this lead to the conclusion that, if removing duplicates in a distributed system, *e.g.,* Apache Spark, cannot exploit the computing cluster, it might be worth considering a highly optimised local solution. Nevertheless, the approach presented in this thesis is promising to address the processing of this type of data, *i.e.,* Bitcoin transactions, with the employment of heuristics in CAPS.

Acknowledgments

First of all, I want to thank my supervisor Eder Scheid for supporting me during the process of writing the thesis with excellent feedback and suggestions. Furthermore, I want to thank Prof. Dr. Claudio Tessone for the helpful discussions about Bitcoin and address clustering. In addition, a thank to Kevin Primicerio who helped me a lot with answering my questions on the use of his Bitcoin blockchain library `bclib`.

Contents

Chapter 1

Introduction

The Bitcoin blockchain [1], introduced in 2009, is considered one of the first, if not the first, blockchain implementation. From its release to the public until now, this blockchain has reached, in size, more than 200 GB [2]. Moreover, with the increasing interest in cryptocurrencies by the general public, the Bitcoin blockchain size continues to grow. Thus, the size of this blockchain is becoming a challenge not only for peers to maintain but also for researchers to perform analytical tasks on such data without systems dedicated to process large amounts of data. Big data-specific frameworks, such as Apache Spark [3], offer the possibility to distribute workloads on a cluster of computers in order to achieve high performance, and hence, help reducing processing time. Therefore, the employment of these frameworks might help researchers to process and analyse the Bitcoin blockchain to find patterns and insights about transaction flows in the network.

There has been some effort to combine parallel processing with the Bitcoin blockchain. The authors of [4] utilise the Apache Spark framework, which is a unified analytic engine for big data processing, to provide an environment for researchers and developers to query data from the Bitcoin blockchain and to build analysis tools. Unlike in [4], the authors of [5] exploit parallel processing on a local machine and combine it with in-memory optimisations in their solution. Furthermore, in [6] a set of analyses of the Bitcoin blockchain is presented, including, user graphs, detection of the nodes which are critical for the network connectivity and economic analysis. However, these works do not rely on parallel processing to perform such analyses.

Among the different analysis topics in the Bitcoin ecosystem the aggregation of Bitcoin addresses that are controlled by the same user, also known as address clustering, has been a strong research topic since the beginning of the Bitcoin network. The authors of [7] explain how heuristics help to identify which addresses belong together. The insights gathered through address clustering can further be used to study the economics of the Bitcoin network or to analyse user privacy.

1.1 Description of Work

The goal of this thesis is to investigate how a graph database, such as Neo4j, in combination with a big data framework, such as Apache Spark, can help in the processing of data gathered from the Bitcoin blockchain. To guide the output of the processing, heuristics retrieved from the literature on Bitcoin blockchain analysis are employed. These heuristics are based on different characteristics of a Bitcoin transaction, such as input transactions, output addresses, and types of addresses.

1.2 Thesis Outline

The remainder of this thesis is structured as follows: In Chapter 2, information on the background of this thesis is provided. This information includes an overview of the Bitcoin blockchain focusing on details relevant for address clustering, and introduces Neo4j and Apache Spark. Then, in Chapter 3, related work in the field of Bitcoin blockchain analysis with a focus on address clustering and blockchain analysis tools is presented and discussed. Next, in Chapter 4, the developed Bitcoin address clustering solution is described in two steps. First, the solution architecture is presented and design decisions are explained. In the second step, the implementation of the solution is presented in detail. Further, in Chapter 5, the implementation is evaluated with regard to its performance and challenges faced during the development phase are discussed. Finally, in Chapter 6, the thesis is summarised presenting insights gathered throughout this work, and future research directions are listed.

Chapter 2

Background

This chapter describes the technologies that were employed in the development of the approach, such as the Bitcoin blockchain, Neo4j, and Apache Spark. Further sections describe these technologies in details.

2.1 Bitcoin

Bitcoin is a decentralised digital currency developed by an anonymous person (or group) of people under the pseudonym of Satoshi Nakamoto. This pseudonym is the author of the white paper publication entitled "Bitcoin: A Peer-to-Peer Electronic Cash System" released in 2008 [1]. However, the actual Bitcoin blockchain network started in 2009 with the reference implementation published by Nakamoto.

The following section explains the technical background of Bitcoin necessary to understand the later chapters of this thesis. The information presented in this section is based on the book "Mastering Bitcoin" [8].

2.1.1 Blockchain

The Bitcoin blockchain is the distributed ledger that holds all the verified Bitcoin transactions. It is an ordered list of blocks containing transactions. Each block in the list is linked to the previous block by including the identifier (*e.g.,* the block hash) of the previous block in its block header. The block hash of the previous block affects therefore the block hash of the current block. This inclusion of block hashes ensures that blocks cannot be changed. Because, to change one block, it is required to change all the following blocks in the blockchain, and convince all the peers in the network to accept the changes, which is a difficult task. Thus, by employing this concept, the blockchain provides immutability of the data.

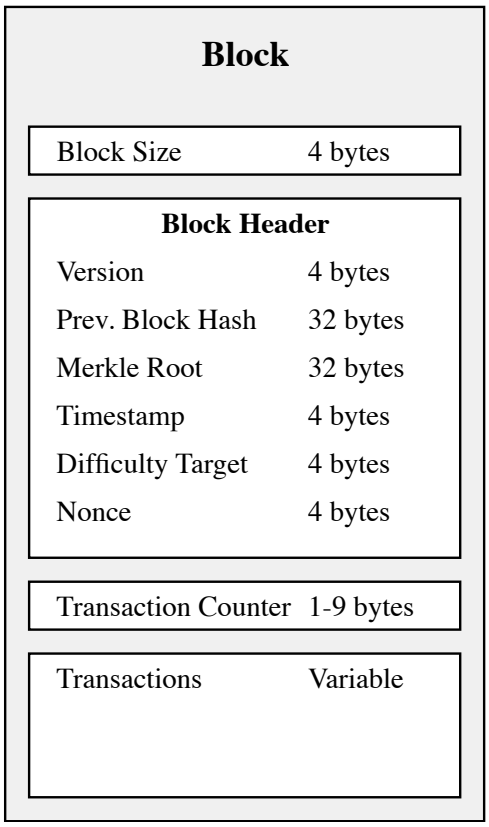


Figure 2.1: Bitcoin Block Example

Block

Figure 2.1 depicts the structure of a block in the Bitcoin blockchain. The *Block Size* field corresponds to the size, in bytes, of the whole block. The *Block Header* itself contains the metadata of the block. Its fields, such as *Version*, *Merkle Root*, and *Nonce*, are used to calculate the block hash. The calculation of this hash is performed by hashing the the *Block Header* twice using the SHA-256 algorithm. The result of this algorithm is a 32 byte block hash that uniquely identifies a certain block in the blockchain. Moreover, each block has a block height which indicates the distance of the block to the first block in the chain (*i.e.,* genesis block). However, the block height is not unique since two or more blocks can compete for a single position in the block chain during a fork in the chain. A detailed explanation of blockchain forks can be found in the "Mastering Bitcoin" book chapter "Blockchain Forks" [8].

Addresses

An address in the Bitcoin blockchain is a string of digits and characters that functions similarly to a bank account number. It can be shared with people to allow them to transfer an amount of Bitcoin to it. In contrast to a bank account numbers, Bitcoin addresses can be created autonomously and as many as desired. The Bitcoin Wiki even advises to use a unique address for each transaction [9]. In the Bitcoin *Mainnet* there are currently the following three address formats in use:

- **Pay-to-PubkeyHash (P2PKH)** which begins with the number **1**. For example, `1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2`. This type of address is the most common in the Bitcoin blockchain, they include a script that is resolved sending the public key and signature generated with a private key [10].

- **Pay-to-Script-Hash (P2SH)** which begin with the number **3**. For example, `3J98t1WpEZ73CNmQviecrnyiWrnqRhWNLy`. These addresses allow transactions to be sent to a script, and are only spent if the recipient provides a script that matches with the script hash and data which when input to the script evaluates to true [11].

- **Bech32** which begins with the **bc1** pattern. An example of such an address is `bc1qar0srrr7xfkvy5l643lydnw9re59gtzzwf5mdq`. Those are *segwit* addresses, the most recent Bitcoin address format [12].

All addresses that do not have one of the above mentioned formats are either:

- **Coinbase** is the input of the first transaction of each block. This input might contain arbitrary data, since these transactions only pay the block reward to the miner that created the block. It does not need an input transaction because, the Bitcoins used to pay the block reward are generated from nothing.

- **Invalid** formats have *unspendable* outputs which are called OP_RETURN outputs. These outputs can be used to burn Bitcoins or to store arbitrary data, *e.g.,* a string [13].

Transactions

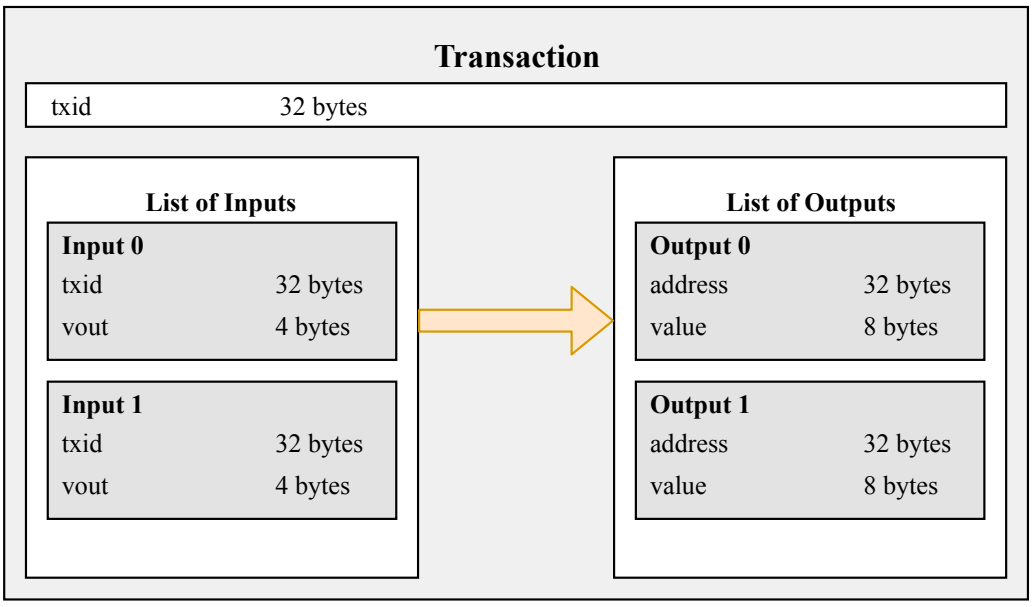


Figure 2.2: Simplified Bitcoin Transaction Example

Transactions are the data structures that hold the information about the transfer of Bitcoins from one or more addresses to one or more addresses. They are a core part of the Bitcoin blockchain. Figure 2.2 depicts a simplified transaction that contains only the data relevant for this thesis. However, there are more fields that should be included in a real-world Bitcoin transaction.

- **txid** identifies a transaction. It is generated by hashing the complete transaction data twice using the SHA-256 algorithm.
- **UTXO** is an unspent transaction output.
- **List of Inputs** contains the information about which UTXOs should be used as input in the transaction.
- Each **Input** references an UTXO by the corresponding txid and vout which indicates the index of the output in that transaction. Additionally, it includes an **Unlocking Script (scriptSig)** that satisfies the conditions set in the Locking Script of the UTXO and hence unlocking it.
- **List of Outputs** contains the information about how many Bitcoins are sent to which address.
- Each **Output** contains
  - **Locking Script (scriptPubKey)** from which the address mentioned in Figure 2.2 can be extracted.
  - **Value** which indicates how many Bitcoins, in satoshis (*i.e.,* smallest Bitcoin unit), are sent to the address in the Locking Script.

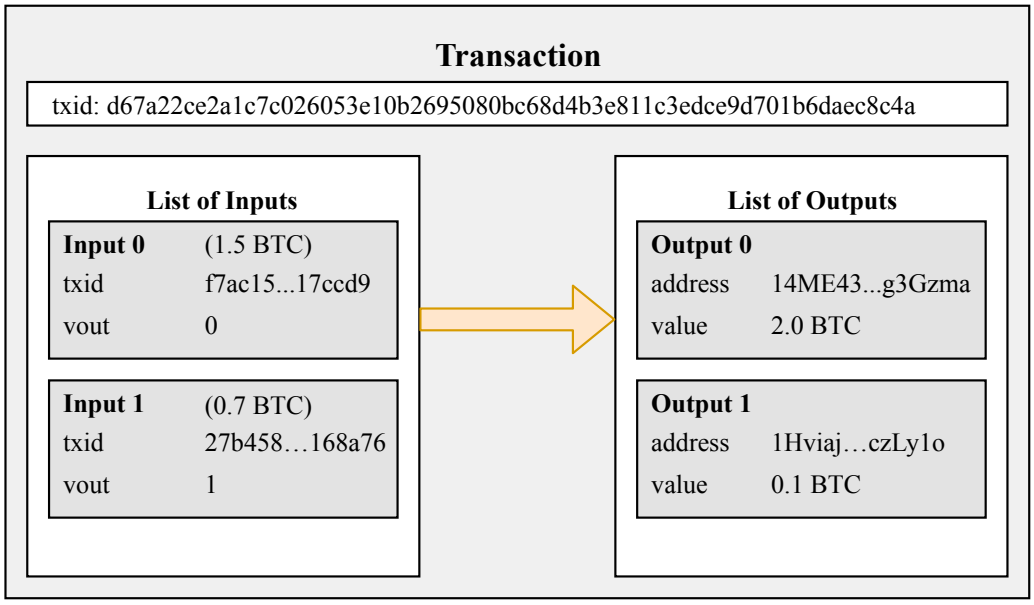


Figure 2.3: Transaction Example with Combination of Two Inputs

It is important to mention that an UTXO cannot be partially spent. Figure 2.3 depicts an example of a transaction where user **A** wants to send 2.0 BTC to user **B**. Since **A** does not have an UTXO with exactly 2.0 BTC, she uses two UTXOs that sum up to 2.2 BTC as transaction input and adds an output (*e.g.,* Output 1) for the change. The sum of both outputs in this example is 2.1 BTC and the difference between the sum of inputs and the sum of outputs — in this example 0.1 BTC — is called the **Transaction Fee**. This fee is used to compensate the miner for his/her work of including the transaction in the block and later the block in the blockchain.

2.1.2 Mining

Mining refers to the process of adding transactions to blocks and later blocks to the Bitcoin blockchain. In this process miners validate and include new transactions, which were propagated on the Bitcoin network, into new blocks independently. Then they solve a computationally intensive cryptographic problem to demonstrate the work that they have done — this is called the Proof-of-Work (PoW). If a miner has found a solution she propagates the newly created block on the Bitcoin network. If, however, a new block reaches the miner while she is working on a block, she will stop her work on the current block and start creating a new one [14].

A miner who was able to add a block to the blockchain is paid a block reward and all the transaction fees included in the block for securing the blockchain. To receive this funds the miner includes a coinbase transaction with his address (in the list of transaction outputs) as the first transaction in the block. Through these block rewards the mining mechanism also serves as the money supply of the Bitcoin system [14].

Proof-of-Work

A PoW is a piece of data that is difficult to produce but easy to verify. In the Bitcoin system the block hash serves as the PoW. The block hash is calculated by hashing all the data in the *Block Header* (see Figure 2.1) through SHA-256. To provide a valid PoW a miner calculates the block hash of his candidate block repeatedly while incrementing the nonce field in the Block Header until the block hash is smaller or equal to the difficulty target in the Block Header [15]. The PoW is employed as the consensus mechanism in the Bitcoin blockchain. It ensures that all the peers can trust in the mined blocks.

2.1.3 Bitcoin Core Client

The Bitcoin Core is an open source Bitcoin client based on the original Bitcoin client released by Satoshi Nakamoto. It consists of a software to maintain a *full node*, *i.e.,* maintain a full copy of the blockchain, as well as a Bitcoin wallet to sign transactions and send transactions to the blockchain [16].

Bitcoin Core stores the blocks, in network format, in `blk*.dat` files. The blocks are stored as soon as they are received. Thus, do not necessarily follow the order of the blocks [17]. Figure 2.4 illustrates the first block of the Bitcoin blockchain (*i.e.*, genesis block) in hexadecimal data format.

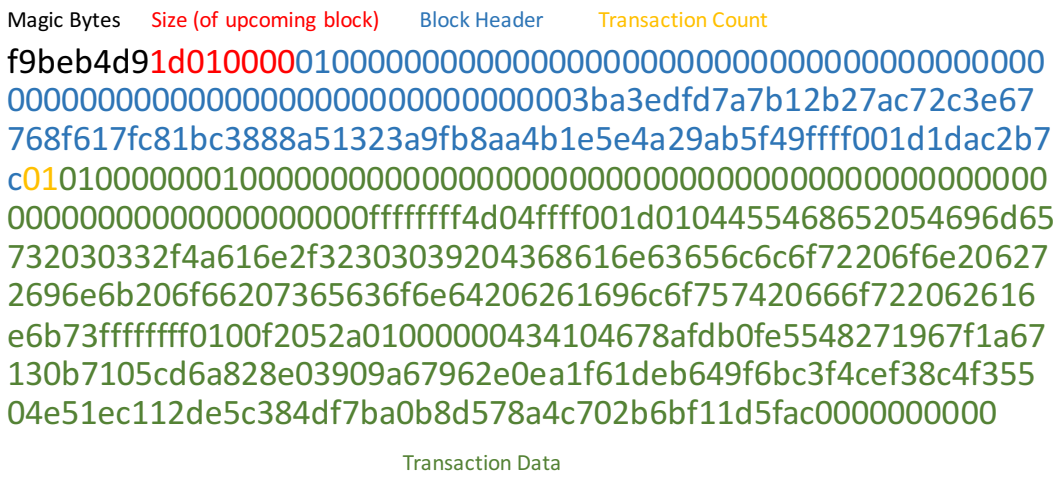


Figure 2.4: First 293 bytes of `blk00000.dat` (Genesis Block)

- The **Magic Bytes** indicate to which Bitcoin network a block belongs. There are four possibilities [18]:
  - **Mainnet**: `f9beb4d9`
  - **Testnet**: `fabfb5da`
  - **Testnet3**: `0b110907`
  - **Namecoin**: `f9beb4fe`
- The **Size** of the block in number of bytes.
- The **Block Header** contains the concatenated byte strings of the Block Header from Figure 2.1.
- The **Transaction Counter** tells the number of transactions included the block.
- **Transaction Data** contains the transactions included in this block. For detailed information see Bitcoin Wiki: Protocol Documentation [18].

2.2 Neo4j

Neo4j [19] is a transactional graph database developed by Neo4j, Inc. In contrast to SQL and NoSQL databases, Neo4j stores the data as a network of nodes and relationships between them, which is well suited for data that shows the characteristics of a network, such as Bitcoin transaction data. At the same time it also supports the ACID properties for consistent and reliable data. Neo4j has its own query language called Cypher. This

language allows to write complex queries in an easy way by using ASCII art to illustrate nodes and relationships.

The Neo4j Community Edition is freely available as GPLv3 licensed open-source project. The Neo4j Enterprise Edition that contains additional closed-source components requires a commercial license. Amongst other differences the two versions differ in graph size limitation. The Enterprise Edition has no graph size limitation, whereas the Community Edition is limited to 34 billion nodes/relationships [20]. Even though the Enterprise Edition runs under a commercial license, Neo4j provides Neo4j Desktop, that contains the full Enterprise Edition, for developers for free with registration.

2.3 Apache Spark

Apache Spark is a general-purpose cluster computing platform that promises high performance while being easy to use. Developed at UC Berkeley's AMPLab was it later donated to the Apache Software Foundation, which maintains it as an open-source project. It makes it easy to schedule and distribute computational tasks across many worker machines, *i.e.,* a cluster. To interact with the platform it provides APIs in Python, Java, Scala and SQL, as well as rich built-in libraries for varies purposes [21].

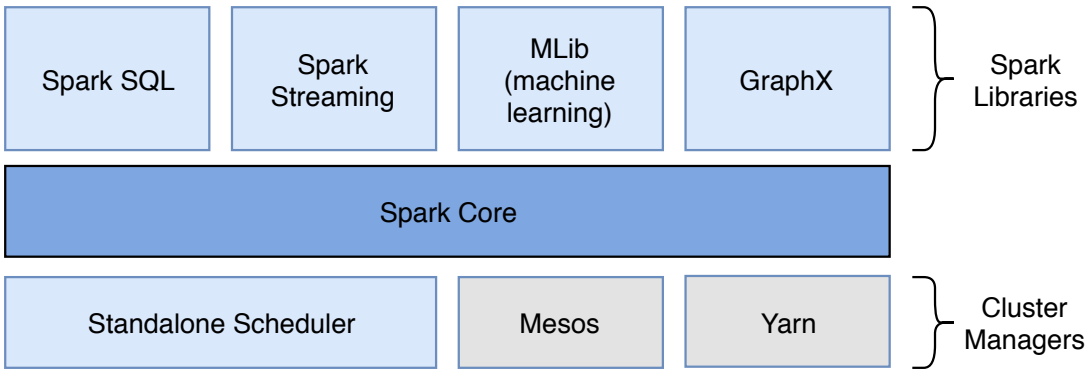


Figure 2.5: Components of Apache Spark

Figure 2.5 depicts the components present in the Apache Spark system. *Spark Core* provides basic functionality, such as task scheduling, memory management, and interaction with disk storage. On top of this component, Spark contains additional libraries, such as *Spark SQL* which allows to query structured data using SQL and Hive Query Language (HQL), *Spark Streaming* that enables processing of data streams, *MLib* that provides machine learning functionality, and *GraphX*, which is a graph engine that provides operators for graph manipulations and common graph algorithms. It is possible to use Spark with different cluster managers, *e.g.,* Apache *Mesos* or Hadoop *Yarn*. Nonetheless, Spark itself also features a cluster manager that allows to easily set up a cluster if there is not already an existing *Mesos* or *Yarn* cluster available [21].

Chapter 3

Related Work

Since Bitcoin started in 2009, there have been various publications on the subject of blockchain analysis. Thus, in this chapter an overview of related work in this context is provided. First, address clustering-related works are described. Then, tools that were developed to analyse blockchain data and which are closely related to the present work, are presented. Finally, a discussion is conducted on the performance evaluation and shortcomings of such works.

3.1 Address Clustering

A considerable number of works on the topic of Bitcoin blockchain analysis rely on a user graph that can be constructed using heuristics to cluster Bitcoin addresses, such as [7, 22–25]. Following this approach, a scalable clustering algorithm is presented in [26]. Harrigan and Fretter [27] explain why the Multi-Input heuristic (*cf.* Section 4.1.2) is effective. In contrast to the above mentioned works, [28] use off-chain data in addition to Blockchain-based heuristics for the clustering. Moreover, Chang and Svetinovic [29] combine the well known heuristics (*i.e.,* Multi-Input heuristic and Change heuristic) with heuristics based on transaction patterns characterised by [30].

3.2 Blockchain Analysis Tools

Over the last years in the topic of blockchain analysis, tools were developed to parse the Bitcoin blockchain and cluster addresses. Reid and Harrigan [22] in 2011, as well as Meiklejohn *et al.* [7] in 2013 use their own modified versions of the `bitcointools` Python library developed by Gavin Anderson [31] to parse the blockchain.

However, these approaches involved manual steps, such as the labelling of address clusters. Therefore, Spagnuolo *et al.* [32] presented BitIodine that automatically parses the blockchain, clusters addresses, classifies addresses and users, graphs, exports, and visualises information from the Bitcoin network. They use a modified version of `znort987`'s

C++ `blockparser` [33] for performance reasons and store the blockchain data in a embedded SQLite database for simplicity. To cluster addresses according to the Multi-Input heuristic and the shadow address heuristic they use NetworkX, a Python library for complex networks.

In 2015, Fleder *et al.* [24] state that newer Bitcoin Core clients use LevelDB for the blockchain indices making `bitcointools` obsolete. Instead, they use Armory [34] with additional wrapper classes to parse the blockchain and construct a transaction graph. Armory is a Bitcoin client mainly written in Python with a C++/SWIG backend for fast blockchain processing [35].

Because of the lack of open-source solutions for blockchain analysis which are able to deal with the increasing amount of data, Jeremy Rubin presented BTCSpark in 2015 [4]. BTCSpark is a layer on top of Apache Spark that allows to query the Bitcoin blockchain in a distributed fashion and to build distributed analysis tools.

Bartoletti *et al.* [36] proposed a general-purpose Scala framework for Bitcoin and Ethereum called Blockchain analytics API [37] in 2017. The tool first parses the blockchain and stores a view of it together with external data as desired in either a SQL (MySQL) or a NoSQL (MongoDB) database. The blockchain analysis is performed using the query language of the correspondent Database Management System (DBMS). The authors improved the solution by extending it to support the Litecoin blockchain as well as additional DBMS, *e.g.,* PostgreSQL and Fuseki.

In the same year Kalodner *et al.* [5] presented an open-source software platform for blockchain analysis called BlockSci. BlockSci supports different blockchains such as Bitcoin, Litecoin, Namecoin, and Zcash. According to Kalodner *et al.* , existing tools have three problems: *(i)* poor performance, *(ii)* limited capabilities, and *(iii)* a cumbersome programming interface. Thus, BlockSci addresses these weaknesses by being 15× to 600× faster, providing analysis tools such address clustering and two different interfaces, *i.e.,* Jupiter notebook and C++. In contrast to BTCSpark, BlockSci does not scale horizontally, because commodity cloud instances offer enough memory to load the complete Bitcoin blockchain in BlockSci's data format and analyse it in the foreseeable future. Moreover, the blockchain data's graph-structured nature makes it hard to partition the data effectively. Since parsing the blockchain needs to be sequentially to generate the BlockSci analysis format, the blockchain parser is single-threaded and highly optimised. To reduce analysis run-times BlockSci Analysis Library can be run multi-threaded.

Greg Walker has developed a tool called `bitcoin-to-neo4j` [38] that parses the Bitcoin blockchain and writes directly to Neo4j. It creates a graph of the complete blockchain. However, the resultant graph has 6× the size of the actual blockchain. Thus, if the Bitcoin blockchain has a size of 200 GB, the resultant graph will have 1.2 TB, which can present a challenge to store and process.

Peter Petkanic developed an approach in his bachelor thesis called "Bitcoin Blockchain Analysis" [39]. In the approach, first the Bitcoin blockchain is parsed using a *Go* parser that writes the whole blockchain to *JSON* files. Then a clustering software clusters the addresses, and writes nodes and relationships to *csv* ready to import them into Neo4j. Finally, these csv files are imported using the Neo4j import tool to be processed later on.

3.3 Discussion

3.3.1 Performance Evaluation

Some of the above mentioned papers [7, 22, 24] do not present information on the performance of the used tools. [32] contains a performance evaluation section for BitIodine. However, it does neither mention the size or height of the blockchain nor the date at which it was parsed. Hence, it is hard to compare the evaluation with other works. Kalodner *et al.* [5] compare BlockSci's performance with previous tools. To perform a fair as possible comparison, they utilize the same hardware for both tools whenever possible. The used setup was configured as follows:

- AWS EC2 instance with 8 vCPU at 2.5 GHz, Intel Xeon E5-2670v2, 61 GB memory, and 160 GB Storage Capacity.
- Blockheight 478,559, which is equal to approximately 128 GB of Bitcoin data.

On a single EC2 instance BlockSci executes the Total Output Amount Distribution (TOAD) query in 28.3 seconds, whereas BTCSpark [4] takes 3.7 minutes with 10 AWS EC2 m3.large(6.5 ECUs, 2 vCPUs at 2.5 GHz, Intel Xeon E5-2670v2, 7.5 GB memory, 32 GB of storage) and a blockheight of around 390,000. Thus, it can be observed that BlockSci is substantially faster than BTCSpark.

Möser and Böhme use Neo4j for their blockchain analysis [40, 41]. Kalodner *et al.* execute three queries on the Neo4j database they received from Möser and Böhme on the single AWS EC2 instance mentioned above and compare it to the execution of the same three queries with BlockSci in multi-threaded mode. They observe that BlockSci is 279× to 600× faster than the analysis with Neo4j.

For their comparison with `blockparser` [33] Kalodner *et al.* run its Simple Stats benchmark (*i.e.,* computing average input count, average output count and average value) on their EC2 instance and compare it with a single-threaded and a multi-threaded implementation of this benchmark for BlockSci. BlockSci's single-threaded version is 39× faster and the multi-threaded version is 1319× faster than `blockparser`.

3.3.2 General Remarks

There have been several proposed solutions to address the problem of processing the Bitcoin blokchain data. Some of them rely on `bitcointools` as a basis. However, this tool seems to be outdated if compared to recent works. Others, such as Blockchain analytics API [37], have long query creation times (*cf.* [5]), which hinders its practical employment in scientific research. There are also tools, such as `znort987`'s `blockparser`, that do not take advantage of parallel processing, *e.g.,* multi-threading or distributed computing. Thus, resulting in slower processing performance. Moreover, even BlockSci, that does not suffer from any of the above mentioned problems, has a downside, which is

that to run BlockSci, nowadays and in future, it needs hardware that is equipped with a considerable amount of RAM (*e.g.,* 61 GB). This RAM constrain cannot be easily solved. Therefore, it limits the employment of BlockSci in projects.

Based on this discussion, there is space for a solution that not only *(i)* allows to process Bitcoin data for address clustering in a distributed fashion on a cluster of commodity hardware while allowing to create or change heuristics easily, but also *(ii)* allows to combine different heuristics and compare different combinations of them.

Chapter 4

Processing Bitcoin Data

This chapter details the design and implementation of the proposed solution that allows to cluster Bitcoin addresses taking advantage of distributed processing. First, in Section 4.1, the design of the complete solution is described, which includes an overview of the architecture, an explanation of two address clustering heuristics, and design decisions concerning the chosen database. Then, in Section 4.2, the implementation of the different modules of the proposed solution is explained in detail.

4.1 Solution Design Decisions

4.1.1 Solution Architecture

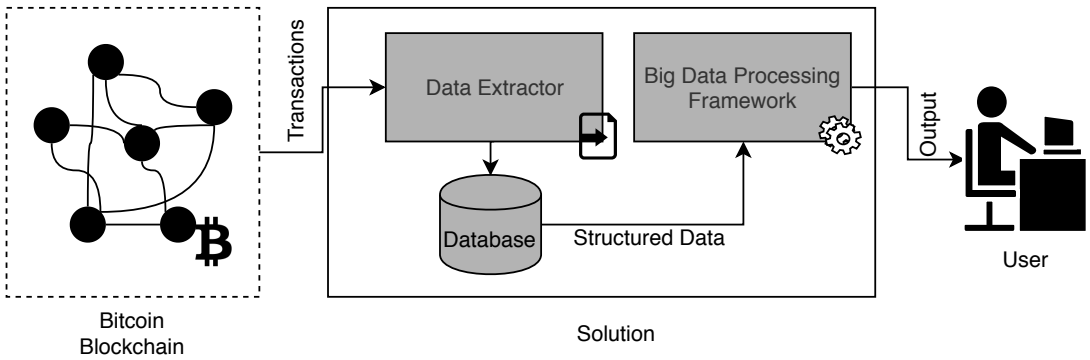


Figure 4.1: Foreseen Architecture of the Solution

Figure 4.1 depicts an overview of foreseen components of the solution architecture. The *Data Extractor* obtains the necessary data, such as transactions and blocks, from the Bitcoin blockchain and stores this information using the specific format schema of the *Database*, *e.g.,* a Neo4j graph. After all the data is stored, the *Big Data Processing Framework* retrieves this structured data and applies the algorithms and heuristics (*e.g.,* Multi-Input heuristic) to output useful information about the blockchain data to the *user*

and store the output in the *Database*. A *user* in the architecture solution is considered to be a blockchain or economics researcher that is interested in the output of the solution.

4.1.2 Address Clustering Heuristics

To provide valuable output for the user, address clustering heuristics, such as Multi-Input and Change, must be applied on the imported data in the Big Data Processing Framework. These heuristics are described in the next sections.

Multi-Input Heuristic

The Multi-Input heuristic was already described in Nakamoto's whitepaper [1] and explained in more detail in [7] as follows:

If two (or more) addresses are inputs to the same transaction, they are controlled by the same user; i.e., for any transaction t, all $pk \in inputs(t)$ are controlled by the same user.

Figure 4.2 shows a simplified Bitcoin transaction with two addresses as input (**A1** and **A2**), and two addresses as output, (**A3** and **A4**). The Multi-Input heuristic declares that the two input addresses, **A1** and **A2**, belong to the same user.

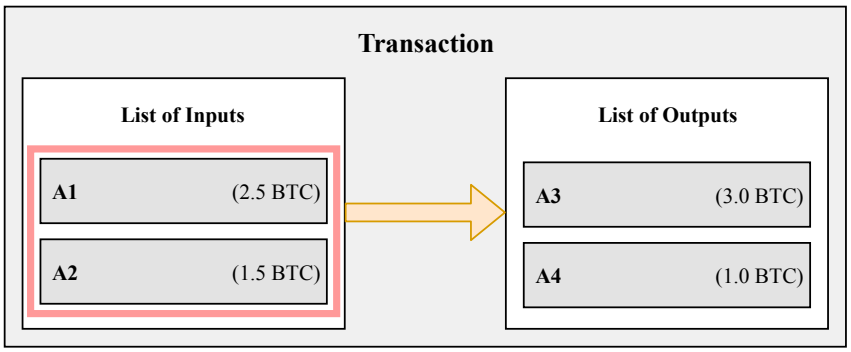


Figure 4.2: Transaction where the Multi-Input heuristic is applied

The Multi-Input heuristic does not always hold because of two reasons: The first being that partially signed transactions are allowed since BIP174 (Bitcoin Improvement Proposal 174), and also before BIP174 it was possible to give a private key to someone else in order to create a transaction with inputs from multiple parties. And the second is that it is possible to obfuscate payments by creating CoinJoin transactions [42].

However, the Multi-Input heuristic seems to be effective as it has been shown in an experimental analysis that the heuristic could identify more than 69% of the addresses in the wallets stored by lightweight clients [27].

Change Heuristic

The Change heuristic is based on the fact that several Bitcoin clients automatically generate new addresses for the change of transactions, referred to as *change addresses*. These addresses are only used once to receive change and spend it later. Since this occurs inside the wallet client, it is unlikely that a user will share a *change address* with other peers to receive payments.

Because the Change heuristic relies on a programming logic of Bitcoin clients instead of a property of the Bitcoin protocol, it application may lead to falsely linked addresses. Therefore, it is important to carefully decide whether an output address is a *change address* or not. Thus, Meiklejohn *et al.* proposed the following definition [7]:

DEFINITION *A public key pk is a one-time change address for a transaction t if the following conditions are met*:

1. $\mathrm{d}^{+}_{addr}(pk) = 1$; *i.e.,* this is the first appearance of pk.
2. The transaction t is not a coin generation.
3. There is no $pk' \in \text{outputs}(t)$ such that $pk' \in \text{inputs}(t)$; *i.e.,* there is no self-change address.
4. There is no $pk' \in \text{outputs}(t)$ such that $pk' \neq pk$ but $\mathrm{d}^{+}_{addr}(pk') = 1$; *i.e.,* for all the outputs in the transaction, condition 1 is met for only pk.

The definition of the one-time change address is then used in the Change heuristic as follows:

The one-time change address is controlled by the same user as the input addresses, i.e., for any transaction t, the controller of inputs(t) also controls the one-time change address pk in outputs(t) (if such an address exists).

Figure 4.3 shows a simplified Bitcoin transaction with one address as input, *i.e.,* **A1**, and two addresses as output, *i.e.,* **A2** and **A3**. Since **A1** is a valid address, the transaction is not a coinbase transaction. The Change heuristic says now that the two input addresses, **A1** and **A3**, belong to the same user.

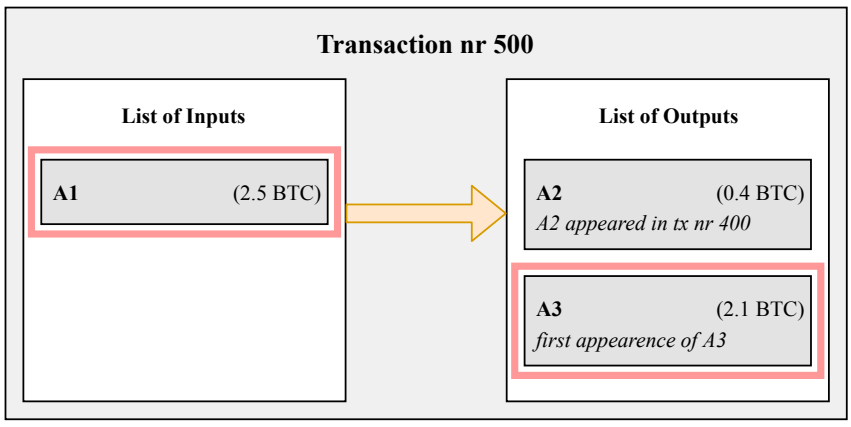


Figure 4.3: Transaction where the Change heuristic is applied

4.1.3 Address Graph

The result of applying an address clustering heuristic, such as the above mentioned Multi-Input heuristic, is that we know which addresses are connected. To be able to reason about the result, it is useful to picture it as a graph, the *Address Graph*, where addresses are represented by vertices and edges between two vertices indicate that they are connected. The connected components of an *Address Graph* represent clusters of addresses, whereby the addresses of one cluster belong to one user. The *Address Graph* presented in Figure 4.4 has three connected components depicted using green, blue, and red squares. The addresses of the green connected component (*i.e.,* **A1**, **A3**, **A4** and **A8**) belong to **User1**. Whereas **A2**, **A6** and **A7** are controlled by **User2**. And **User3** has only one address, **A5**.

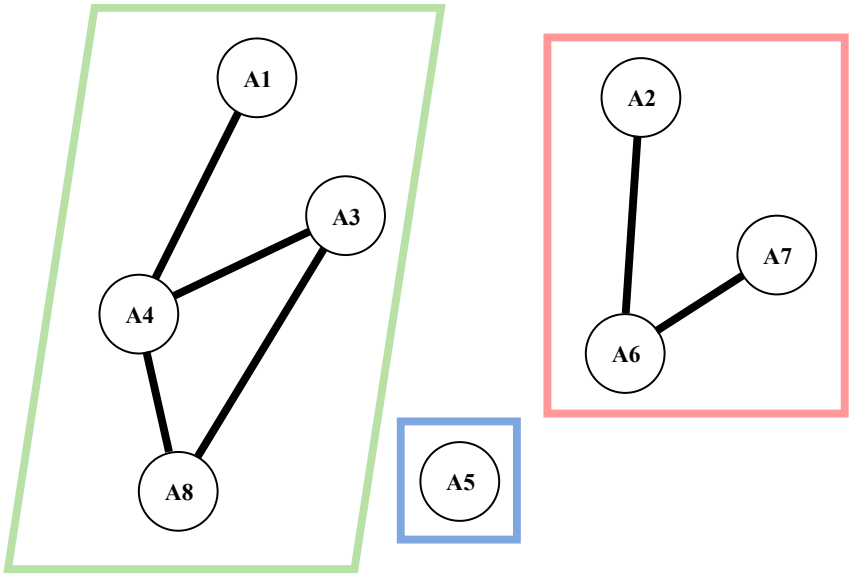


Figure 4.4: Address Graph with three connected components

4.1.4 Database

As we started thinking about how to store the data we would get after clustering the addresses, we realised that it is possible to store that data directly as a graph, the *Address Graph*, such that we could later on detect the connected components. Graph databases allow to store data directly as graphs. Neo4j is the most popular graph database [43] and it is freely available. Therefore, it is relatively well documented and discussed in forums. Another advantage of using Neo4j is that it provides a variety of graph algorithms, such as one to detect strongly connected components [44]. Those are the reasons why Neo4j was chosen as a database for this solution.

The more it was researched about the information necessary to apply the address clustering heuristics, the more it was realised that this data also is of a graph nature. Neo4j's query language Cypher allows to implement address clustering heuristics, that exploit the structure of the Bitcoin blockchain data, easily. It is even possible to apply Cypher queries

on a Neo4j graph in a distributed fashion with Cypher for Apache Spark (CAPS). This lead to the decision to store the necessary blockchain data in Neo4j. Figure 4.5 shows the database schema used to store the Bitcoin transaction data, *e.g.,* the *Transaction Graph*. This schema is composed of following nodes:

- **Transaction**: This node represents a Bitcoin transaction which can be identified through its transaction id (`txid`).
- **Block**: This node represents a block of the Bitcoin blockchain. It can be identified by its block hash (`hash`). It also stores the `height` in which this block can be found in the blockchain.
- **Address**: This node represents an address from the Bitcoin blockchain. It can be identified through the `address` property.

The nodes mentioned above are connected through the following relationships:

- A Block **IS_BEFORE** another Block.
- A Transaction **BELONGS_TO** a Block.
- An Address that is the input of a Transaction **SENDS** an amount of bitcoin in satoshis (`value`) to that Transaction.
- An Address that is the output of a Transaction **RECEIVES** an amount of bitcoin in satoshis (`value`) from that Transaction. The **RECEIVES** relationship also has a property `output_nr`.

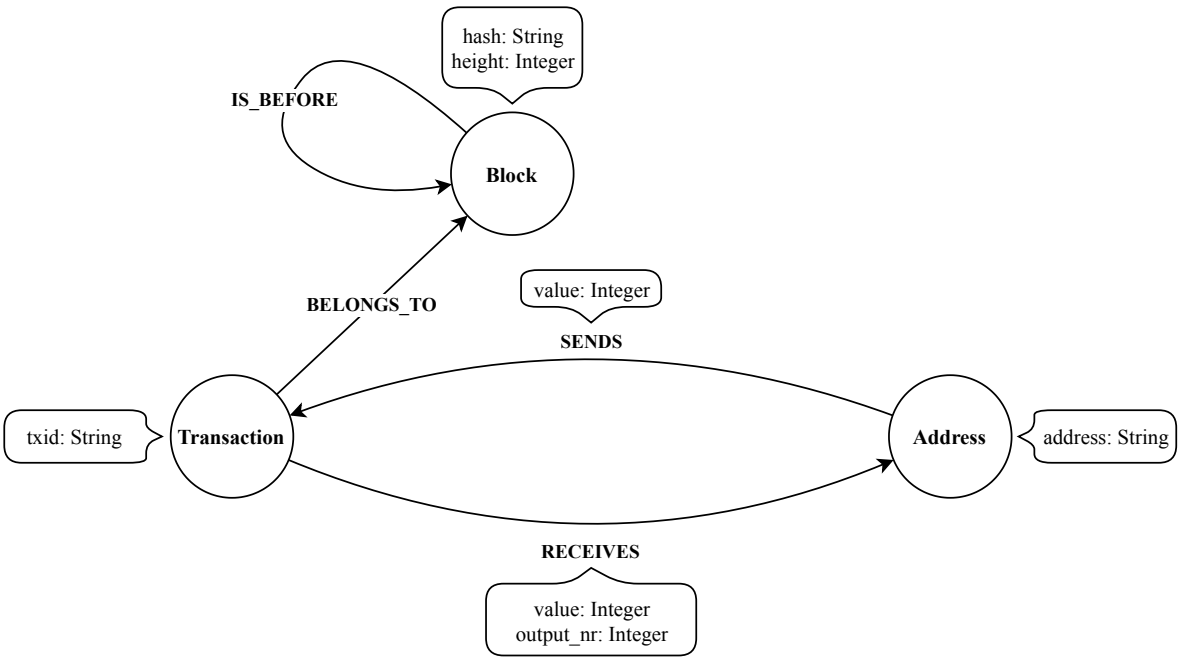


Figure 4.5: Database Schema for the Transaction Graph

4.2 Implementation

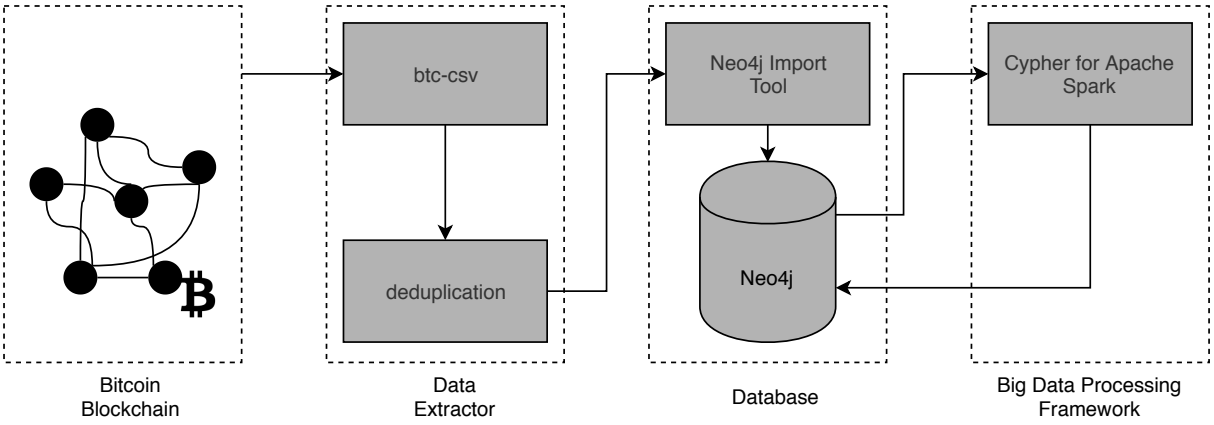


Figure 4.6: Data Flow of the Solution Implementation

Figure 4.6 shows the phases of data flow of the solution implementation. The *Data Extractor* phase consists of two steps: first, `btc-csv`, a program written in Go, parses the Bitcoin blockchain and creates csv files containing nodes and relationships (see Data Import Method 4.2.2). In a second phase, duplicate Address nodes are removed from the csv file using a process called `deduplication`. Thirdly, in the *Database* phase, the previously created nodes and relationships are imported into the Neo4j graph database using the `Neo4j Import Tool`. Finally, in the Big Data Processing phase, relying on CAPS, a set of heuristics is applied taking advantage of the distributed processing provided by Apache Spark and the result is stored in the database for future retrieval by the user.

4.2.1 Data Extractor

To interact with the Bitcoin blockchain the `chainswatch/bclib` [45] library was employed. `bclib` is a Bitcoin blockchain library developed by Kevin Primicerio using the Go language. It provides a set of functions and models, such as the function `LoadFile` (see Listing 4.1) and the struct models for transactions and block headers (see Listing 4.2), that allow to parse the Bitcoin blockchain.

Listing 4.1: LoadFile Function in bclib Library

```
1 // LoadFile allows to traverse the blocks by height order while applying a
    function argFn
2 func LoadFile(fromh, toh uint32, newFn apply, argFn interface{}) error {...}
```

The function `LoadFile` parses the blockchain from blockheight `fromh` to blockheight `toh`. For each block in the blockchain that is decoded, a data structure `Block` (see Listing 4.2) is created and filled with the decoded information and the function `fn` is called with the `Block struct` as parameter. The function `fn(b *models.Block)` is returned by `argFn` which is a parameter of `LoadFile`.

Listing 4.2: Implemented Models in bclib Library

```
// TxInput holds tx inputs
type TxInput struct {
    Hash         []byte `db:"hash"`   // Hash previous tx
    Index        uint32 `db:"index"`  // Output previous tx
    Script       []byte `db:"script"` // Not used
    Sequence     uint32 `db:"sequence"` // Always 0xFFFFFFFF
    ScriptWitness [][]byte
}

// TxOutput holds tx outputs
type TxOutput struct {
    Index uint32 `db:"index"` // Output index
    Value uint64 `db:"value"` // Satoshis
    Addr     []byte `db:"addr"` // Public key
    AddrType uint8
    Script   []byte `db:"script"` // Script Code
}

// Tx holds transaction
type Tx struct {
    NVersion int32 `db:"n_version"` // Always 1 or 2
    Hash     []byte `db:"tx_hash"` // Transaction hash (computed)
    NVin     uint32 `db:"n_vin"`   // Number of inputs
    NVout    uint32 `db:"n_vout"`  // Number of outputs
    Vin      []TxInput
    Vout     []TxOutput
    Locktime uint32 `db:"locktime"`
    Segwit     bool
}

// BlockHeader contains general index records parameters
// It defines the structure of the postgres table
type BlockHeader struct {
    NVersion       uint32 `db:"n_version"`      // Version
    NHeight        uint32 `db:"n_height"`       //
    NStatus        uint32 `db:"n_status"`       //
    NTx            uint32 `db:"n_tx"`           // Number of txs
    NFile          uint32 `db:"n_file"`         // File number
    NDataPos       uint32 `db:"n_data_pos"`     // (Index)
    NUndoPos       uint32 `db:"n_undo_pos"`     // (Index)
    Hash           []byte `db:"hash_block"`     // current block hash (Added)
    HashPrev       []byte `db:"hash_prev_block"` // previous block hash (Index)
    HashMerkleRoot []byte `db:"hash_merkle_root"` //
    NTime          uint32 `db:"n_time"`         // (Index)
    NBits          uint32 `db:"n_bits"`         // (Index)
    NNonce         uint32 `db:"n_nonce"`        // (Index)
    TargetDifficulty uint32 `db:"target_difficulty"` //
    NSize          uint32 `db:"n_size"`         // Block size
}
```

```
50
51 // Block contains block infos
52 type Block struct {
53     BlockHeader
54     Txs []Tx
55 }
```

In addition, `bclib` contains several functions to extract and calculate information related to the Bitcoin blockchain, such as the function `putTxHash` presented in Listing 4.3 to calculate the hash of a transaction (*reversed* txid). This function produced a wrong hash in some transactions that contained thousands of inputs or outputs. This error was encountered while parsing the blockchain data. The error was located, after some research by Kevin Primicerio and the authors, in the `parser.CompactSize(n uint64) []byte` function, which converts an *integer* to an array of 1 to 8 bytes. This error was fixed in a new version of the library.

Listing 4.3: Function that calculates the transaction hash (*reversed* txid) and adds it to the transaction

```
1  func putTxHash(tx *models.Tx) {
2      bin := make([]byte, 0)
3      version := make([]byte, 4)
4      binary.LittleEndian.PutUint32(version, uint32(tx.NVersion))
5      bin = append(bin, version...)
6
7      vinLength := parser.CompactSize(uint64(tx.NVin))
8      bin = append(bin, vinLength...)
9      for _, in := range tx.Vin {
10         bin = append(bin, getInputBinary(in)...)
11     }
12
13     voutLength := parser.CompactSize(uint64(tx.NVout))
14     bin = append(bin, voutLength...)
15     for _, out := range tx.Vout {
16         bin = append(bin, getOutputBinary(out)...)
17     }
18
19     locktime := make([]byte, 4)
20     binary.LittleEndian.PutUint32(locktime, tx.Locktime)
21     bin = append(bin, locktime...)
22
23     tx.Hash = serial.DoubleSha256(bin)
24 }
```

btc-csv

`btc-csv` parses the Bitcoin blockchain using `bclib` and generates the Transaction graph shown in Figure 4.5. As mentioned in Section 2.1.1, a Bitcoin transaction input does not

contain the address but a link to a previous UTXO. To resolve the address for a transaction input a `leveldb` [46] (a fast persistent key-value store) instance is used to maintain a key-value store of all the UTXO at the current block height during the parsing. For each transaction output an entry is added in the UTXO `leveldb`, called `utxo`. As shown in Listing 4.4, the key of the new entry is a string constructed of the txid and the output number of the UTXO. The value of the entry is the address and the amount of bitcoin sent. For each transaction input the `utxo` is queried to retrieve the corresponding address and amount of bitcoin. Then the entry in `utxo` is deleted to make look-ups faster.

Listing 4.4: New UTXO is added to leveldb instance called `utxo`

```
txOutput.Addr = addr
txOutput.Value = vout.Value
data, err := proto.Marshal(txOutput)
if err != nil {
        log.Fatal("marshalling error: ", err)
}
utxo.Put([]byte(txid+fmt.Sprint(vout.Index)), data, nil)
```

To create the Transaction Graph, the above mentioned function `LoadFile` from `bclib` is called with the function `Build(x interface{})` presented in Listing 4.5. The function returned by `Build(x interface{})` creates nodes and relationships for a block and writes them to the corresponding csv files.

Listing 4.5: Function that builds the Transaction Graph

```
func Build(x interface{}) (func(b *models.Block) error, error) {
    // Omitted code
    return func(b *models.Block) error {
        // Here are the nodes and relationships for the current block
        // created.
    }
}
```

If the blockchain shall parsed from a certain block height $b > 0$, it is important provide the `utxo` leveldb for block height $(b-1)$. If it is missing, then the input addresses cannot be resolved, and hence the program fails. If the newly created nodes and relationships shall be added to en existing Transaction Graph (see the Load CSV function in Section 4.2.2), it is important to either rename the existing csv files or move it to another folder. Otherwise, the new nodes and relationships are added to the existing csv files.

Deduplication

To avoid maintaining a second leveldb key-value store for addresses, all addresses are directly written to the address node csv file. This leads to duplicate Address nodes. Since removing duplicate nodes makes the Neo4j import slow, it is important to remove them in advance.

The deduplication of the address nodes is achieved by sorting and omitting duplicates in the address.csv file. Listing 4.6 shows the command used to remove duplicate address nodes. After `-T` a folder with enough space to hold the temporary data shall be specified. This is needed if the Linux root partition, *e.g.,* the partition where the operating system resides, is not large enough.

Listing 4.6: Deduplication using Linux sort

```
1 user@hostname:~$ sort -u addresses.csv -o addresses.csv -T /path/to/folder
```

4.2.2 Data Import Method

To import the Bitcoin data to the Neo4j database three options are available, *(i)* Neo4j Go Driver, *(ii)* Neo4j Import Tool, and *(iii)* Load CSV.

i The **Neo4j Go Driver** [47] is the officially supported tool to interact with the Neo4j database from within a Go program. It allows to execute Cypher queries to either add nodes and/or relationships to the database or just getting information from the database. One can refer to [48] for a more detailed description.

ii The **Neo4j Import Tool** [49] can be used to import large amounts of data in a batch. For each sort of node and relationship at least one CSV file needs to be provided. The big advantage of the Import Tool is that it works in parallel.

iii **Load CSV** [50] is a statement of Neo4j's query language Cypher. In contrast to the Import Tool it allows to add nodes and relationships from CSV files to an existing database.

In this work the Neo4j Import Tool was chosen because a first implementation using the Neo4j Go Driver has shown that it is too slow, and hence not usable to import the complete Bitcoin blockchain. Load CSV was not used because Neo4j suggests to use the Neo4j Import Tool for large amounts of data.

To be able to import the data using the Neo4j Import Tool, the following *csv-header* files containing these exact information need to be provided:

- **addresses-header.csv:**
  address:ID(Address)
- **blocks-header.csv:**
  hash:ID(Block),height:int
- **transactions-header.csv:**
  txid:ID(Transaction)
- **before_rel-header.csv:**
  :START_ID(Block),:END_ID(Block)

- **belongs_to_rel-header.csv:**
  :START_ID(Transaction),:END_ID(Block)

- **receives_rel-header.csv:**
  :START_ID(Transaction),value,output_nr:int,:END_ID(Address)

- **sends_rel-header.csv:**
  :START_ID(Address),value,:END_ID(Transaction)

The csv files holding the data need to be structured in the way of the corresponding header file, *e.g.,* the addresses.csv file needs to be structured as the addresses-header.csv. The nodes and relationships can be imported into Neo4j using the import script shown in Listing 4.7.

Listing 4.7: Import Script

```
export DATA=/path/to/folder/containing/csv-files/
export HEADERS=/path/to/folder/containing/csv-headers/

./bin/neo4j-admin import \
    --mode=csv \
    --database=btc.db \
    --nodes:Address $HEADERS/addresses-header.csv,$DATA/addresses.csv \
    --nodes:Block $HEADERS/blocks-header.csv,$DATA/blocks.csv \
    --nodes:Transaction
        $HEADERS/transactions-header.csv,$DATA/transactions.csv \
    --relationships:IS_BEFORE
        $HEADERS/before_rel-header.csv,$DATA/before_rel.csv \
    --relationships:BELONGS_TO
        $HEADERS/belongs_to_rel-header.csv,$DATA/belongs_to_rel.csv \
    --relationships:RECEIVES
        $HEADERS/receives_rel-header.csv,$DATA/receives_rel.csv \
    --relationships:SENDS $HEADERS/sends_rel-header.csv,$DATA/sends_rel.csv \
    --ignore-missing-nodes=true \
    --ignore-duplicate-nodes=true \
    --multiline-fields=true \
    --high-io=true
```

However, if new data is to be added to an existing Transaction Graph, *e.g.,* nodes and relationships from newly parsed blocks, then the Load CSV method must be used. Listing 4.8 shows an example of how to import new address nodes.

Listing 4.8: Load CSV addresses.csv

```
USING PERIODIC COMMIT 1000
LOAD CSV FROM '/path/to/addresses.csv' AS line
CREATE (:Address { address: line[1]})
```

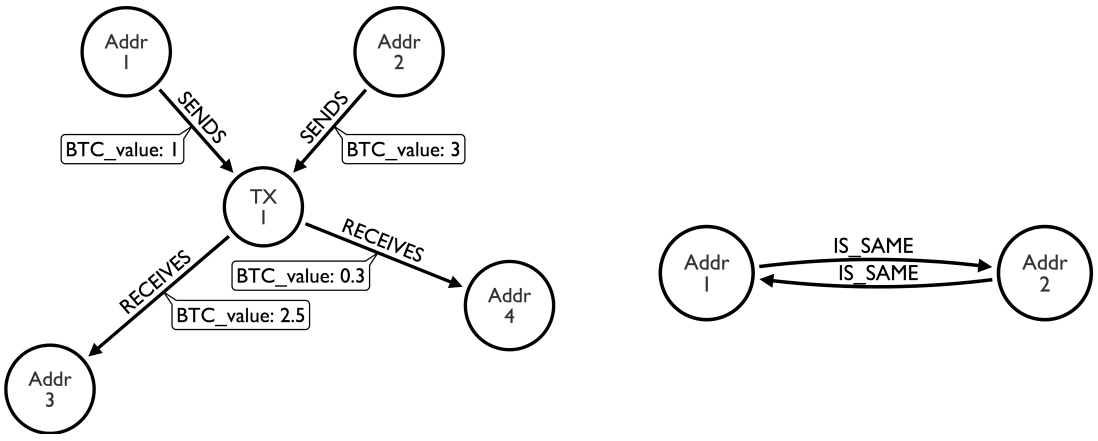
4.2.3 Distributed Data Analysis

The Multi-Input heuristic was implemented using **Cypher for Apache Spark** (CAPS) [51]. CAPS allows to use Cypher in a Apache Spark context. It gives the opportunity of integrating different data sources, such as Neo4j graphs and graphs stored as csv files, as well as it supports queries that involve multiple graphs at once. It allows to run Cypher graph queries on a Spark cluster. CAPS is maintained by the openCypher project which also maintains the Cypher language [51]. The Cypher language will also become an integral part of a new graph module in Apache Spark 3.0 [52].

Cypher not only allows to exploit the graph structure of the Bitcoin transaction data, but gives also the opportunity to express the heuristics easily. Listing 4.9 shows the Multi-Input heuristic as Cypher query that can be run directly on a Neo4j graph via the Neo4j Browser or the Neo4j Cypher Shell. An example of its application on a Transaction Graph can be seen in Figure 4.7, where the heuristic is applied to the Transaction Graph in (a) and (b) shows the result of that application. In fact two new **IS_SAME** relationships are created in the same Transaction Graph. However, the other nodes and relationships are omitted in the Figure to emphasize the newly created relationships.

Listing 4.9: Multi-Input heuristic in Cypher

```
1 MATCH (a1:Address)-[:SENDS]->(t:Transaction)<-[:SENDS]-(a2:Address)
2 MERGE (a1)-[:IS_SAME]->(a2)
```



(a) Transaction Graph (b) Address Graph

Figure 4.7: Cypher Multi-Input heuristic application

There are currently two ways to use CAPS, *(i)* using a Zeppelin Notebook [53], and *(ii)* in a Scala Maven project. Both approaches use the programming language Scala. Therefore, the Scala implementation of the Multi-Input heuristic presented in Listing 4.10 can be used in a Zeppelin Notebook as well as in a Scala Maven project. This code, however, could not be tested because of an error in CAPS's Maven dependency.

Listing 4.10: Multi-Input heuristic in Scala

```
1  // Create CAPS caps
2  implicit val caps: CAPSSession = CAPSSession.create(spark)
3
4  val neo4jConfig = Neo4jConfig(URI.create("bolt://localhost:7687"), user =
       "neo4j", password = Some("password"), encrypted = false)
5
6
7  caps.registerSource(Namespace("txGraph"),
       GraphSources.cypher.neo4j(neo4jConfig))
8
9  def multiInputQuery(fromGraph: String): String =
10     s"""FROM GRAPH $fromGraph
11         |MATCH (a1:Address)-[:SENDS]->(t)<-[:SENDS]-(a2:Address)
12         |CONSTRUCT
13         |  CREATE (a1)-[:IS_SAME]->(a2)
14         |RETURN GRAPH
15       """.stripMargin
16
17 // Create the Address Graph by applying the Multi-Input heuristic
18 val addressGraph =
       caps.cypher(multiInputQuery(s"txGraph.$entireGraphName")).graph
19
20 // Merge graph into existing Neo4j database
21 Neo4jGraphMerge.merge(entireGraphName, addressGraph, neo4jConfig)
```

In the following paragraph the Multi-Input Scala program presented in Listing 4.10 is explained: First, a `CAPSSession` is created. A `CAPSSession` wraps a SparkSession in order to use the underlying Spark APIs. In the fourth line a `Neo4jConfig` is created which is needed to access the graph currently running in the Neo4j database. Afterwards, the graph stored in Neo4j is registered as a source with the name `txGraph` in the CAPSSession. The function `multiInputQuery` holds the Cypher implementation of the Multi-Input heuristic and returns it as a string. This implementation creates a new graph consisting of Address nodes and IS_SAME relationships. In contrast to Cypher used directly in Neo4j, it is not possible in CAPS to create new nodes and relationships directly in the existing Neo4j database, and hence a new graph must be created which exists only in CAPS. This new graph can then for example be merged into the existing Neo4j graph using `Neo4jGraphMerge`, as can be seen in line 21.

Chapter 5

Evaluation and Challenges Discussion

This chapter presents a performance evaluation of the solution proposed in Chapter 4 as well as a discussion of the challenges faced in the development phase. The performance evaluation is structured in the way of the solution workflow. Firstly, the Bitcoin blockchain is parsed and the Transaction Graph is created. Secondly, duplicate Address nodes are removed. Thirdly, the Transaction Graph is imported into Neo4j. Finally, the addresses are clustered using a heuristic.

5.1 Evaluation

In this section, the solution is evaluated in terms of processing time and storage use. However, due to errors encountered in the employed tools, such as the CAPS project used for the distributed data analysis mentioned in Section 4.2.3, not all the parts of the solution could be evaluated. Thus, the evaluation only comprises the fully functional parts. The hardware and height of the blockchain used are always described in the specific section.

5.1.1 Blockchain Parsing

For this evaluation the Bitcoin blockchain was parsed until block height 564,700 using `btc-csv` parser. At this height, the size of the blockchain was around 205 GB. Table 5.1 presents the hardware used for the parsing.

Component	Description
Processor	Intel Core i5-2500 CPU @ 3.30GHz x 4
Memory	16 GB Memory
Disk	1 TB SSD

Table 5.1: System Used to Parse the Bitcoin Blockchain

It took 29 hours, 59 minutes, and 31 seconds to run `btc-csv` from block height 0 to 564,700. The outcome was csv files holding nodes and relationships with a total size of 331.4 GB. The sizes of the individual files can be found in Table 5.2. The total amount of disk space needed for the nodes and relationships exceeds the size of the blockchain by more than 120 GB. The files for the relationships with a combined size of 270.0734 GB make up a large proportion (81%) of the total size. The combined size of the node files is only 61.3405 GB. This difference is not only because there might be more relationships than nodes, *i.e.,* from address reuse, but also because each relationship contains two long node ID's, *i.e.,* a SENDS relationship contains a Bitcoin address and a txid.

File	Size
`addresses.csv`	36.2 GB
`blocks.csv`	40.5 MB
`transactions.csv`	25.1 GB
`before_rel.csv`	73.4 MB
`belongs_to_rel.csv`	50.3 GB
`receives_rel.csv`	114.4 GB
`sends_rel.csv`	105.3 GB
Total Size	331.4139 GB

Table 5.2: Sizes of the Node and Relationship csv Files

As described in Section 4.2.1, `btc-csv` uses a leveldb UTXO store, called `utxo`, to resolve Bitcoin addresses for transaction inputs. Since, an entry is deleted as soon as it is spent, its size varies. After parsing was finished at block height 564,700 the size of the complete leveldb was 6.6 GB.

Figure 5.1 shows a the parsing time for different block heights. To create this chart the parsing time was logged after each 100 blocks. Petkanic's blockchain parser needed 23 hours 49 minutes for 508,000 blocks with Solid State Drive (SSD) (without an SSD it would take more than 150 hours) [39]. In contrast `btc-csv` 21 hours 58 minutes for the same amount of blocks. Nevertheless, it is very difficult to compare different parsers, since they generate different files as output. Petkanic's solution converts raw blocks into JSON files holding the relevant information of the block. In contrast, `btc-csv` creates nodes and relationships. Both solutions, however, resolve input addresses while parsing.

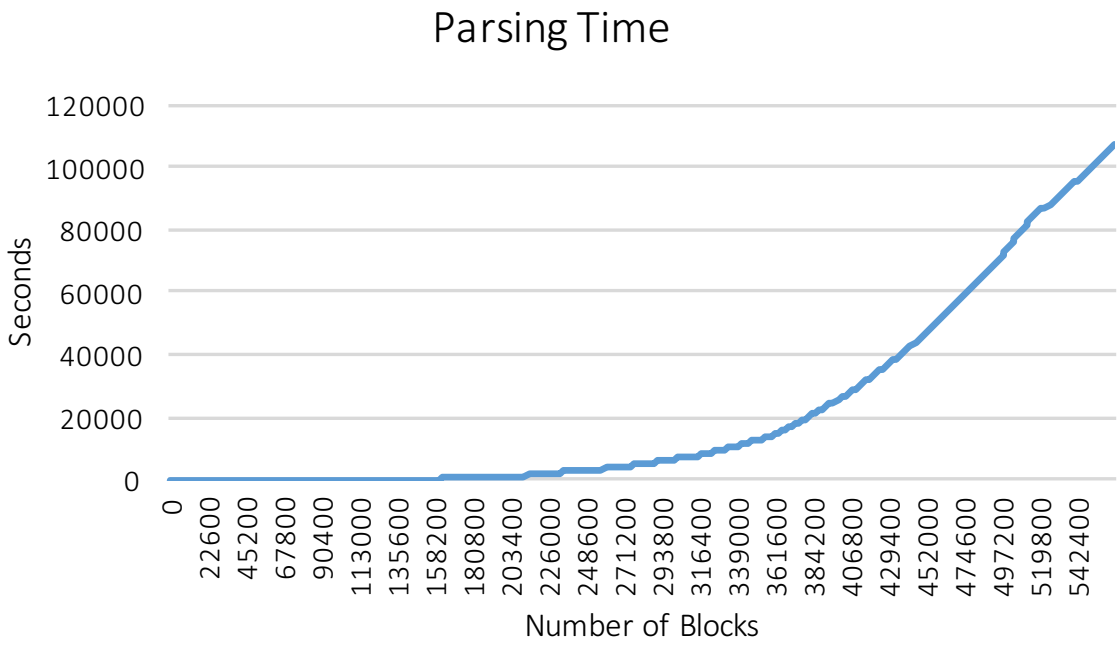


Figure 5.1: Time to Parse a Number of Blocks

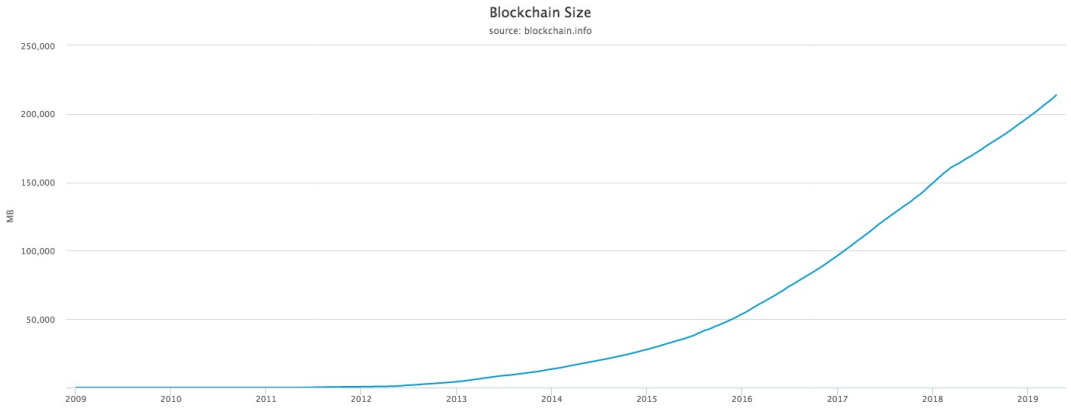


Figure 5.2: Size of the Bitcoin blockchain

Figure 5.2 shows the increase in size of the Bitcoin blockchain from 2009 until 2019. By comparing the parsing times in Figure 5.1 with the blockchain sizes in Figure 5.2 it can be seen that the parsing times behave equally to the blockchain sizes. An experiment where the spent UTXOs were not deleted from `utxo` showed that the parsing times tend to grow faster than the blockchain. Parsing 546,700 blocks took 40 hours 20 minutes which can be related to longer look-up times.

5.1.2 Deduplication

The removal of duplicated address nodes using Linux's sort algorithm took 31 minutes and 51 seconds on the hardware presented in Table 5.3. The size of the file `address.csv` reduced from initially 36.2 GB to 17.0 GB.

Component	Description
Processor	Intel Core i5-2500 CPU @ 3.30GHz x 4
Memory	16 GB Memory
Disk	1 TB SSD
Linux OS	Ubuntu 18.04 LTS

Table 5.3: System Used for Address Node Deduplication

File	Size
`addresses.csv`	17.0 GB
`blocks.csv`	40.5 MB
`transactions.csv`	25.1 GB
`before_rel.csv`	73.4 MB
`belongs_to_rel.csv`	50.3 GB
`receives_rel.csv`	114.4 GB
`sends_rel.csv`	105.3 GB
Total Size	312.2139 GB

Table 5.4: Sizes of the Deduplicated Node and Relationship csv Files

5.1.3 Neo4j Import

To import the nodes and relationships from the deduplicated csv files presented in Table 5.4 the Neo4j Import Tool was used on the system shown in Table 5.5. The complete import took 7 hours 41 minutes 34 seconds 899 ms with a peak memory usage of 10.74 GB. The resulting Neo4j graph has a size of 246.27 GB, and contains the Neo4j nodes presented in Table 5.6 and relationships described in Table 5.7.

Component	Description
Processor	Intel Core i5-2500 CPU @ 3.30GHz x 4
Memory	16 GB Memory
Disk	1 TB SSD
Linux OS	Ubuntu 18.04 LTS
Neo4j	Neo4j Desktop 1.1.15 (Neo4j 3.5.3)

Table 5.5: System Used for Neo4j Import

Node Type	Count
Address	48,4364,667
Block	564,701
Transaction	38,6749,163
All Nodes	87,1678,531

Table 5.6: Nodes in Neo4j

Relationship Type	Count
IS_BEFORE	564,700
BELONGS_TO	386,749,165
RECEIVES	1,031,175,599
SENDS	974,350,717
All Relationships	2,392,840,181

Table 5.7: Relationships in Neo4j

5.1.4 Address Clustering

Due to the problems encountered when applying the CAPS approach, alternative applications of the Multi-Input heuristic are evaluated in this subsection. These approaches (Approach #1, Approach #2, and Approach #3) were applied on the Neo4j database directly.

Component	Description
Processor	Intel Core i7-3520 CPU @ 2.90GHz x 4
Memory	16 GB Memory
Disk	500 GB SSD
Linux OS	Ubuntu 18.04 LTS
Neo4j	Neo4j Desktop 1.1.15 (Neo4j 3.5.3)

Table 5.8: System Used to Cluster Addresses

For the following performance measurements, a Neo4j Transaction Graph with block height 100,000 was used if nothing else is mentioned. The size of the Bitcoin blockchain was around 10 MB at this block height [2]. The resulting Neo4j graph has a size of 108.33 MB, and contains the Neo4j nodes presented in Table 5.9 and relationships described in Table 5.10.

Node Type	Count
Address	174,701
Block	100,001
Transaction	216,575
All Nodes	491,277

Table 5.9: Nodes in Neo4j at Block height 100,000

Relationship Type	Count
IS_BEFORE	100,000
BELONGS_TO	216,577
RECEIVES	264,251
SENDS	192,363
All Relationships	5,162,023

Table 5.10: Relationships in Neo4j at Block height 100,000

Approach #1

This approach applies the Multi-Input heuristic on the Neo4j graph and writes an **IS_SAME** relationship to the file `is_same_rel.csv` for every match. Listing 5.1 shows the Cypher

query employed in this approach.

Listing 5.1: Batch Multi-Input heuristic in Cypher

```
CALL apoc.export.csv.query(
'MATCH (a1:Address)-[:SENDS]->(t:Transaction)<-[:SENDS]-(a2:Address)
RETURN a1.address, a2.address','/home/csg/BA/heuristics/is_same_rel.csv',
    {useTypes: false, quotes: false});
```

The execution time of the Cypher query was 21 seconds 474 ms on the system presented in 5.8, and the deduplication using `sort` took less than a second. The size of the file `is_same_rel.csv` was 721.8 MB before removing duplicates and 324.4 MB after. This means that 55% of all the relationships in the file were duplicates.

The execution of this query on the complete Transaction Graph (block height 564,700) had to be manually stopped after 12 hours because the size of the csv file exceeded 600 GB. This size makes deduplication using `sort` impossible. This experiment was carried out on the system presented in Table 5.5.

Approach #2

This approach applies the Multi-Input heuristic on the Neo4j graph and creates an **IS_SAME** relationship for every match if there does not exist one. The result is that each two addresses belonging together are connected through two **IS_SAME** relationships, as presented in Figure 5.3. Listing 5.2 shows the Cypher query used in this approach.

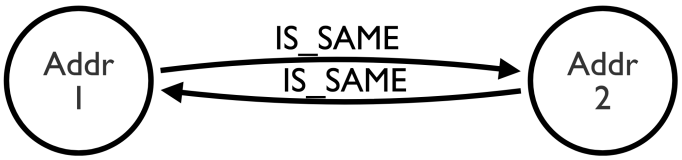


Figure 5.3: Two Addresses that Belong Together According to #Approach 2

Listing 5.2: Batch Multi-Input heuristic in Cypher

```
CALL apoc.periodic.iterate(
"MATCH (a1:Address)-[:SENDS]->(t:Transaction)<-[:SENDS]-(a2:Address) RETURN
    a1, a2",
"MERGE (a1)-[:IS:SAME]->(a2)",
{batchSize:10000,iterateList:true,parallel:false})
```

The execution time of the Cypher query was 9 minutes 17 seconds 316 ms on the system shown in 5.8 and 4,388,808 **IS_SAME** relationships were created. The size of the Neo4j graph increased from 108.33 MB to 1.51 GB, *i.e.,* an increase of 1.4 GB.

Approach #3

This approach applies the Multi-Input heuristic on the Neo4j graph and creates an **IS_SAME** relationship for every match if there does not exist one no matter in which direction. The result is that each two addresses belonging together are connected through one **IS_SAME** relationship, as presented in Figure 5.4. Listing 5.3 shows the Cypher query used in this approach.

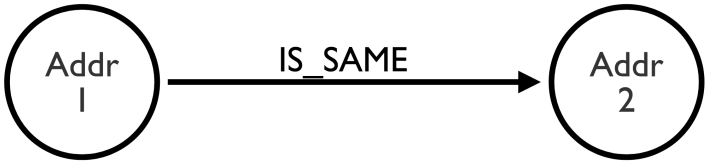


Figure 5.4: Two Addresses that Belong Together According to Approach #3

Listing 5.3: Batch Multi-Input heuristic in Cypher

```
1 CALL apoc.periodic.iterate(
2 "MATCH (a1:Address)-[:SENDS]->(t:Transaction)<-[:SENDS]-(a2:Address) RETURN
     a1, a2",
3 "MERGE (a1)-[:IS_SAME]-(a2)",
4 {batchSize:10000,iterateList:true,parallel:false})
```

The execution time of the Cypher query was 8 minutes 14 seconds 131 ms on the system presented in 5.8 and 2,194,978 **IS_SAME** relationships were created. The size of the Neo4j graph increased from 108.33 MB to 753.17 MB, *i.e.,* an increase of 644.84 MB.

Comparison of Approaches

Although these three approaches are not practical for the complete blockchain, because of their extensive execution times due to the blockchain size, they provide interesting insights. As can be seen in *Approach #1*, 55% of all the results were duplicates. Since all the approaches use the same `MATCH` clause, they all find the same duplicates. The difference is that *Approach #1* does not eliminate any duplicate relationships, whereas *Approach #2* and *Approach #3* check whether the current relationship already exists or not. Moreover, any possible implementation of the Multi-Input heuristic will have to eliminate duplicate relationships in order to provide a valuable output, either during the application of the heuristic or later on.

Moreover, when *Approach #1* was applied to the complete Transaction graph, the csv file holding the results (including duplicates) exceeded 600 GB, meaning more than 8.7 billion relationships. It becomes difficult to deduplicate a file of this size using `sort`. This problem can be reduced by using address IDs instead of the 34 byte Bitcoin addresses. Using address IDs would also improve performance in other parts of the solution, and save disk space.

To compare the complete run times of the three approaches, the deduplicated relationships from *Approach 1* were imported together with the address nodes using the Neo4j Import Tool. Including the import, which took 16 seconds and 754 ms, to complete *Approach 1* took 39 seconds and 228ms. This is much faster than the 9 minutes 17 seconds 316 ms of *Approach #2* (around 7% of *Approach #2*) and the 8 minutes 14 seconds 131 ms of *Approach #3* (around 8% of *Approach #3*).

The insights gathered in this comparison lead to the following conclusions: Firstly, it is important to use address IDs instead of addresses to reduce the size of a relationship. Secondly, it is important for the performance of the address clustering solution to have a fast mechanism to remove duplicated relationships. As the comparison has shown, Neo4j's `MERGE` seems to be relatively slow in comparison to the `sort` method. If the deduplication in the Spark cluster (CAPS) using either the Cypher `MERGE` or a distributed sorting algorithm cannot exploit the computing cluster, it might be worth considering to change to a highly optimised local solution for the complete workflow. However, such a solution would not be as adaptable as the solution presented in this thesis.

5.2 Challenges Discussion

During the development of the presented solution various challenges were encountered. Some of them lead to a changes in the approach, while others could not be implemented.

The first challenge was the use of `bclib`. Since it is a relatively new blockchain parsing library, it lacks official documentation. Although the authors of this work were not familiar with Go and the library was lacking documentation, it was decided to use this library because the developer, after a brief contact about the library, provided all the support and documentation available.

After implementing the Bitcoin blockchain parser `btc-csv` using `bclib`, the parser was tested with a relatively small portion of the blockchain (*e.g.,* first 300,000 blocks) and everything ran without errors. When 564,700 blocks were given as input for the parser, an UTXO used in a transaction from block 324,121 could not be resolved. At that time, a Go map was used as `utxo` in the implementation. Although the estimated size of `utxo` should not exceed the machines memory, the in-memory solution was replaced by a leveldb solution on disk. Further, a next test showed that it was not the `utxo` that caused the problem, but rather a problem in the `bclib`. After further investigation of the library developer, Kevin Primicerio, he could detect the problem and fix it.

The initial approach for the workflow was envisioned to write the nodes and relationships of the Transaction Graph directly to Neo4j using the Neo4j Go Driver. However, since the Neo4j Go Driver was first released in November 2018 [54], there was not much information regarding its effectiveness and correct functioning. The first implementation of `btc-csv` using Neo4j Go Driver was extremely slow. In the first version, UTXO look-ups were also done in Neo4j. However, even the change of performing UTXO look-ups in a different manner, *i.e.,* with the UTXO store `utxo`, did not have noticeably effects on the performance of the program. Therefore, the approach had to be changed to write the

nodes and relationships to csv files and then, after the parsing, import them using the Neo4j Import Tool.

The next problem faced was the data import using the Neo4j Import Tool. This import method is supposed to be the fastest of all import methods because it is able to use parallelisation. As a first experiment, it was tried to import the csv files with a total size of around 45 GB. After 16 hours the import was still at 60% and was just using one CPU core. Thus, as it was not clear whether that was normal behaviour or what caused the problem (*e.g.,* too long IDs or incorrect Neo4j configuration), the Neo4j Community [55] was contacted. After the contact, the developers of Neo4j suspected that the Import Tool's deduplication was the root of the problem. Therefore, a deduplication step was included before the import, which resulted in a import time of less than 30 minutes.

After the import of the data to Neo4j, it was envisioned to employ Cypher for Apache Spark in a Zeppelin Notebook to implement the heuristics. By relying on this approach, a user of the solution does not have to modify a Scala project with Maven dependencies if a new heuristic should be executed. Even after deep research on the openCypher's wiki regarding the use of CAPS in a Zeppelin Notebook [56], and conversations held by Neo4j community, such as one from Martin Junghanns [57], this approach presented to be complex for the thesis time-frame. Therefore, it was chosen to use CAPS in a Scala project. The CAPS project provides examples which use different data sources, *e.g.,* Neo4j or graphs stored in a special structure of csv and JSON files. To utilise CAPS in a Scala project it must be added as Maven dependency. After several attempts to use CAPS as a Maven dependency, it was decided to write an issue on the CAPS *GitHub* repository. To allow the developers to reproduce the error, it was created a small program based on an example published by the CAPS team. They were able to reproduce the error and suspect that it was due to a packaging problem. However, the issue, at the time of this thesis writing, is still marked as bug and not closed. Thus, the further development of the present thesis was halted and the remaining steps marked as future work.

Chapter 6

Conclusions and Future Work

In this thesis, an approach to process the Bitcoin blockchain data and output helpful insights, *e.g.,* address clusters, was presented. In order to be highly adaptable, this approach is based on Neo4j and Apache Spark. This allows to reason about address clustering heuristics based on the Transaction Graph and implement them in Neo4j exploiting the graph nature of the data, while shorten the execution time by distributing the work via Apache Spark.

The propsed approach was evaluated in terms of the time needed to import and extract the data to a Neo4j graph. The evaluation showed that it took 9 hours, 59 minutes, and 31 seconds to parse the Bitcoin blockchain with a size of 205.5 GB and create a Transaction Graph from it. The removal of duplicate address nodes took 31 minutes and 51 seconds, and the resulting deduplicated Transaction Graph was imported into Neo4j within 7 hours, 41 minutes, 34 seconds, and 899 milliseconds. However, the processing of the Bitcoin blockchain data using Cypher for Apache Spark presented some challenges, such as lack of documentation, and a software bug in CAPS Maven dependency.

Nonetheless, through applying three different approaches of the Multi-Input heuristic, valuable insights were found. The Multi-Input heuristic detected more than 8.7 billion relationships, that is more than 600 GB in csv format, when applied to the complete blockchain. By applying it to a smaller Transaction Graph it could be shown that 55% of the detected relationships were duplicates. These insights lead to the following conclusions: Firstly, it is important to use address IDs instead of addresses to reduce the size of a relationship. Secondly, it is important for the performance of the address clustering to have a fast mechanism to remove duplicated relationships. If the deduplication in a distributed approach cannot exploit the computing cluster, it might be worth considering to change to a highly optimised local solution.

Based on the related work description in Chapter 3 and the challenges faced during the development of this thesis described in Section 5.2, it can be seen that processing to gather useful insights about the Bitcoin blockchain is a promising research topic that the academia is focused on. However, the tools and solutions employed in this thesis are still under development, but present an active community which works to fix software bugs and improve them. In conclusion, the approach of combining Neo4j with Apache Spark is

promising. Unfortunately, due to the time-frame of this bachelor thesis and the described challenges, some parts of the approach were not implemented. Nevertheless, it is expected that the present thesis shed light on future works on the topic.

Therefore, future work proposals include, but are not limited to, *(i)* introduce address IDs instead of addresses in transactions *(ii)* finish the address clustering approach using CAPS as soon as the bug is fixed, *(iii)* apply different heuristics, *(iv)* implement the Multi-Input heuristic in Go, and *(v)* compare the performance of the distributed solution with the local implementation.

Bibliography

[1] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 11/23/2018).

[2] Blockchain Luxembourg S.A. *Blockchain Size*. 2019. URL: https://blockchain.info/charts/blocks-size (visited on 03/21/2019).

[3] The Apache Software Foundation. *Apache Spark - Unified Analytics Engine for Big Data*. URL: https://spark.apache.org/ (visited on 04/04/2019).

[4] Jeremy Rubin. *BTCSpark: Scalable Analysis of the Bitcoin Blockchain using Spark*. 2015. URL: https://rubin.io/public/pdfs/s897report.pdf (visited on 03/21/2019).

[5] Harry Kalodner et al. "BlockSci: Design and applications of a blockchain analysis platform". *arXiv:1709.02489 [cs]* (Sept. 7, 2017). arXiv: 1709.02489. URL: http://arxiv.org/abs/1709.02489 (visited on 03/26/2019).

[6] D. D. F. Maesa, A. Marino, and L. Ricci. "Uncovering the Bitcoin Blockchain: An Analysis of the Full Users Graph". *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. Oct. 2016, pp. 537–546. DOI: 10.1109/DSAA.2016.52.

[7] Sarah Meiklejohn et al. "A Fistful of Bitcoins: Characterizing Payments Among Men with No Names". *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. New York, NY, USA: ACM, 2013, pp. 127–140. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504747. URL: http://doi.acm.org/10.1145/2504730.2504747 (visited on 03/23/2019).

[8] Andreas M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. 1st. O'Reilly Media, Inc., 2014. ISBN: 9781449374044.

[9] Bitcoin. *Address Documentation*. URL: https://en.bitcoin.it/wiki/Address (visited on 03/18/2019).

[10] Bitcoin. *Pay-to-Pubkey Hash*. URL: https://en.bitcoinwiki.org/wiki/Pay-to-Pubkey_Hash (visited on 04/23/2019).

[11] Bitcoin. *Pay to Script Hash*. URL: https://en.bitcoin.it/wiki/Pay_to_script_hash#Addresses (visited on 04/23/2019).

[12] Bitcoin. *Bech32*. URL: https://en.bitcoin.it/wiki/Bech32 (visited on 04/23/2019).

[13] Bitcoin. *OP_RETURN Documentation*. URL: https://en.bitcoin.it/wiki/OP_RETURN (visited on 03/19/2019).

[14] Bitcoin. *Mining Documentation*. URL: https://en.bitcoin.it/wiki/Mining (visited on 03/21/2019).

[15] Bitcoin. *Proof-of-Work Documentation*. URL: https://en.bitcoin.it/wiki/Proof_of_work (visited on 03/21/2019).

[16] BitcoinCore. *About*. URL: https://bitcoincore.org/en/about/ (visited on 03/21/2019).

[17] Bitcoin. *Bitcoin Core 0.11: Data Storage*. URL: https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage (visited on 03/21/2019).

[18] Bitcoin. *Protocol Documentation*. URL: https://en.bitcoin.it/wiki/Protocol_documentation#Block_Headers (visited on 03/08/2019).

[19] Neo4j. *Graph Platform*. URL: https://neo4j.com/product/ (visited on 04/03/2019).

[20] Neo4j. *Compare Neo4j Editions*. URL: https://neo4j.com/subscriptions/#editions (visited on 04/03/2019).

[21] Holden Karau et al. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

[22] F. Reid and M. Harrigan. "An Analysis of Anonymity in the Bitcoin System". *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*. Oct. 2011, pp. 1318–1326. DOI: 10.1109/PASSAT/SocialCom.2011.79.

[23] Elli Androulaki et al. "Evaluating User Privacy in Bitcoin". *Financial Cryptography and Data Security*. Ed. by Ahmad-Reza Sadeghi. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 34–51. ISBN: 978-3-642-39884-1.

[24] Michael Fleder, Michael S. Kester, and Sudeep Pillai. "Bitcoin Transaction Graph Analysis". *arXiv:1502.01657 [cs]* (Feb. 5, 2015). arXiv: 1502.01657. URL: http://arxiv.org/abs/1502.01657 (visited on 03/24/2019).

[25] Matthias Lischke and Benjamin Fabian. "Analyzing the Bitcoin Network: The First Four Years". *Future Internet* 8.1 (2016). ISSN: 1999-5903. DOI: 10.3390/fi8010007. URL: http://www.mdpi.com/1999-5903/8/1/7.

[26] D. D. F. Maesa, A. Marino, and L. Ricci. "Uncovering the Bitcoin Blockchain: An Analysis of the Full Users Graph". *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. Oct. 2016, pp. 537–546. DOI: 10.1109/DSAA.2016.52.

[27] M. Harrigan and C. Fretter. "The Unreasonable Effectiveness of Address Clustering". *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. July 2016, pp. 368–373. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0071.

[28] D. Ermilov, M. Panov, and Y. Yanovich. "Automatic Bitcoin Address Clustering". *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. Dec. 2017, pp. 461–466. DOI: 10.1109/ICMLA.2017.0-118.

[29] T. Chang and D. Svetinovic. "Improving Bitcoin Ownership Identification Using Transaction Patterns Analysis". *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2018), pp. 1–12. ISSN: 2168-2216. DOI: 10.1109/TSMC.2018.2867497.

[30] Danno Ferrin. "A preliminary field guide for bitcoin transaction patterns". *Texas Bitcoin Conference*. 2015, pp. 1–8.

[31] G. Andresen. *bitcointools*. URL: https://github.com/tuxsoul/bitcoin-tools (visited on 03/27/2019).

[32] Michele Spagnuolo, Federico Maggi, and Stefano Zanero. "BitIodine: Extracting Intelligence from the Bitcoin Network". *Financial Cryptography and Data Security*.

Ed. by Nicolas Christin and Reihaneh Safavi-Naini. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 457–468. ISBN: 978-3-662-45472-5.

[33] znort987. *blockparser*. URL: https://github.com/znort987/blockparser (visited on 03/27/2019).

[34] Armory Technologies Inc. *Armory*. URL: https://github.com/etotheipi/BitcoinArmory (visited on 03/27/2019).

[35] Core Armory Developer etotheipi. *Armory - Discussion Thread*. URL: https://bitcointalk.org/index.php?topic=56424.0 (visited on 03/27/2019).

[36] Massimo Bartoletti et al. "A General Framework for Blockchain Analytics". *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL '17. New York, NY, USA: ACM, 2017, 7:1–7:6. ISBN: 978-1-4503-5173-7. DOI: 10.1145/3152824.3152831. URL: http://doi.acm.org/10.1145/3152824.3152831 (visited on 03/26/2019).

[37] Bartoletti et al. *Blockchain Analytics*. URL: http://blockchain.unica.it/projects/blockchain-analytics/ (visited on 03/27/2019).

[38] Greg Walker. *bitcoin-to-neo4j*. URL: https://github.com/in3rsha/bitcoin-to-neo4j (visited on 03/28/2019).

[39] Peter Petkanič. *Bitcoin Blockchain Analysis*. 2018. URL: https://is.muni.cz/th/v2dsl/bp_petkanic.pdf (visited on 03/31/2019).

[40] Malte Möser and Rainer Böhme. "Trends, Tips, Tolls: A Longitudinal Study of Bitcoin Transaction Fees". *Financial Cryptography and Data Security*. Ed. by Michael Brenner et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 19–33. ISBN: 978-3-662-48051-9.

[41] Malte Möser and Rainer Böhme. "The price of anonymity: empirical evidence from a market for Bitcoin anonymization". *Journal of Cybersecurity* 3.2 (June 1, 2017), pp. 127–135. ISSN: 2057-2085. DOI: 10.1093/cybsec/tyx007. URL: https://academic.oup.com/cybersecurity/article/3/2/127/4057584 (visited on 03/26/2019).

[42] Bitcoin. *CoinJoin*. URL: https://en.bitcoin.it/wiki/CoinJoin (visited on 04/23/2019).

[43] solid IT gmbh. *DB-Engines Ranking of Graph DBMS*. URL: https://db-engines.com/en/ranking/graph+dbms (visited on 04/05/2019).

[44] Neo4j. *Graph Algorithms in Neo4j: Strongly Connected Components*. URL: https://neo4j.com/blog/graph-algorithms-neo4j-strongly-connected-components/ (visited on 04/06/2019).

[45] Kevin Primicerio. *A Golang Bitcoin library*. URL: https://github.com/chainswatch/bclib (visited on 04/15/2019).

[46] Go Developers. *The LevelDB key-value database in the Go programming language*. URL: https://github.com/golang/leveldb (visited on 04/15/2019).

[47] Neo4j. *Neo4j Go Driver*. URL: https://github.com/neo4j/neo4j-go-driver (visited on 04/01/2019).

[48] Neo4j. *Using Neo4j from Go*. URL: https://neo4j.com/developer/go/ (visited on 04/01/2019).

[49] Neo4j. *Import Tool*. URL: https://neo4j.com/docs/operations-manual/current/tools/import/ (visited on 04/01/2019).

[50] Neo4j. *LOAD CSV*. URL: https://neo4j.com/docs/cypher-manual/3.5/clauses/load-csv/#load-csv-importing-large-amounts-of-data (visited on 04/01/2019).

[51] openCypher. *Cypher for Apache Spark*. URL: https://github.com/opencypher/cypher-for-apache-spark (visited on 04/20/2019).

[52] openCypher. *What is openCypher?* URL: https://www.opencypher.org/ (visited on 04/20/2019).

[53] The Apache Software Foundation. *Apache Zeppelin*. URL: https://zeppelin.apache.org/ (visited on 04/20/2019).

[54] Neo4j. *Neo4j Go Driver Releases*. URL: https://github.com/neo4j/neo4j-go-driver/releases (visited on 04/20/2019).

[55] Neo4j. *Neo4j-admin import uses only one cpu core after a while*. URL: https://community.neo4j.com/t/neo4j-admin-import-uses-only-one-cpu-core-after-a-while/5393 (visited on 04/23/2019).

[56] openCypher. *Use CAPS in a Zeppelin notebook*. URL: https://github.com/opencypher/cypher-for-apache-spark/wiki/Use-CAPS-in-a-Zeppelin-notebook (visited on 04/23/2019).

[57] Martin Junghanns Neo4j. *Matching Patterns and Constructing Graphs with Cypher for Apache Spark with Martin Junghanns Neo4j*. URL: https://www.youtube.com/watch?v=XYDzx14cznc (visited on 04/23/2019).

Abbreviations

BIP	Bitcoin Improvement Protocol
CAPS	Cypher for Apache Spark
CSV	Comma Separated Value
DBMS	Database Management System
GB	Gigabyte
JSON	JavaScript Object Notation
MB	Megabyte
P2PKH	Pay-to-PubkeyHash
P2SH	Pay-to-Script-Hash
PoW	Proof-of-Work
RAM	Random Access Memory
SSD	Solid State Drive
UTXO	Unspent Transaction Output

Glossary

Address Clustering Grouping of Bitcoin addresses that belong to the same user.

Blockchain A Decentralized Distributed ledger that enables the transaction of cryptocurrencies between unstrusted peers

Blockchain Fork A situation on the blockchain where a parallel chain is maintained besides the original one. This situation happens when miners are working on different chains. In Bitcoin it is resolved by selecting the longest chain.

Big Data Massive amount of data stored not yet. It can be viewed as the act of processing this data to produce useful insights about its meaning and relations.

Bitcoin The first cryptocurrency based on the blockchain technology. The creator is still unknown to date.

CoinJoin A trustless method for combining Bitcoin payments from multiple spenders in a single transaction used to improve privacy.

Cypher A graph database query language maintained by the openCypher project.

Go A statically typed, compiled programming language.

LevelDB A fast persistent key-value store.

Mainnet Main Bitcoin blockchain.

Parsing The act of transforming raw data into a defined syntax following rules to be later managed or interpreted.

Satoshi The currently smallest unit of Bitcoin currency possible (0.00000001 BTC).

Scala An object-oriented and functional programming language.

SWIG A software development tool that connects C and C++ programs with a variety of high-level programming languages, such as Python.

Testnet An alternative Bitcoin blockchain used for testing. There have been three version, the current one being Testnet3.

List of Figures

List of Tables

Appendix A

Installation Guidelines

A.1 btc-csv

This program allows to parse the bitcoin blockchain and write the nodes and relationships of the created Tranaction Graph to csv.

A.1.1 Getting Started

The following section describes how to setup the environment to be able to run btc-csv.

Prerequisits

1. Install Golang and set up your environment.
2. Get this program:

   ```
   $ go get -u github.com/wallerprogramm/btc-csv
   ```

3. Create a file with name .env in
   `$GOPATH/src/github.com/wallerprogramm/btc-csv`
   with the following content:

   ```
   # The package
   PKG=btc-csv
   # The folder where the BTC blockchain is stored
   DATADIR=/path/to/bitcoinfolder
   #often: DATADIR=/home/username/.bitcoin
   # The folder where the utxo leveldb is stored
   DBDIR=/path/to/leveldb/utxo
   ```

A.1.2 Run btc-csv

Prerequisits

Choose from which height to which height should be parsed. To do so enter main.go and change height to the wished end block height. If the starting block height should be changed to n, be aware that the `utxo` leveldb with block height (n-1) has to be provided. To change the start height change the 0 in

```
btc.LoadFile(0, height, grapho.Build, c)
```

to the wished start height.

Install all dependecies of btc-csv:

```
$ cd $GOPATH/github.com/wallerprogramm/btc-csv
$ go get ./...
```

Install btc-csv:

```
$ cd $GOPATH/github.com/wallerprogramm/btc-csv
$ go install
```

Run it

1. Make sure that all the csv files that should not be changed are moved away from `$GOPATH/github.com/wallerprogramm/btc-csv`.

2. Make sure that the leveldb is either inexistent (if start height = 0) or the needed leveldb is in place (if start height > 0)

3. Run btc-csv:

   ```
   $ $GOBIN/btc-csv
   ```

4. Wait for a couple of hours.

A.1.3 Remove duplicate addresses from addresses.csv

```
$ cd $GOPATH/github.com/wallerprogramm/btc-csv
$ sort -u addresses.csv -o addresses.csv -T /path/to/folder/for/tmp/files
```

A.1.4 Import the nodes and relationships into Neo4j

Prerequisits

1. Install Neo4j Desktop (download it here `https://neo4j.com/download-center/`) if not already installed.

Run it

1. Start Neo4j Desktop
2. Create a new graph (choose a name, pw: password).
3. Change the memory settings of the graph under Settings to the following:

```
dbms.memory.heap.initial_size=6G
dbms.memory.heap.max_size=6G
dbms.memory.pagecache.size=6G
```

4. Run the following command in the Neo4j Desktop Terminal:

```
$ export DATA=/path/to/folder/containing/csv-files/
$ export HEADERS=/path/to/folder/containing/csv-headers/

$ ./bin/neo4j-admin import \
    --mode=csv \
    --database=btc.db \
    --nodes:Address $HEADERS/addresses-header.csv,$DATA/addresses.csv \
    --nodes:Block $HEADERS/blocks-header.csv,$DATA/blocks.csv \
    --nodes:Transaction \
        $HEADERS/transactions-header.csv,$DATA/transactions.csv \
    --relationships:IS\_BEFORE \
        $HEADERS/before_rel-header.csv,$DATA/before\_rel.csv \
    --relationships:BELONGS\_TO \
        $HEADERS/belongs_to_rel-header.csv,$DATA/belongs\_to\_rel.csv \
    --relationships:RECEIVES \
        $HEADERS/receives_rel-header.csv,$DATA/receives\_rel.csv \
    --relationships:SENDS \
        $HEADERS/sends_rel-header.csv,$DATA/sends\_rel.csv \
    --ignore-missing-nodes=true \
    --ignore-duplicate-nodes=true \
    --multiline-fields=true \
    --high-io=true
```

5. Change the used database in graph Settings by adding:

```
dbms.active\_database=btc.db
```

A.2 Distributed Data Analysis

A.2.1 Getting Started

The following section describes how to setup the environment to be able to use the program addressClustering that uses Cypher for Apache Spark to cluster Bitcoin addresses in a Spark cluster.

Prerequisits

1. Install Java 8
2. Scala 2.12
3. Spark 2.4

Run it

Further instructions are omitted because of the current bug in CAPS.

Appendix B

Contents of the CD

Thesis-BA-DS.pdf contains the thesis in a PDF.

Thesis-BA-DS.zip contains the thesis and figures as LaTeX source file.

Figures contains the figures as pdf and draw.io files.

btc-neo4j contains the Go sources of the parsing approach that writes to Neo4j directly.

btc-csv contains the Go sources of `btc-csv`.

csv_headers contains the csv header files needed for the Neo4j import.

zeppelin contains Zeppelin Notebooks.

addressClustering contains a Scala Maven project that contains different approaches to use CAPS, one of them being the example used to create the github issue.

Midterm_presentation.pptx contains the slides of the midterm presentation.

Midterm_presentation.pdf contains the slides of the midterm presentation as pdf.

measurements contains different files containing measurements.