BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PROCESSING OF THE BLOCKCHAIN EMPLOYING IPFS

VYUŽITÍ IPFS PRO ZPRACOVÁNÍ BLOCKCHAINU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MATÚŠ MÚČKA**
AUTOR PRÁCE

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

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2019/2020

Master's Thesis Specification

Student: **Múčka Matúš, Bc.**
Programme: Information Technology Field of study: Information Systems
Title: **Processing of the Blockchain Employing IPFS**
Category: Networking
Assignment:

1. Learn about cryptocurrencies and other blockchain technologies (namely, Ethereum, Bitcoin, DigiByte, Decred, Monero).
2. Study the Interplanetary File System, the principles of network communication, distribution and content addressing.
3. Design a prototype of the IPFS connector for generic blockchain access in compliance with the supervisor's recommendations.
4. Implement a prototype that delivers demanded functionality on a particular blockchain of considerable size. Write the API for integration with other applications.
5. Perform validation testing and measure prototype performance characteristics. Discuss possible extensions.

Recommended literature:

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

Requirements for the semestral defence:

- Tasks 1, 2 and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Veselý Vladimír, Ing., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: June 3, 2020
Approval date: October 22, 2019

Abstract

This work aims to design a system for processing blockchain data of selected cryptocurrencies for further exploring using IPFS. In this thesis, we created a proprietary decentralized and distributed database system that supports simple queries. The solution provides a well-arranged graphical user interface for data visualization as well as API, which makes it easy to connect to other systems. The benefit of this work is a new view of blockchain processing which opens up new possibilities in its exploring.

Abstrakt

Cieľom tejto práce je navrhnúť systém na spracovanie a preskúmavanie blockchainu vybraných kryptomien pri použití IPFS. Na riešenie tohoto problému bolo potrobné navrhnúť vlastný decentralizovaný a distribuovaný databázový systém, ktorý podporuje jednoduché dotazy. Vytvorené riešenie poskytuje prehľadné grafické užívateľské rozhranie, ktoré slúži na vizualizáciu dát a taktiež RestAPI, vďaka ktorému sa dá systém jednoducho napojiť na iné systémy. Prínosom tejto práce je nový pohľad na zpracovávanie blockchainu čo otvára nové možnosti v jeho prehľadávaní.

Keywords

IPFS, decentralization, blockchain, cryptocurrecny

Kľúčové slová

IPFS, decentralizácia, blockchain, cryptocurrecny

Reference

MÚČKA, Matúš. *Processing of the Blockchain Employing IPFS*. Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Veselý, Ph.D.

Rozšírený abstrakt

[[Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce. Cca 3 normostrany]]

Processing of the Blockchain Employing IPFS

Declaration

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

.....
Matúš Múčka
May 13, 2020

Acknowledgements

[[V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).]]

Contents

Chapter 1

Introduction

HTTP is "good enough" for the most use cases of distributing files over the network. However, when we want to stream lots of data to multiple connected clients at once, we are starting to hit its limits. When two clients are requesting the same data, there is no mechanism in HTTP that would allow sending the data only once. Sending duplicate data has become a problem in large companies because of bandwidth capacity. Blizzard[1] started to distribute video game content via a distributed solution because it was cheaper for the company and faster for players[11]. Linux's distributions use BitTorrent to transmit disk images[2].

The Bitcoin blockchain has now 242 gigabytes[3]. When blockchain is processed (all its data are parsed), the size can grow twice as much. If there are multiple blockchains, then data can have few terabytes. When we are sharing blockchains data from the server for several clients, there is a big chance that multiple clients want the same data. They may be working on the same case and investigating the same wallets. So in standard solution with relational database and some HTTP server, the server has to search in all data (that can have a size of few terabytes) and transmits selected data to the client for every request. This problem happens even if a different client asks for the same data in a few minutes ago. Behaviour mentioned above dramatically limits the scalability of the server.

Services that are using HTTP, often have client-server architecture, so there is also a problem with a single point of failure. If the server for some reason stops working, the client can not receive data. In a distributed file system such as IPFS, there is no such problem as one point of failure because all data are duplicated on multiple clients.

This Master thesis is divided into six chapters. Chapter 2 describes the differences between cryptocurrencies used in this project. Chapter 3 describes IPFS and all its layers. Design of the system created in this thesis is in Chapter 4. The implementation of all applications that are provided with this project is in chapter 5. Finally, summarization of results achieved in this work is in chapter 8.

[1] Game company – https://www.blizzard.com/

[2] Image of Debian downloadable by BitTorrent https://www.debian.org/CD/torrent-cd/

[3] A current size of bitcoin blockchain can be seen at https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/

Chapter 2

Cryptocurrencies

There were hundreds of failed attempts of creating cryptographic payment systems before cryptocurrencies like Bitcoin and Ethereum came into existence. Some of these systems are listed in Figure 2.1. All of them were created before Bitcoin. Despite that, only a few of them survived to these days. Some of these attempts were only academic proposals while others were deployed and tested systems. One of the survival is PayPal. It is only because it quickly gave up its original idea of hand-held devices for cryptographic payments. [22]

So there is a question, what makes cryptocurrencies successful nowadays? It may be easy to use principle and no need for external hardware. Another critical component of cryptocurrencies discussed in this work is blockchain. Generally, it is a ledger in which all transactions are securely stored. The idea behind blockchains is pretty old, and it was initially used for timestamping digital documents. [9]

ACC	CyberCents	iKP	MPTP	Proton
Agora	CyberCoin	IMB-MP	Net900	Redi-Charge
AIMP	CyberGold	InterCoin	NetBill	S/PAY
Allopass	DigiGold	Ipin	NetCard	Sandia Lab E-Cash
b-money	Digital Silk Road	Javien	NetCash	Secure Courier
BankNet	e-Comm	Karma	NetCheque	Semopo
Bitbit	E-Gold	LotteryTickets	NetFare	SET
Bitgold	Ecash	Lucre	No3rd	SET2Go
Bitpass	eCharge	MagicMoney	One Click Charge	SubScrip
C-SET	eCoin	Mandate	PayMe	Trivnet
CAFÉ	Edd	MicroMint	PayNet	TUB
CheckFree	eVend	Micromoney	PayPal	Twitpay
ClickandBuy	First Virtual	MilliCent	PaySafeCard	VeriFone
ClickShare	FSTC Electronic Check	Mini-Pay	PayTrust	VisaCash
CommerceNet	Geldkarte	Minitix	PayWord	Wallie
CommercePOINT	Globe Left	MobileMoney	Peppercoin	Way2Pay
CommerceSTAGE	Hashcash	Mojo	PhoneTicks	WorldPay
Cybank	HINDE	Mollie	Playspan	X-Pay
CyberCash	iBill	Mondex	Polling	

Figure 2.1: Electronic payment systems before cryptocurrencies[18]

2.1 Bitcoin

Bitcoin is probably the most famous cryptocurrency. It started as a digital currency transaction protocol but founded a new concept of blockchain[4]. On 3 January 2009, Satoshi Nakamoto (an alias for a person or group persons authored the bitcoin white paper) mined the genesis block of bitcoin (block with height 0). Satoshi got the reward of 50 bitcoins (half a million US dollars in the time of writing). This text was embedded in the block (see Figure 2.2): *The Times 03/Jan/2009 Chancellor on brink of second bailout for banks.*[1][6]. Bitcoin has a limited supply of coins to approximately 21 million. New coins are emitted only when a new block is created. [17]

```
00000000   01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000010   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000020   00 00 00 00 3B A3 ED FD  7A 7B 12 B2 7A C7 2C 3E   ....;£íýz{.²zÇ,>
00000030   67 76 8F 61 7F C8 1B C3  88 8A 51 32 3A 9F B8 AA   gv.a.È.Ã^ŠQ2:Ÿ¸ª
00000040   4B 1E 5E 4A 29 AB 5F 49  FF FF 00 1D 1D AC 2B 7C   K.^J)«_Iÿÿ...¬+|
00000050   01 01 00 00 00 01 00 00  00 00 00 00 00 00 00 00   ................
00000060   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000070   00 00 00 00 00 00 FF FF  FF FF 4D 04 FF FF 00 1D   ......ÿÿÿÿM.ÿÿ..
00000080   01 04 45 54 68 65 20 54  69 6D 65 73 20 30 33 2F   ..EThe Times 03/
00000090   4A 61 6E 2F 32 30 30 39  20 43 68 61 6E 63 65 6C   Jan/2009 Chancel
000000A0   6C 6F 72 20 6F 6E 20 62  72 69 6E 6B 20 6F 66 20   lor on brink of
000000B0   73 65 63 6F 6E 64 20 62  61 69 6C 6F 75 74 20 66   second bailout f
000000C0   6F 72 20 62 61 6E 6B 73  FF FF FF FF 01 00 F2 05   or banksÿÿÿÿ..ò.
000000D0   2A 01 00 00 00 43 41 04  67 8A FD B0 FE 55 48 27   *....CA.gŠý°þUH'
000000E0   19 67 F1 A6 71 30 B7 10  5C D6 A8 28 E0 39 09 A6   .gñ¦q0·.\Ö¨(à9.¦
000000F0   79 62 E0 EA 1F 61 DE B6  49 F6 BC 3F 4C EF 38 C4   ybàê.aÞ¶Iö¼?Lï8Ä
00000100   F3 55 04 E5 1E C1 12 DE  5C 38 4D F7 BA 0B 8D 57   óU.å.Á.Þ\8M÷°..W
00000110   8A 4C 70 2B 6B F1 1D 5F  AC 00 00 00 00            ŠLp+kñ._¬....
```

Figure 2.2: Genesis bitcoin block[[toto asi dat prec...]]

First documents purchase happened in 22nd May 2010, when Laszlo Hanyecz bought two pizzas for 10 000 bitcoins ($41 then, now about $80 000 000)[2]. This transaction hash is `a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d` and is stored in bitcoin blockchain forever[3].

2.2 DigiByte

DigiByte was developed and released in 2013. It is based on Bitcoin with some adjustment in the code to improving functionality. In late 2017 there was 200 000 pending transaction in Bitcoin. Miners preferred transaction with a higher fee, so to confirm a transaction, user needed to pay $50. DigiByte solved this problem by adding a new block every 15 seconds (new block in Bitcoin is mined every 10 minutes). Average transaction occupies

[1] https://en.bitcoin.it/wiki/Genesis_block
[2] https://bitcointalk.org/index.php?topic=137.0
[3] Actual photos of $80 000 000 pizza are on http://heliacal.net/~solar/bitcoin/pizza/

570 bytes of data. One block can contain approximately 3 500 transactions given the 2 MB limit. This restriction means that in DigiByte, 230 transactions can be confirmed in one second compared to Bitcoin 4-7 transaction per second. DigiByte also has 1000:1 DigiByte to Bitcoin ratio, so for every Bitcoin, there is 1000 DigiByte. [5]

2.3 Ethereum

Ethereum, rather than a cryptocurrency, is a blockchain application platform with Turing-complete programming language. While both the Bitcoin and Ethereum networks are powered by the principle of distributed ledgers and cryptography, the two differ technically in many ways. For example, transactions on the Ethereum network may contain executable code, while data affixed to Bitcoin network transactions are generally only for keeping notes. Other differences include block time (an ether transaction is confirmed in seconds compared to minutes for bitcoin) and the algorithms that they run on (Ethereum uses ethash while Bitcoin uses SHA-256). [23][2]

2.4 Decred

Decred is cryptocurrency build from Bitcoin. The main difference from Bitcoin is the rewarding system from mining. In Bitcoin, the miner gets full reward for a mined block. Sometimes, the Bitcoin blockchain splits when two or more miners found a block at nearly the same time. The fork is resolved when the subsequent block(s) are added, and one of the chains becomes longer than the alternative(s). In Decred the chance of blockchain forks is minimized by hybrid proof of work and proof of state system. Each time a block is created (by a miner), it is not automatically part of the blockchain. Block needs to be approved by ticket holders. Then miner receives a block reward (newly created DCR). If the block is rejected by ticket holders, the miner does not receive a reward. Tickets holders for a new block are chosen randomly. The ticket validates the previous block. A block needs at least three of the five votes chosen to approve it for it to be validated. This hybrid system has many implications, including making a 51% power attack very difficult, and making a minority fork very difficult as well. [10]

2.5 Monero

Three years after Bitcoin, in 2012, the competing Bytecoin cryptocurrency entered the market. The problem with this cryptocurrency, however, was that 80% of all coins were mined in advance by its authors. The chances of mining were, therefore, not balanced. This injustice led to the decision that this cryptocurrency would start again. New cryptocurrency starts on 18 April 2014 and was called BitMonero, a composite of the word coin in Esperanto (Monero) and Bitcoin according to Bitcoin. However, after five days, the community decided to use only Monero for short. Monero's significant advantage is the dynamic size of the mined blocks. Bitcoin has one block size limited to 1 MB, while Monero adapts the block size to the network load. If the number of transactions increases, so does the block size to accommodate all transactions. Thus, unlike Bitcoin, the more transactions users make, the lower the transaction fee. Monero's main benefit is its full anonymity and interchangeability thanks to CryptoNote protocol[21]. Monero hides recipient and sender addresses. [19]

2.6 Analyse of current blockchain explorers

A Blockchain Explorer is a web application that allows us to explore the whole blockchain. Their primary function is to allow everyone with an internet connection to track in real-time all the transactions or interactions made by each cryptocurrencies holders, regardless of his or her level of knowledge and expertise. [14][7]

2.6.1 Blockbook

As a representative of traditional explorers, we chose Blockbook. Blockbook[4] is a blockchain indexer for Trezor Wallet[5], developed by SatoshiLabs[6]. It currently supports more than 30 coins (and the community implemented some others). For data storage, Blockbook is using RocksDB[7] developed by Facebook, which is a NoSQL database that stores only key-value pairs. Blockbook is providing fast API for accessing blocks, addresses and transactions. Main limitations of Blockbook are:

- **Not distributed** (client-server architecture) – problem with scaling for more users.
- **Not an SQL database** – it does not have a relational data model, it does not support SQL queries, and it has no support for indexes.
- **Single-Process** – only a single process (possibly multi-threaded) can access a particular database at a time.

[4] https://github.com/trezor/blockbook
[5] https://wallet.trezor.io/
[6] https://satoshilabs.com/
[7] https://github.com/facebook/rocksdb/wiki

Chapter 3

IPFS

IPFS[1] stands for Interplanetary File System and is a peer-to-peer distributed file system designed to make the Web faster, safer, and more open. In contrast with a standard file system, objects in IPFS are content-addressed by the cryptographic hash of their contents. In the case of the standard Web, when user wants some file, he needs to know on which server is a file located and the full path to the file (see Figure 3.1). Principle of the IPFS is that a user needs only to know the hash of the requested file. He does not care about the location of the file (see Figure 3.2). Let us take an MIT licence text, and add it to IPFS. If somebody tries to add this licence as a file to IPFS, it will return `QmWpvK4bYR7k9b1feM48fskt2XsZfMaPfNnFxdbhJHw7QJ` every time. That is now and will be in the future, the *content address* of that file. Later, when a user tries to get this file by its hash, he can get it from a random person that added it into IPFS in the past.

IPFS can easily represent a file system consisting of files and directories. A small file (less than 256 kB) is represented by an IPFS object with data being the file contents (plus a small header and footer). Note that the file name is not part of the IPFS object, so two files with different names and the same content will have the same IPFS object representation and hence the same hash. A large file (more than 256 kB) is represented by a list of links to file chunks that are less than 256 kB, and only minimal information specifying that this object represents a large file. Currently, there are no known size limitations uploaded file or directory. There are already some big datasets hosted on IPFS such as Geocities archive[2] (704 TB) or Project Apollo Archives[3] (61 GB).

3.1 IPFS stack

We can split IPFS into layers (see Figure 3.3). *Libp2p*[4] is at the bottom, which is a peer-to-peer networking module, that handles peer and content discovery, transport, security, identity, peer routing, and messaging. *IPLD* is the data model of the content-addressable web. It is providing linking between objects and multihash computing. On the top is *IPFS*, which allows publishing and share files (or any data). [1]

[1] https://ipfs.io/
[2] https://ipfs.io/ipfs/QmVCjhoEFC9vwvaa8bKyJgwAByP4MXSogcyDGoz4Lkc3ox
[3] https://ipfs.io/ipfs/QmSnuWmxptJZdLJpKRarxBMS2Ju2oANVrgbr2xWbie9b2D
[4] https://libp2p.io/

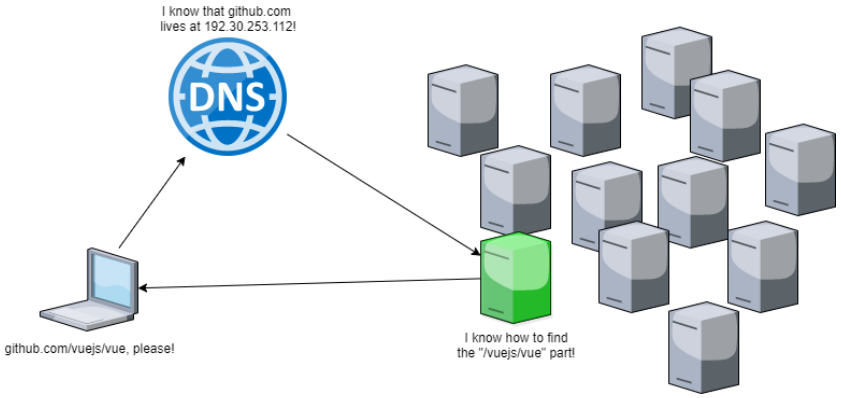


Figure 3.1: Classic web addressing

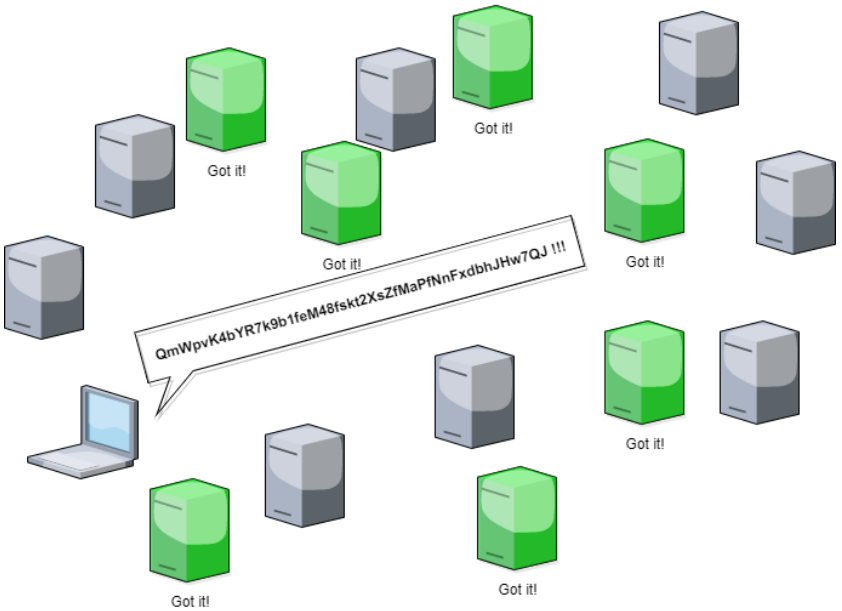


Figure 3.2: Content-based addressing

3.2 libp2p

Libp2p is a modular system of protocols, specifications and libraries that enable the development of peer-to-peer network applications. It provides NAT Traversal, Peer Discovery, Routing, Stream Multiplexing, Protocol Multiplexing, Encryption, Authentication and more. It grew out of IPFS to solve networking problems in p2p networks, but now it does not require or depend on IPFS. Today many projects use libp2p as their network transporting layer, which is responsible for the actual transmission and receipt of data from one peer to another. For both content discovery and peer routing, libp2p uses Kademlia-based distributed hash table. With Kademlia, libp2p iteratively routes requests closer to the desired peer or content using Kademlia routing algorithm[16]. In the future, Kademlia might be changed easily to some other solution that implements a simple interface for publishing and requesting data and finding a peer. [8]

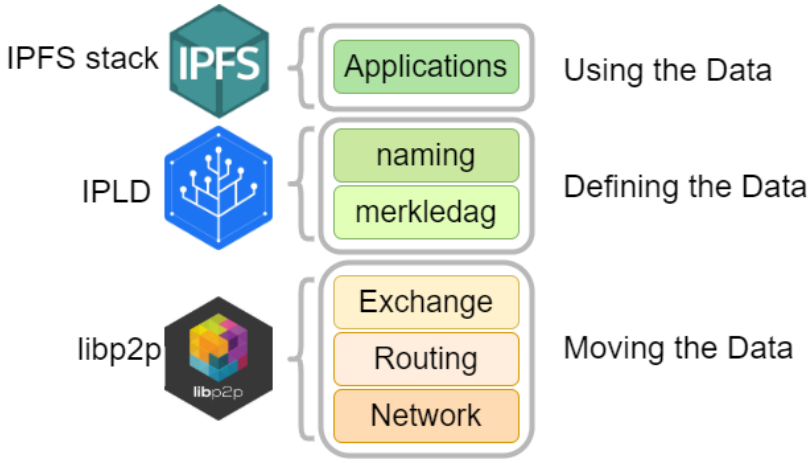


Figure 3.3: IPFS stack

Property		**Value**
Multibase		base58btc
Version		cidv0
Multicodec		dag-pb
Multihash	Hash Type	sha2-256
	Hash Length	256
	Hash	7e1b666c0327...3dc3022f

Table 3.1: Example of a human-readable version of CID

3.3 IPLD

IPLD is providing linking and addressing objects with CID (Content ID). CID is hash-based self-describing content identifier (usually encoded to base58[5] format) which includes codec and multihash. Multihash is then further composed of hash type and hash value. Let us look closer on the MIT licence file, that we add to IPFS at the beginning of this chapter (see Figure 3). Its CID is `QmWpvK4bYR7k9b1feM48fskt2XsZfMaPfNnFxdbhJHw7QJ`. It can be converted to human-readable format as can be seen in Figure 3.1, thanks to multicodec table[6]. We can see that this CID is encoded in base58 format and the file was stored using protobuf[7] codec (this information is necessary to decode file correctly).

3.4 IPFS

IPFS is the top layer from the IPFS stack. It is used for pinning objects and files, naming system and keys management. File or object is automatically pinned when a user adds it (but other IPFS commands do not include automatic pinning). Pinning a CID tells an IPFS server that the data is essential and must not be thrown away. When a garbage collector is triggered on a node, any pinned content is automatically exempt from deletion. Non-pinned data may be deleted. The Interplanetary Name System (IPNS) is a system for

[5] https://en.wikipedia.org/wiki/Base58
[6] https://github.com/multiformats/multicodec/blob/master/table.csv
[7] https://en.wikipedia.org/wiki/Protocol_Buffers

creating and updating mutable links to IPFS content. Since objects in IPFS are content addressed, an object address changes every time an object's content changes. A name in IPNS is the hash of a public key. It is associated with a record containing information about the hash it links to that is signed by the corresponding private key.

3.4.1 IPFS core APIs

IPFS provides several APIs that are working on different abstraction levels. The regular top-level API is Files API[8]. The Files API enables users to use the File System abstraction of IPFS. It has `add`, `cat`, `get` and `ls` methods for manipulating with regular files and directories. For our system are low-level APIs such as `DAG`[9] and `PUBSUB`[10] more interesting. PUBSUB API is used to broadcast messages between peers, and it consists of these methods:

- `subscribe` – listen to specific topic. A topic name and handler function is provided as a param.
- `unsucribe` – stop listening on a topic that is provided as parameter.
- `publish` – publish a message to topic. All peers that subscribed the topic receive the message.
- `ls` – returns the list of subscriptions the peer is subscribed to.
- `peers` – returns the peers that are subscribed to a topic. A topic name is a parameter of this function.

The DAG (stands for Direct Acyclic Graph[11]) API provides these methods for manipulation objects in IPFS:

- `put` – stores object in IPFS. Return IPFS hash of a stored object.
- `get` – retrieves an object by its hash from IPFS.
- `tree` – Enumerate all the entries in a graph.

3.5 IPNS

Naming inside IPFS is governed by IPNS[12], the Inter-Planetary Naming System. IPNS takes ideas from SFS[13] to enable the creation of cryptographically signed mutable pointers, which can be used to the creation of name records inside the network.

3.6 Filecoin

Another impressive part of the IPFS ecosystem is Filecoin. It is a decentralized storage network that turns cloud storage into an algorithmic market. The market runs on a blockchain

[8] https://github.com/ipfs/interface-js-ipfs-core/blob/master/SPEC/FILES.md
[9] https://github.com/ipfs/interface-js-ipfs-core/blob/master/SPEC/DAG.md
[10] https://github.com/ipfs/interface-js-ipfs-core/blob/master/SPEC/PUBSUB.md
[11] https://en.wikipedia.org/wiki/Directed_acyclic_graph
[12] https://docs.ipfs.io/guides/concepts/ipns/
[13] https://en.wikipedia.org/wiki/Self-certifying_File_System

with a native protocol token (also called "Filecoin"), which miners earn by providing storage to clients. Conversely, clients spend Filecoin hiring miners to store or distribute data. As with Bitcoin, Filecoin miners compete to mine blocks with sizable rewards, but Filecoin mining power is proportional to active storage, which directly provides a useful service to clients (unlike Bitcoin mining, whose usefulness is limited to maintaining blockchain consensus). This principle creates a powerful incentive for miners to amass as much storage as they can, and rent it out to clients. The protocol weaves these amassed resources into a self-healing storage network that anybody in the world can rely on. The network achieves robustness by replicating and dispersing content, while automatically detecting and repairing replica failures. Clients can select replication parameters to protect against different threat models. The protocol's cloud storage network also provides security, as the content is encrypted end-to-end at the client, while storage providers do not have access to decryption keys. Filecoin works as an incentive layer on top of IPFS.[12]

3.7 Existing blockchain explorers in IPFS

There are already stored a few blockchains of cryptocurrencies in IPFS. For browsing them we can use dedicated applications[14] [15] or IPLD explorer[16]. Blockchains in IPFS are stored in raw binary format, so custom IPLD codec has to be created for every type of object (there are ten different codecs only for Ethereum blocks, transactions, state tries, accounts, contracts etc.). Using custom codecs for cryptocurrencies allows explorers to request blocks and transactions by its hash very fast, but there are also limitations. Mainly with filtering or requesting data by different property as a hash.

- Existing IPLD codecs for cryptocurrencies are very limited. Only a few of cryptocurrency codecs are currently available in IPLD (namely Leofcoin, Ethereum, Bitcoin, Zcash, Steller, Decred and Dash)[17].
- Addresses are not stored in IPFS, because they are not part of a blockchain. This means the explorer needs to go through the entire blockchain for computing address balance or to find address transactions.
- No additional information (for example transaction value in US dollars) can be stored with objects because it would change content, and therefore hash of the object.
- There is no sorting or filtering. Explorer can only show the object (block or transaction) by its hash.

[14] https://github.com/arcalinea/IPFS-Zcash-Explorer
[15] https://github.com/whyrusleeping/zcash-explorer
[16] https://explore.ipld.io/#/explore/z43AaGEvwdfzjrCZ3Sq7DKxdDHrwoaPQDtqF4jfdkNEVTiqGVFW
[17] https://github.com/multiformats/multicodec/blob/master/table.csv

Chapter 4

Design

This chapter describes the proposed design of the whole system for storing and exploring blockchains in IPFS that is to be created as a result of this thesis. All parts of the system are described in this chapter.

4.1 Database system

We need a database in our system to store and index data. Database system needs to be decentralized and distributed. There are several databases build on top of IPFS already implemented. The two most known are OrbitDB and Textile.

4.1.1 OrbitDB

OrbitDB[1] is a serverless, distributed, peer-to-peer database build on top of IPFS, developed by HAJA networks. OrbitDB is a decent solution for small user's databases. However, it is still in the alpha stage of development, and it is not well optimized for storing hundreds of gigabytes of data. The biggest problem is that OrbitDB performs all queries locally. To perform a query that selects transactions that are more valuable than 1BTC, OrbitDB needs to load the whole database locally and then perform a cycle on all transactions to select only those transactions that meet the criteria. So every client ends up with a full copy of the database. This limitation is not usable for our case when we have a database that has hundreds of gigabytes of data.[15]

4.1.2 Textile

Textile[2] is a set of open-source tools that provide a distributed peer-to-peer database, remote storage, user management, and more, over the IPFS network. Textile already created applications for storing photos, notes or anything else (Anytype). Textile provides a high abstraction on top of the IPFS and provides simple API to store and index files securely. It uses Cafe peers to provide backups and indexing. Data are duplicated on several Cafe peers. When a client is performing some query, it contacts one of the Cafe peers to resolve the query for the client. Neither of these solutions fits our use case to store a large amount of data distributed on several nodes and performs queries that can be

[1] https://orbitdb.org/
[2] https://textile.io/

resolved by downloading only necessary parts of the database. Therefore, we need to create a new database system based on IFPS that would be decentralized and distributed.[20]

4.1.3 Database system design

After some research, we concluded that currently for storing and indexing data in IPFS without large hard disk memory consumption, there is no solution. We created our own indexing system that currently supports three types of indexes. A database system that fits our needs is distributed and decentralized. That brings us lots of synchronization problems to solve. This database system consists of tables that contain records. For faster searching, tables have indexes. Relations between tables are represented via foreign keys. Also, this database system supports fluent query language, used for performing complex queries.

Record

Every record in our database system is stored in an append-only log that contains the whole history of the record. Every update of a record adds a new entry to its log which points to the previous entry.

In centralized systems such as git[3], conflicts are detected on write (for example, when two git users push changes to the server at the same time, one of them gets an error and needs to pull repository)[3]. This approach is impossible in a decentralized system. When we update some record in our database, we can not know if somebody updates it before us (and we do not receive changes yet). For this reason, we need to solve conflicts while reading. Record with conflict has more than one head in a record log, and users need to solve them in application logic. Look at the example in Figure 4.1. There is a record that is updated by every country in the World when they have new statistics about a pandemic. If two countries updates data at the same time (their updates are pointing to the same previous version of the entry), they create conflict. Luckily, this specific conflict is easy to solve. We just need to look at the previous entry and compute increment for both countries, then update the record with final increment.

Indexes

An index is a B-tree structure optimized for IPFS (with no cycled references and node size less than 256 kB). Each table has at least one primary index. We use the primary index to reference record in foreign keys. A primary key is automatically created when a user does not specify it and has a type of Guid. Value of the primary key for an entity can not be changed, because we would need to scan all tables where the entity is referenced and change the referenced value to a new one. Also, when we execute a query without any condition or sorting, we use a primary key to obtain records. Every index has several components:

- **Comparator** – is a function that has two parameters (two B-tree keys) and outputs a number as a result. An index is using a comparator to correctly traverse B-tree while searching records and inserting records to the right place.
- **Key-getter** – is a function that returns a comparable object from the record (a comparable object is an object that can be compared using index Comparator). Every index has Key-getter that is using to obtain keys for objects stored in B-tree.

[3] https://git-scm.com/

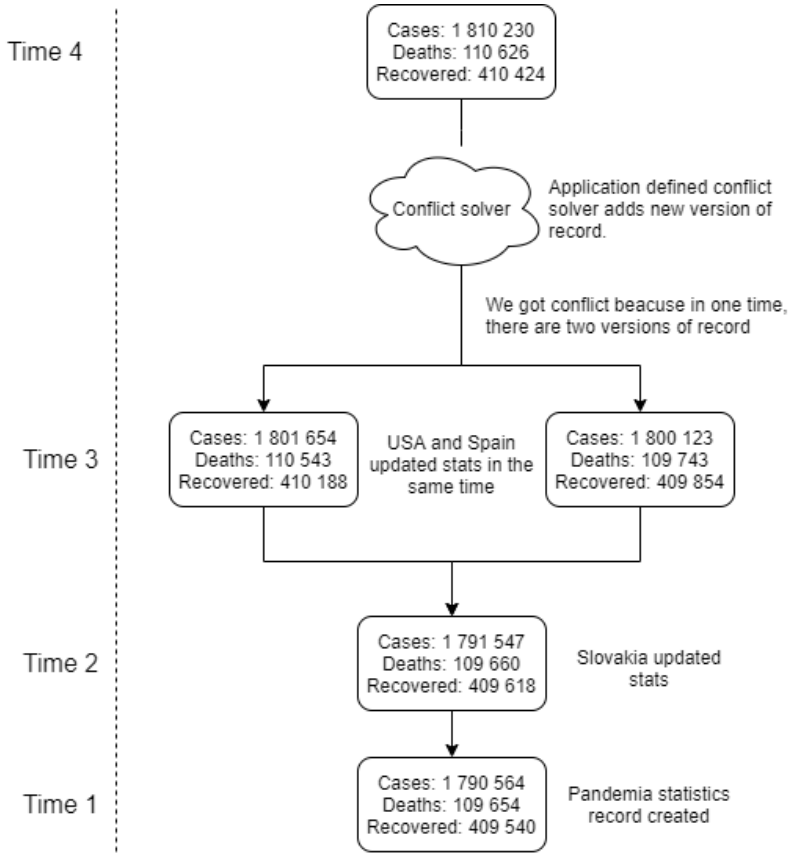


Figure 4.1: Record conflict

Table

A table contains indexes and table name. Also, it implements operations that modify its data:

- **Insert** – creates a new record in a table. Insert operation has these steps:
  1. Creates record log with a new entry and store it in IPFS
  2. Gets record key (via Key-getter) and inserts it to all table indexes with IPFS hash as value.
- **Update** – save new record version. It consists of:
  1. Add new entry to the record log. Save the record log to the IPFS.
  2. Get new record key (via Key-getter) and update all table indexes with new IPFS hash as value.
- **Delete** – remove a record from a table.
  1. Get old record key and remove it from all table indexes.
  2. Add an empty entry to the record log. Save the record log to the IPFS.

Foreign keys

We use foreign keys to represent relationships between tables. A foreign key is simply a table name and a value of a primary key of a referenced record.

Transactions

When we commit a transaction, it is inserted to the transactions queue. If there are more transactions in the queue, a database performs it one by one. We can not execute more transactions in one moment, because it could cause data inconsistency. There are five types of transactions:

- `read` – transaction for reading data from table. It is not logged in a database log (other peers don't have to know what we are reading from database).
- `sync` – when some peer publishes a new version of the database, all other peers have to migrate to it. This transaction has the biggest priority, and therefore it is inserted at the beginning of the transactions queue. After the migration is done, transactions queue can continue working usually.
- `insert`, `update`, `delete` – these types of transactions are written to database logs, because they are modifying the database.

Synchronization

Every time a transaction queue is empty (all pending transactions has been applied), a new version of a database is broadcasted to all connected users via IPFS PUBSUB API (see 3.4.1). When we receive information about the new database version, we load its root. Database versions create an append-only log called database log so each database version has information from which version it was created. Database log is an immutable, operation-based conflict-free replicated data structure (CRDT[4]) for storing database versions. Every version in the log is saved in IPFS, and each points to a hash of the previous version forming a graph.

If more than one peer publishes a new version of the database that has been created from the same database version, other peers need to decide which version they would accept. Opposite to records conflicts, we need to solve database versions conflict fast and automatically. There are multiple ways to solve them. For example:

- **The biggest hash wins** – if there is more than one database version at the same (discrete) time, the one with the biggest IPFS hash wins. This strategy is present in Figure 4.2.
- **The longest connecting time wins** – a peer that is connected to the database for the longest time wins.

Lots of more strategies can be implemented, but they need to be deterministic and as fair as possible. If we have access to geolocation, we can implement a strategy based on a distance to the North Pole (the closest node to the North Pole wins).[13]

[4] https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

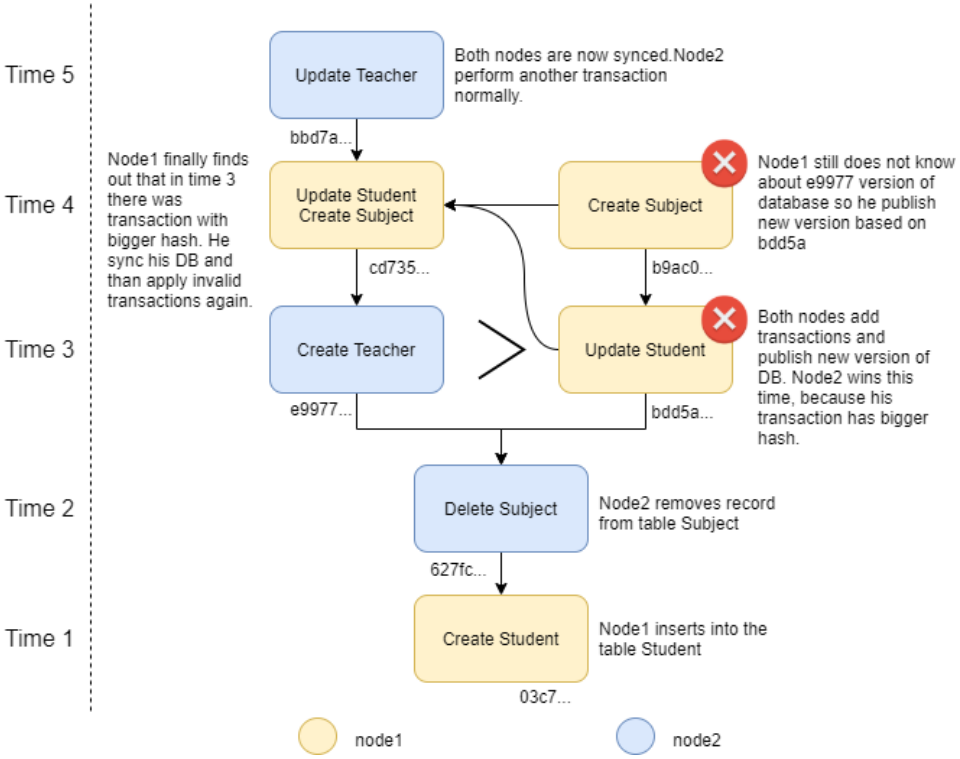


Figure 4.2: Synchronization of a database. Note that in time 4, Node1 adds two transactions in one DB version.

Queries

A Database system provides query language for performing selects. A query consists of:

- **conditions** – query may contain multiple conditions. There are logical operators (and/conjunction, or/disjunction) between conditions. When it is possible (an index is available on condition property), we use indexes for evaluating conditions. If there are more than one conditions, we use an intersection between `AND` conditions and union between `OR` conditions. These types of conditions are supported:
  - **equal** – record property equals to specified value,
  - **greater than** – record property is greater than specified value,
  - **less than** – record property is less than specified value,
  - **between** – record property is greater than specified minimal value, and less than specified maximal value.
- **filters** – filters are similar to conditions, but they can be more complicated. They are functions that are being applied to query results. If any filter return `false` for the query result, the result is ignored. A query can contain several filters.
- **offset** – offset is saying how many query results should be ignored from the beginning.
- **evaluator** – we use evaluator for accessing results of the query.
  - **all** – returns array of all results of the query.

– **first** – returns only first result.

– **take** – returns `N` number of results where `N` is an argument.

– **paginate** – returns page of results. Accept two arguments. `perPage` specifies the number of results in one page. `page` is number of requested page.

– **iterate** – returns iterator that can be used in cycles such as `for` or `while`.

4.2 System components

The system consists of one or more Feeders and Explorers. Feeders are connected to data sources and provide synchronization with cryptocurrency blockchains. Explorer can request data from the system network and display them to a user (see Figure 5.1).

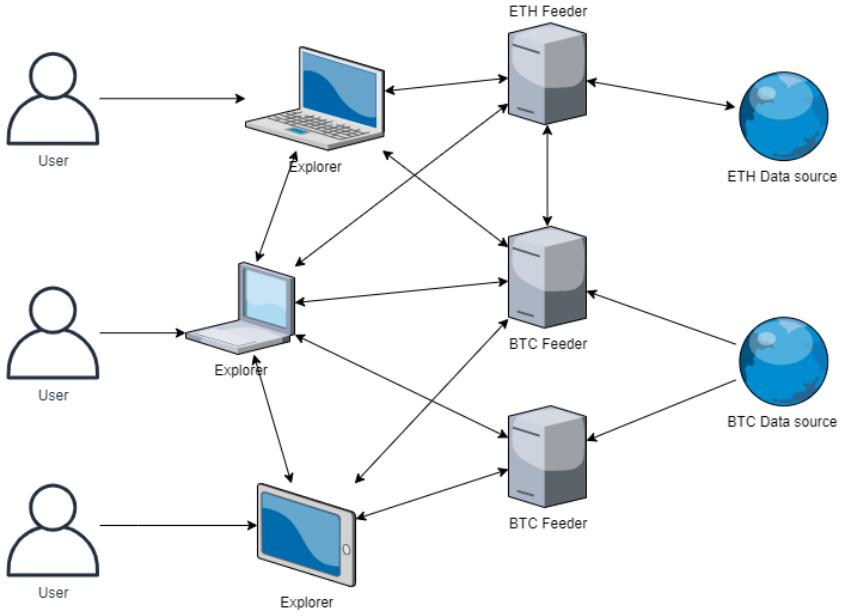


Figure 4.3: System architecture

4.2.1 Database design

Both Explorer and Feeder use the same database schema. The database schema is designed in the way to optimize typical queries. It consists of five tables:

- **Block** – table for storing blockchain blocks. It has a primary key on unique block hash, and indexes on height and time. With these indexes, we can perform queries like search block by its hash or height very efficiently. Also filtering or ordering blocks by time has good performance.
- **Transaction** – a table that contains blockchain transactions has a primary key on transaction hash and foreign key, that references block in which is transaction confirmed, on `blockHash` column. Indexes for efficient filtering are on `blockHeight` and `blockTime` columns.

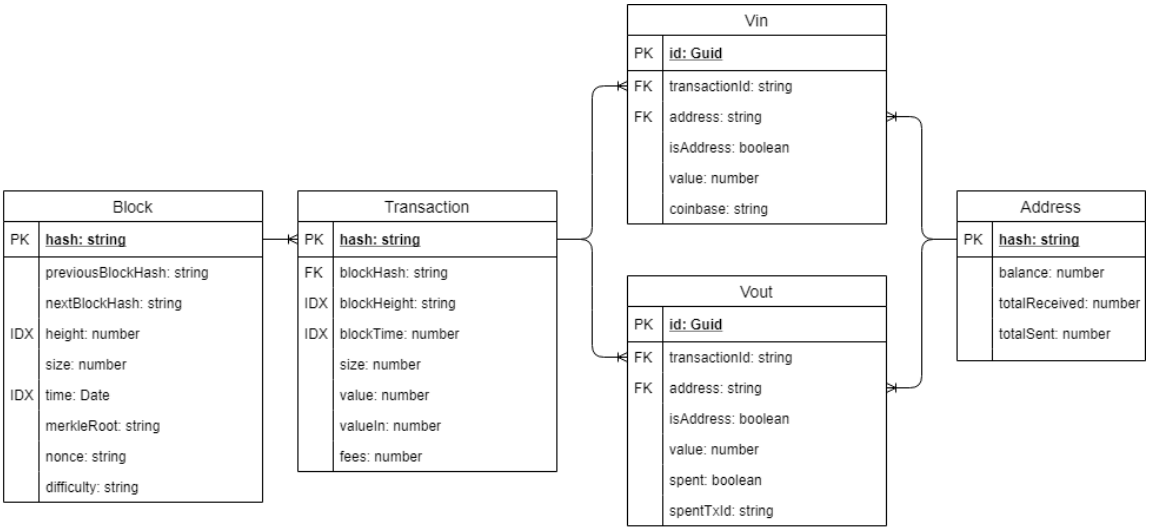


Figure 4.4: Entity-relationships diagram

- **Vin** – every transaction can have multiple inputs. These inputs are stored in Vin table. Vin has not got any unique property, so a custom Guid[5] needs to be created as a primary key. It has two foreign keys to reference transaction and address. Foreign key `address` can be null, because the input can be mined block reward.
- **Vout** – transactions can have multiple outputs. Every output has a foreign key to transaction and address. If the output is already spent, it has a link to the transaction where this output is used as an input.
- **Address** – Address contains multiple columns. As the primary key, it uses the address hash. Address also contains its `balance`, `totalReceived` and `totalSent`. These values are updated every time new input or output is added to the transaction.

4.2.2 Feeder

A Feeder is a service that stores data in IPFS and indexes them in our database system. Once all blocks are indexed, Feeder waits for new blocks and is periodically publishing most up to date database version. This way, new clients synchronize database more quickly. For optimization purposes, Feeder is performing transactions in bulks. This optimalization means that Feeder is not publishing a new version of the database every time it parsed block or transaction, but rather after it parsed transactions bulk which can consist of hundreds of transactions.

A Feeder can use different sources for obtaining blockchains data. It can be connected directly to the blockchain as a full node or to some other data source providing blockchain data as Blockbook[6] or Insight[7].

[5] https://en.wikipedia.org/wiki/Universally_unique_identifier
[6] https://github.com/trezor/blockbook
[7] https://insight.is

4.2.3 Explorer

Explorer can perform basic queries like a search for block by its height or hash and search address and transaction by hash. Nevertheless, Explorer can also make more complex queries (for example, get the first 20 transaction where the sum of inputs is more than some value, or get transactions between some time interval). Explorer is an application that can be used in two environments. In browser with graphical user interface, or in node.js[8] as an API.

ExplorerGUI

ExplorerGUI is browser version of Explorer. It is implemented as an SPA[9] to prevent restarting connection with IPFS after each time a user visits a different page. It has separated views for block, transaction and address details. Also paginating, filtering and sorting objects by all its properties that have index is supported. Every user of ExplorerGUI keeps part of the database that he uses in his local storage. This principle helps balance Feeders load and helps to distribute data for other users. Authentication to the ExplorerGUI is not necessary, because users can not modify data from there. They can only explore multiple blockchain data. All views have the same top toolbar, to allow user performs quick searches. User can see enabled cryptocurrencies at the home screen (see Figure 4.5a). When a user selects cryptocurrency that he wants to explore, he will be redirected to the Blocks view where he can see and filter all parsed blocks from a selected cryptocurrency (see Figure 4.5b). User can select block by clicking on them. This action redirects a user to Block detail view where he can see all blocks transactions (see Figure 4.6a). User can also navigate to the next/previous block from this view. A user gets to the Address view every time he clicks on transaction input or output address. In this view, a user can see the current balance of the address and transactions which this address is part of (see Figure 4.6b). Thanks to the record log, a user can also see the history of the address.

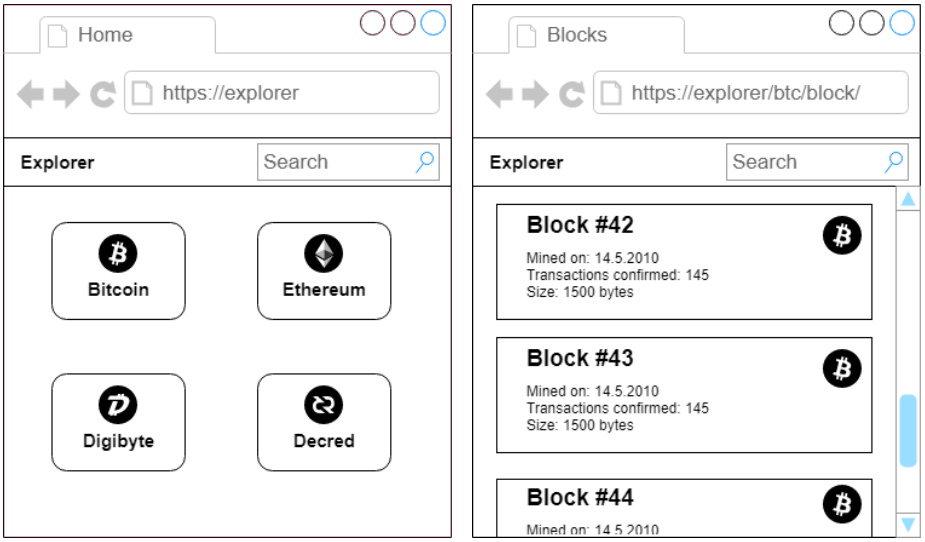
ExplorerAPI

ExplorerAPI is a server-side application that provides a simple HTTP API for obtaining data. Although our system is decentralized and therefore does not need servers, ExplorerAPI can be used with older devices on which it is not possible to run an IPFS node, or when integrating with other applications. Through RestApi, we could not use the full potential of our database system, so we get inspired by GraphQL[10] during the design. ExplorerAPI provides endpoints for these database tables:

- `/:currency/block` – endpoint for obtaining blocks. Each endpoint hash `currency` path parameter that can be currency unit (for example `btc`) or currency name (for example `bitcoin`).
- `/:currency/transaction` – endpoint for retrieving transactions.
- `/:currency/address` – endpoint for obtaining data from address.

Every endpoint can be accessed with `GET` or `POST` HTTP method. A request contains a query object that consists of conditions, filters, skip and resolver (see Figure 4.7a). This

[8] https://nodejs.org/
[9] https://en.wikipedia.org/wiki/Single-page_application
[10] https://graphql.org/

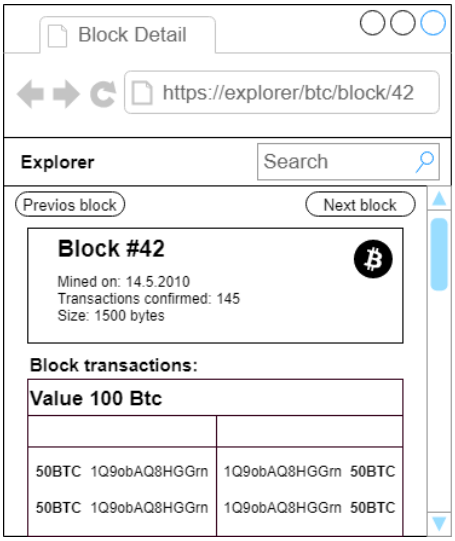


(a) Home view (b) Blocks list view

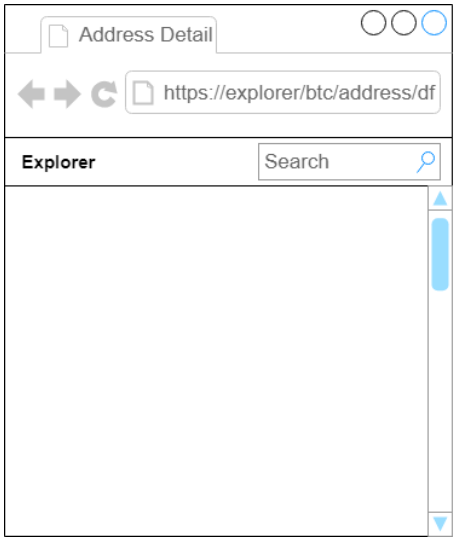
Figure 4.5: ExplorerGUI mockups

query object is translated to our query system and executed. Query results in JSON format are returned (example result is on Figure 4.7b). If a request is sent by `GET` method, a query object is stringified and urlencoded[11] JSON in query parameters (for example `/block?query=%3D%7Bconditions...%7D`). If `POST` method is used, query object is in request body (like on Figure 4.7a). With this design we can use multiple conditions and filter in query.

[11] https://en.wikipedia.org/wiki/Percent-encoding



(a) Block detail view



(b) Address view [[dorobit mockup]]

Figure 4.6: ExplorerGUI mockups

```
{
    "query": {
        "conditions": [
            {
                "type": "and",
                "comparator": "between",
                "property": "height",
                "values": [11990, 12000]
            }
        ],
        "filters": [
            "(b) => b.height % 2"
        ],
        "resolver": {
            "type": "all"
        }
    }
}
```

(a) Request that returns blocks with an odd height that is between 11990 and 12000.

```
[
    {
        "hash": "000000001e564...",
        "height": 11991,
        ...
    },
    {
        "hash": "00000000b3637...",
        "height": 11993,
        ...
    },
    {
        "hash": "000000007b4da...",
        "height": 11995,
        ...
    },
    {
        "hash": "00000000e4361...",
        "height": 11997,
        ...
    },
    {
        "hash": "00000000e9935...",
        "height": 11999,
        ...
    }
]
```

(b) Response

Figure 4.7: ExplorerAPI example

Chapter 5

Implementation

This chapter describes the implementation details of the system and shows the internal architecture. All system components are implemented in Typescript that is later compiled to JavaScript. This choice of language enables code sharing between individual components and allow us to support multiple platforms (desktop via Node.js, and browser). All system components are dependent on Explorer-core module where is the whole database system implemented (see Figure 5.1).

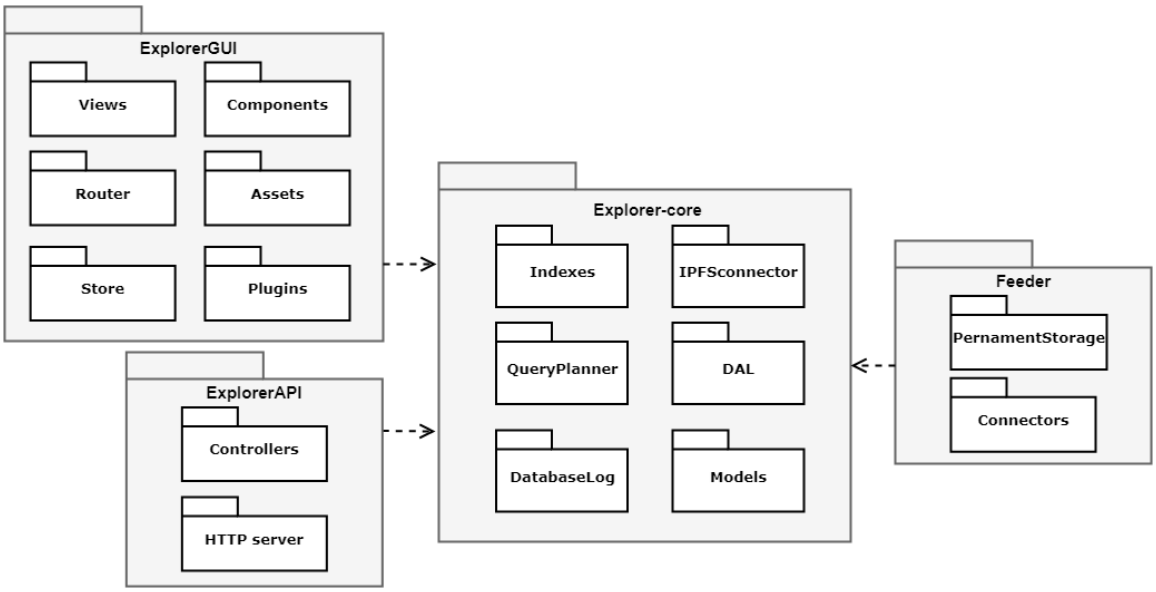


Figure 5.1: System architecture

5.1 Explorer-core implementation

Explorer-core is the most complex module of a whole system with more than 5000 lines of code. The database is a main part of the Explorer-core module. Database system consists of a query system, indexes and an abstract database layer. Explorer-core is only system components that communicates with IPFS via js-ipfs[1] implementation.

[1] https://github.com/ipfs/js-ipfs

5.1.1 IPFS connector

IPFS connector is a singleton class that provides a connection to IPFS. It has a method `getInstanceAsync` which will returns promise that resolves into IPFS node instance. This instance is stored in a private static class variable. Next call of function `getInstanceAsync` returns that static class variable. Also, swarm key is present in IPFS connector. This key is used to make private IPFS network with only Explorers and Feeders peers. Other peers (without swarm key) can not connect to our network. IPFS connector contains settings for node.js and browser IPFS peer. Browser peers use WebRTC and WebSockets for transport. Node.js peers use TCP and WebSockets. That means that Feeders (a node.js application) can communicate with each other with TCP. Explorer communicates with other explorers with WebSockets or WebRTC. Explorer uses WebSockets when communicating with Feeders.

5.1.2 Indexes

For indexing there is currently implemented only B-tree, but different structures such as tries[2] can be implemented easily. They only need to implement index interface (function such as `insert`, `delete`, `update`, `find`). We can create indexes on database entities with decorators[3], which are part of ECMAScript 6 standard. Every database entity has to have `PrimaryKey` decorator on property that is used as a primary index. Index has three parameters:

- `comparator` – is a function that accepts two arguments and returns number that is less than zero if first arguments is greater than a second, zero if arguments are the same, more than zero if a second argument is greater than first. If a user has not set any custom comparator default one (`(a, b) => a < b ? -1 : +(a > b)`) is used. This default comparator works on atomic keys such as string or numbers.
- `keyGetter` – is another function that accepts a whole entity as arguments and returns key, that is used in an index. A default key getter is function that returns value of index property (for example default key getter for property `height` is `(e) => e["height"]`).
- `branching` – the branching factor is the number of children at each node, the outdegree. Default branching factor is 16.

The B-tree structure (see Figure 5.2) is optimized for IPFS. Unlike PostgreSQL B-tree[4], leaf nodes have not got left and right sibling references due to cyclic reference that is not possible in IPFS, because links are changing CID of an object. Therefore, if we store object `A` with a link to the object `B`, CID of `A` will change. Then if we add backlink from `B` to `A`, CID of `B` will be changed and so `A` is now linked to an old version of `B`. If we update link of object `A` to point on the new version of object `B`, CID of object `A` will change and therefore object `B` is now pointing to the old version of object `A`. From this example, it appears that there is no way of making cyclic references in IPFS. Without siblings references, we don't have to store data only in leaves, but we can some store data in nodes itself. This approach leads to better performance when executing queries. In insert queries, statistically fewer

[2] https://en.wikipedia.org/wiki/Trie
[3] https://www.typescriptlang.org/docs/handbook/decorators.html
[4] https://github.com/postgres/postgres/tree/master/src/backend/access/nbtree

nodes need to be updated. Also in search queries, there is a chance that we found a key in some non-leaf node, and therefore we would need fewer node visits.

All search queries (equal, less, greater, between) has two steps:

- **Find subtree** – First, we need to find minimal subtree that contains all results. We start in a root node of the B-tree. Then, we use an index's `comparator` function to determine if we can visit some child node and still have all the result in it. If not, we found a minimal subtree for the query.
- **Traverse** – To get results of the query, we can traverse minimal subtree in two directions. In-order for `greater than` and `between`. For `less than` reversed in-order. With `equal`, it does not matter.

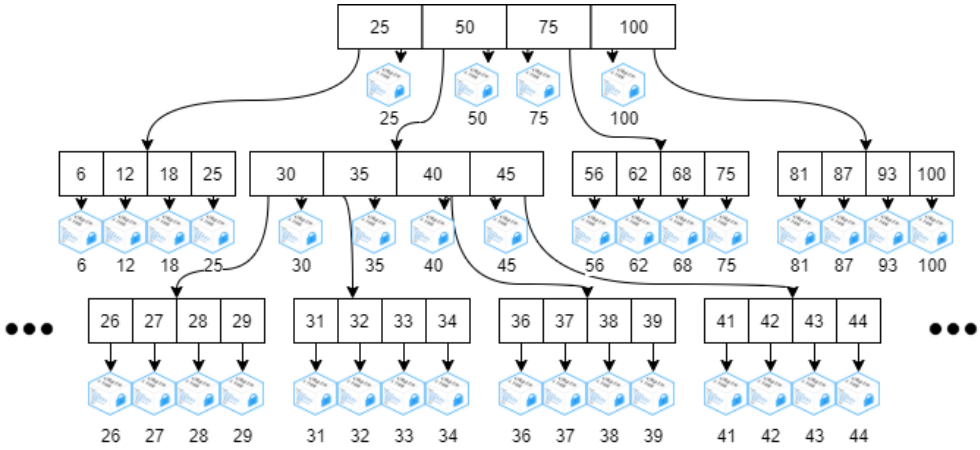


Figure 5.2: B-tree indexing first 100 blocks by their height

5.1.3 Query system

Database system offers a complex query system. A query can consist of multiple conditions, and the `Query Planner` is responsible for resolving them. It decides which indexes are used for query and choose a strategy. If there is no condition, a primary key is used for query execution. In the case of a single condition, `Query Planner` checks if there is an index on the condition's property. If yes, then this index is used to perform the query. Else condition is transformed to filter, and a primary key is used to obtain results which are then filtered. For multiple conditions connected with logical operators `AND` or `OR`, `Query Planner` creates `OR-hashset` and `AND-hashset`. `AND-hashset` is initialized with the results of a condition which has the smallest number of results and uses `AND` operator. Then we check for each result hash for all other `AND` conditions if `AND-hashset` contains it. If no, a hash is deleted from `AND-hashset`. This creates intersection between all `AND` conditions. The `OR-hashset` is created empty. Then, we add a hash of every result of all `OR` conditions to it. This creates union between `OR` conditions. At the end, we perform union between `AND-hashset` and `OR-hashset`. This creates final hashset which is used later to obtain actual data with resolvers such as `all`, `first`, `paginate` etc. If one of the `AND` conditions has great selectivity (it has less than a hundred results), `Query Planner` can decide to don't use other indexes, but transform condition to filter and cycle over the results. Example query returning blocks between 38 and 42 is shown in Figure 5.3. A peer needs to download only data that are

highlighted in green. After downloading them, they are stored on a peer's filesystem and are offered to other peers.

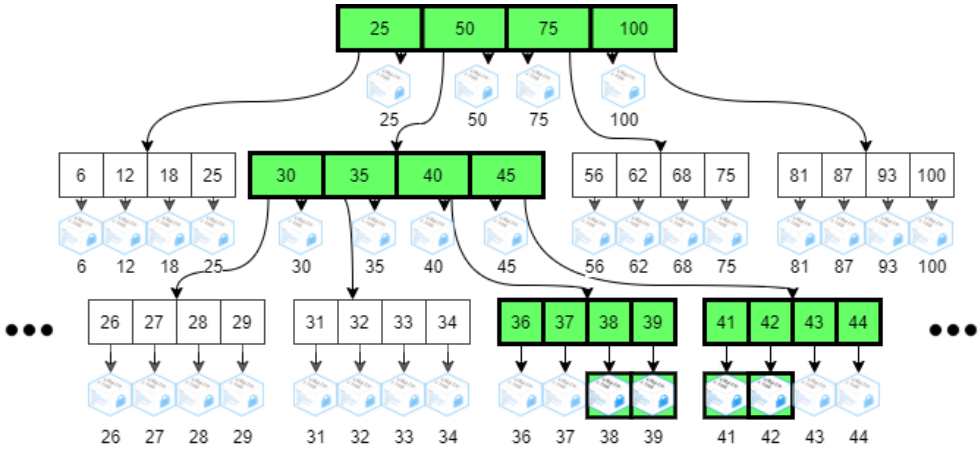


Figure 5.3: Data access for performing query that returns blocks that have a height between 38 and 42.

We can make queries with every database table model (such as model on Figure 5.5). Whole query system is displayed in the state diagram on Figure 5.4. The query system consists of these functions:

- `where(propertyName)` – create a condition on the property. There can be multiple conditions in one query. A condition hash to be followed by one of the functions:
  - `gt(value)` – property is greater or equal than `value`. The QueryPlanner finds in an index first object that has property (set by `propertyName` in `where` function) equal or greater than `value`, and traverse index to the right (to bigger objects).
  - `lt(value)` – property is less or equal than `value`. Similar as in the `gt` function, the QueryPlanner finds the closest object that has property equal or less than `value`. Then, the query traverses index to the smaller objects with smaller index value (to the left).
  - `between(min, max)` – property is greater or equal than `min` and less or equal than `max`.
  - `equal(value)` – returns all objects that `keyGetter` function returns same key as `value`.
- `skip(offsetValue)` – query will skip `offsetValue` number of results,
- `limit(limitValue)` – set maximum number of results. After query has `limitValue` count of matched objects it will stop browsing the index,
- `all()` – return all objects that matched query,
- `first()` – return the first object that matched a query,
- `and(childQuery)` – logical and between two queries. Parent query resolves `childQuery` (call `all()` function) and add its results to `AND-hashset`.

- `and(property)` – Thanks to available generic programming[5] in Typescript, `and` can be also called with string argument. This register another query condition.
- `or(childQuery)` – logical or between two queries. `childQuery` will be resolved when parent query needs it, and its results stored in `OR-hashset` of parent query.
- `or(property)` – similar as `and` function, `or` can be also called with string parameter. This register another condition to query.
- `iterate()` – returns iterator that can be used in `for (result of query)` cycle.

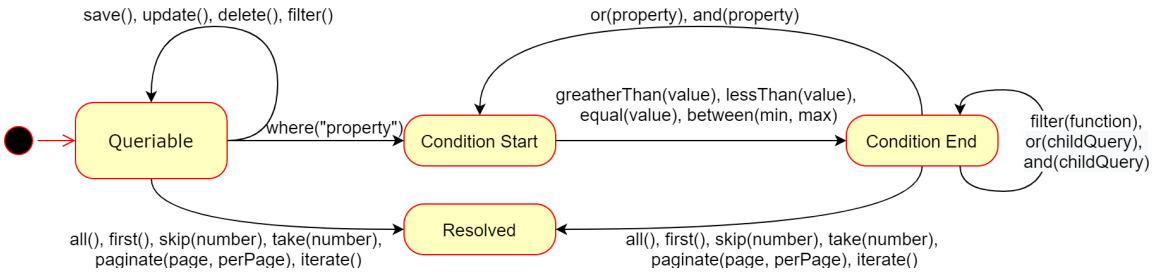


Figure 5.4: State diagram of the query system

5.1.4 Database

The most important part of the Explorer-core module is the database system which connects the query system with indexes. When we make a query, it translates to the database, and data are obtained with the help of the indexes.

Tables

A database contains tables that consist of indexes. A table has an interface that provides operations to modify table data:

- `create` – creates history log for entity with first entity version. Then adds it to every table index.
- `update` – adds new version of the entity to its history log. Then update every index of the table.
- `delete` – removes entity from every table index. From now, an entity can not be found in this table.

There is no `select` operation because a table itself does not perform queries for obtaining data (table has not got necessary logic for choosing optimal index for query). Selects are performed by QueryPlanner after analysing all query conditions.

Example of the class User is in Figure 5.5. It has two indexes. One primary on property `name` and one normal index on property `age`. The second index has also specified `comparator` that is a bit faster than default one but works only on numbers, and `keyGetter` that returns a year when a user has born.

[5] https://en.wikipedia.org/wiki/Generic_programming

```
class User extends Queriable<User> {
    @PrimaryKey()
    name: string;

    @Index(
        (a, b) => a - b,
        u => new Date().getFullYear() - u.age,
    )
    age: number;
}
```

Figure 5.5: Example of database table abstraction

Transactions

A database can execute only a single transaction at the time to prevent data inconsistency. For that reason, we implemented a transaction queue where transactions are stored before executing in the order in which they came. Executing more transactions in a row is significantly more effective than executing them one by one. If a transactions queue has only one transaction, it waits 50ms for more transactions to come. Transactions in our system are represented by classes `Transaction` and `TransactionsBulk`. Both classes implements `ITrasnaction` interface with `run` function. Transactions bulk are treated the same way as a single transaction thanks to Composite pattern[6] (see Figure 5.6). After executing, each transaction is appended to the database log. After we execute all transactions in transactions queue (transaction queue is empty), we publish a new database version to all connected peers.

```
interface ITransaction{
    run();
}

public class Transaction implements ITransaction {
    operation: DbOperation;
    data: any;
    run() {
        ...
    }
}

public class TransactionsBulk implements ITransaction {
    public transactions: ITransaction[];
    public run() {
        for (const transaction of this.transactions)
            await transaction.run(database);
    }
}
```

Figure 5.6: Transactions use composite pattern. Body of function `run` of the `Transaction` class is omitted due to complexity.

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

Synchronization

There are two types of transactions. Those that changes the database state (`create`, `update`, `delete`) and those that don't (`select`). Every transaction that changes database state needs to be synchronized with other peers. We created Database log for that. It is an append-only log with a discrete-time. Every entry of a database log has these properties:

- **hash** – an IPFS hash of this entry,
- **payload** – payload is object that contains all entry data. Usually, there is a transaction (or transactions bulk if in this entry is published more transactions) and an IPFS hash of the database.
- **parent** – an IPFS hash to previous entry. We can re-create a whole database log from a head entry be accessing parent property until it is `null`.
- **clock** – A Lamport clock [7] with entry create time.

There can be only one valid database transaction at every point in time. There are several strategies to choose which transaction is globally accepted and which transactions need to rollbacks (described in design 4.1.3). Merging database logs is a common operation in our database system. Every time a peer publishes new database version, other peers need to merge this version with theirs. Example of merging database logs can be seen in Figure 5.7. In this example there are two peers (`Node A` and `Node B`). First three database version (`A1`, `B1`, `A2`) follows each other and there is no problem with them. But versions `A3` and `B2` are published in the same time. Node A and B do not know about this issue yet, so they both publish another version (`A4` and `B3`) on top of their head. After nodes get notified about each other versions, their logs are merged by database log class function `merge` which accepts another log as a parameter (see Figure 5.8). At first, we create a new instance of `TransactionsBulk` where we store transactions that we may need to rollback. Then we check if both logs heads are published at the same time. If one head clock time is greater than another head clock time, we need to get previous database log entry from this log with the same time as another log head to be able to compare these versions. Luckily, both logs head from our example (`A4`, `B3`) are published at the same time. We can get a previous entry from accessing `parent` property of database log entry. Finally, after we got both logs heads from the same time, we can start traversing them and comparing their parents. If logs have the same parent (in our example it is version `A2`), we found a database version where the fork began. Now, peers need to decide which version of the database to accept. The simplest method is to compare hashes of database log entries. In our example `A3` entry has bigger hash than `B2`. This means that `Node B` needs to rollback transactions published in versions `B2` and `B3` and apply them after `A4`. After the merge, database log has a single head again and all published transactions have been applied.

5.2 Feeder

A Feeder is a simple command-line application written in Typescript (see Algorithm 1). It strongly depends on the Explorer-core module. Currently, we support only Blockbook connector as a source of blockchain's data. However, Feeder can be simply expanded to support more data source such as InsightAPI or direct connection to a blockchain as a

[7] https://en.wikipedia.org/wiki/Lamport_timestamps

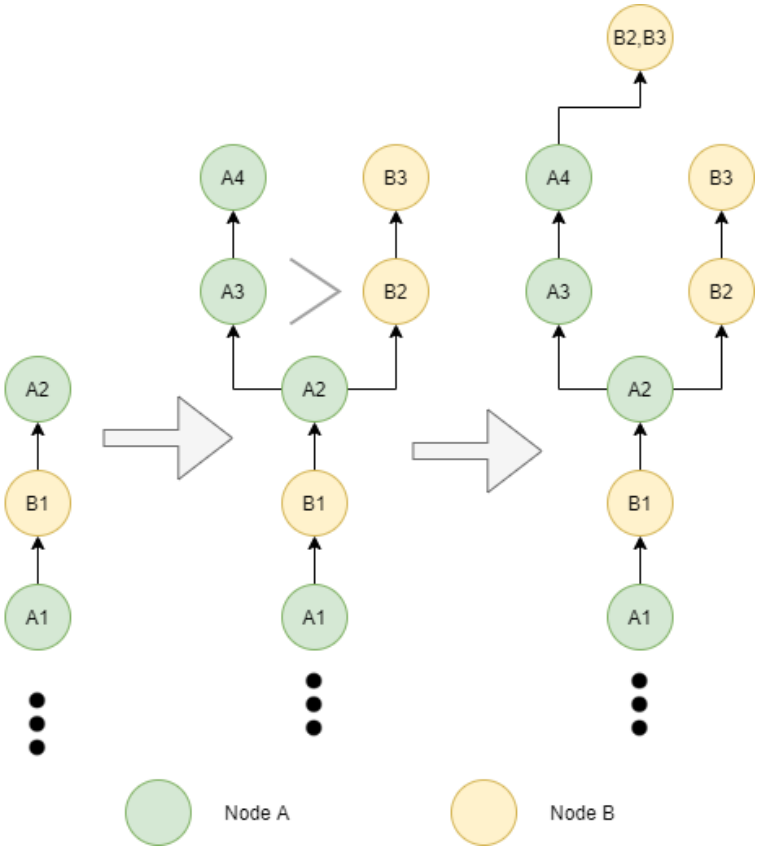


Figure 5.7: Database synchronization

full node. Each Feeder has configuration file (usually called `.env`) with Feeder's settings. Main Feeder settings are `URL` of the source for the blockchain's data and `DB_NAME` which is a name of the database where Feeder inserts new blocks. A Feeder can be connected to only one blockchain. We can create multiple Feeders for single blockchain, but we need to provide some deterministic algorithm, that ensures that each block is parsed by only one Feeder. This can be done by providing `FeederId` and `FeedersCount` in `.env` config file. If those values are provided, Feeder parses only blocks that have height modulo `FeedersCount` equals to `FeederId`.

Algorithm 1: Simplified Feeder algorithm

```
load configuration;
while there is new block do
    fetch block;
    for transaction in block do
        save transaction;
    end
    save block;
end
```

```
public async merge(log) {
    let thisHead = this.head;
    let otherHead = log.head;
    const rollbackOperations = new TransactionsBulk();
    while (thisHead.clock.time != otherHead.clock.time) {
        if (thisHead.clock.time > otherHead.clock.time) {
            if (thisHead.identity.isMe())
                rollbackOperations.push(thisHead.transaction);
            thisHead = await this.get(thisHead.parent);
        } else {
            otherHead = await log.get(otherHead.parent);
        }
    }
    while (thisHead.payload.parent != otherHead.parent) {
        if (thisHead.identity.isMe())
            rollbackOperations.push(thisHead.payload.transaction);
        otherHead = await log.get(otherHead.parent);
        thisHead = await this.get(thisHead.parent);
    }
    if (otherHead.compare(thisHead.hash)) {
        await this.migrate(log, rollbackOperations);
    }
}
```

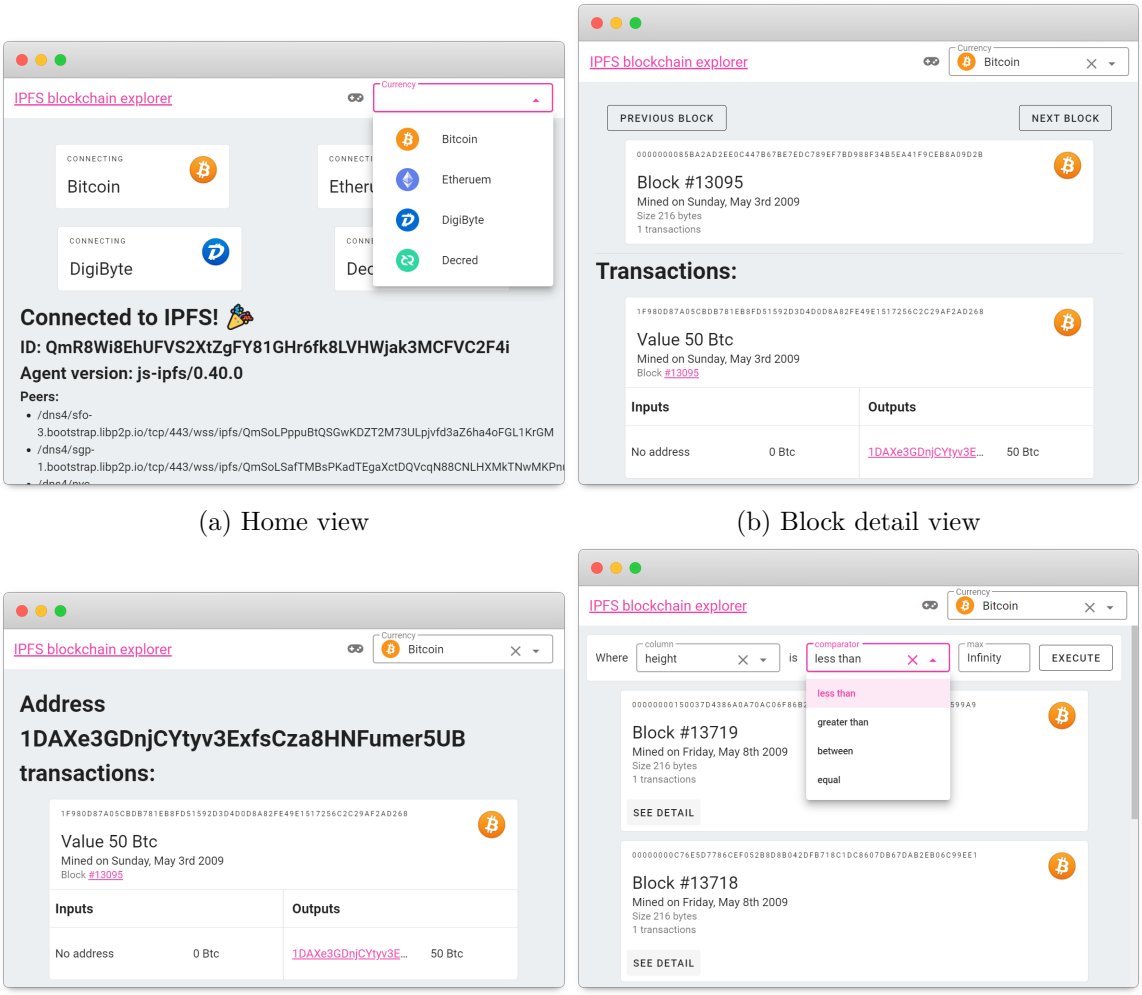
Figure 5.8: [[Zapisat algoritmom. Nie kodom]]Algorithm for merging database logs

5.2.1 ExplorerGUI

ExplorerGUI is a single page application with a simple user interface implemented with Vue.js[8] that runs in a browser. We use Vuetify[9] as a user interface library and browser implementation of IndexedDB[10] as a storage for IPFS. Communication with other peers is provided though WebRTC[11] and WebSockets because a web page in a browser can not open TCP socket. Every opened ExplorerGUI browser tab is the same IPFS node instance. Opening a new tab in incognito mode or different browser spawns new IPFS node.

After page with ExplorerGUI is loaded, it tries to connect to all supported blockchains. The home screen contains all enabled cryptocurrencies with their statuses (see Figure 5.9a). Blocks list view (Figure 5.9d) is displayed when a user selects specific cryptocurrency. A user can filter blocks by all blocks indexes in this view. A default filter gets all blocks that height is less than infinity. This query sorts blocks from the newest (highest) to the oldest (see Figure 5.9d). When a user clicks on the Execute button, ExplorerGUI starts loading blocks that match the selected query by the asynchronous algorithm shown in Figure 5.10. First, we use `iterate` function (implemented in Explorer-core module) to obtain iterator over subtree of all results. This function is asynchronous, so it does not block the main JavaScript thread while it is searching the whole index and looking for the subtree. Then we create an array of promises[12] in which the individual blocks that are displayed on the page are loaded. In for cycle, we push asynchronous tasks to the array. A task contains single parameter that is `pagePosition`. This parameter is the order of the block that tasks

[8] https://vuejs.org/
[9] https://vuetifyjs.com/
[10] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
[11] https://webrtc.org/
[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise



(a) Home view

(b) Block detail view

(c) Address view

(d) Blocks view

Figure 5.9: ExplorerGUI views

loads. Every task for each block then runs in parallel. First, task call `next` function on subtree iterator. This function returns the next block in the subtree. `next` function can return left or right sibling of the current block. It depends on the condition. A subtree is traversed to the left if `greaterThan` or `between` condition is used and to the right if there is `lessThan` condition. `next` function returns a pair of `blockPromise` which is a task that loads blockchain block from IPFS and `done` that is boolean, which signals if there are any more results in the subtree. Finally, task waits until blocks are loaded, and then assigns block in the right index of the blocks array. Thanks to this optimized asynchronous algorithm, all blocks that are displayed on the page are loaded in parallel.

When a user clicks on a block from the Block view, he is redirected to the Block detail view (see Figure 5.9b). All transactions that are confirmed in this block are listed here. Transactions have inputs and output that are usually cryptocurrency address (input can also be a coin base). Every address in a transaction in link to the Address View (see Figure 5.9c). The Address view contains all transactions that address is part of as input or output.

The last view that we implemented in ExplorerGUI is Playground (see Figure 5.11). In this view, the user can write a query to the web editor and execute it. Syntax highlighting

```
const subtree = await query.iterate()
const tasks = [];
for (let i = 0; i < pageSize; i++) {
    tasks.push(
        (async pagePosition => {
            const { blockPromise, done } = await subtree.iterator.next();
            if (done)
                return;
            blocks[page - 1 * pageSize + pagePosition] = await blockPromise;
        })(i),
    );
}
await Promise.all(tasks);
```

Figure 5.10: Loading blocks from database

in editor is done by Prism[13]. Result of a user query is returned in JSON format bellow query.

5.2.2 ExplorerAPI

ExplorerAPI is a server-side application that provides simple HTTP API. After ExplorerAPI starts, it connects to the all enabled cryptocurrency databases (specified in `.env` file). Then it registers all three controllers (for blocks, transactions and addresses) and starts HTTP server on port described in `.env` file or `5000` if there is no specific port in the environment file. ExplorerAPI has only three endpoints (`block`, `address` and `transaction`). All these endpoints accepts query parameter (in case of `GET` HTTP method) or body parameter (`POST` HTTP method) in which query is described (see Figure 4.7a). This parameter can consist of array of conditions and filters, skip property and resolver. The parameter is translated to the Query system and result is return in JSON format (see Figure 4.7b). ExplorerAPI can accept multiple connections at once, because each request is processed in separated promise (task).

[13] https://prismjs.com/

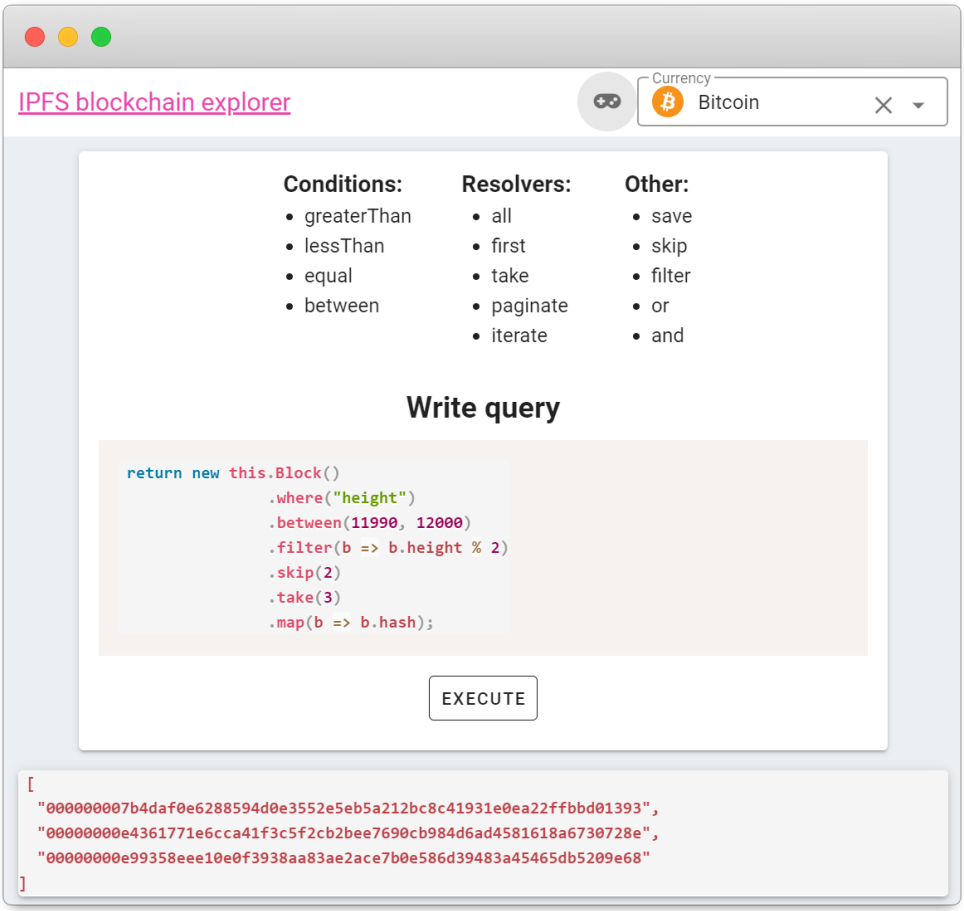


Figure 5.11: Playground view

Chapter 6

Testing

Testing is an essential part of the development of any project. As part of my master thesis, I performed two types of testing. Unit testing to ensure that every system module works correctly, and testing the user interface of ExplorerGUI in which I tested how quickly and efficiently the user handles tasks, and how user-friendly the user interface is.

6.1 Unit testing

We tested all critical parts of the Explorer-core module through unit tests. We use jest.js[1] framework for writing tests. We test every operation on B-Tree index (add, remove, find, update), an append-only log that is used for database log and record log (add, merge, difference).

6.2 Testing of user interface

We also test ExplorerGUI by a user. We invited for this type of testing five people whose task was to perform several simple steps:

1. Load ExplorerGUI web page.
2. Select cryptocurrency.
3. Make filter that selects all blocks between 1000 and 2000.
4. Click on any block to view its transactions.
5. Select some address and view its transactions.

There were almost no problems during testing. As this is a relatively complex system that requires at least a minimum knowledge of cryptocurrencies, the principles of blocks and transactions were not clear to all tested. However, testers completed the tasks in a short time. The user interface seemed to everyone intuitive and straightforward. Subsequently, given the test results, we performed a few graphical changes, such as enlarging and highlighting some controls. Also, most testers liked the graphic design and the chosen colours of the application.

[1] https://jestjs.io/

Chapter 7

Benchmarking

7.1 Comparing to other database systems

Comparing our database system with typical database systems is very difficult. We need to prepare the same environment.

7.1.1 MySQL

MySQL is free and open-source software under the terms of the GNU General Public Licence and is also available under a variety of proprietary licences. MySQL was owned and sponsored by the Swedish company MySQL AB, which was bought by Sun Microsystems (now Oracle Corporation). MySQL is used by many database-driven web applications, including Drupal, Joomla, phpBB, and WordPress. MySQL is also used by many popular websites, including Facebook, Flickr, MediaWiki, Twitter, and YouTube.

7.1.2 PostgreSQL

PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance. It was initially named POSTGRES, referring to its origins as a successor to the Ingres database developed at the University of California, Berkeley. In 1996, the project was renamed to PostgreSQL to reflect its support for SQL. After a review in 2007, the development team decided to keep the name PostgreSQL. PostgreSQL features transactions with Atomicity, Consistency, Isolation, Durability (ACID) properties, automatically updatable views, materialized views, triggers, foreign keys, and stored procedures. It is designed to handle a range of workloads, from single machines to data warehouses or Web services with many concurrent users. It is available for Linux, FreeBSD, OpenBSD, and Windows.

7.1.3 Environment

We compare MySQL and PostgresSQL with our p2p database system. These database systems run virtualized in Docker containers on hardware with 4 cores processor Intel Pentium G4560 3.50GHz and 16 GB of RAM. We start clients that connect to these database systems and performs queries on different hardware. Then we measure average queries per seconds that clients execute. Clients are written in JavaScript with official node.js

connectors for MySQL[1] and PostgreSQL[2] and are connected in the LAN with database systems over 1Gb Ethernet. Database structure that is used in benchmarks can be seen in Figure 7.1. Table `authors` contains 10000 records and table `posts` contains 50000 records (each author has 5 posts). Test query for each database system can be seen in Figure 7.2.

```
CREATE TABLE `authors` (
    `id` int(11) PRIMARY KEY,
    `first_name` varchar(50),
    `last_name` varchar(50),
    `email` varchar(100),
    `birthdate` date,
    `added` timestamp,
    PRIMARY KEY (`id`),
    UNIQUE KEY `email` (`email`)
);

CREATE TABLE `posts` (
    `id` int(11) PRIMARY KEY,
    `author_id` int(11),
    `title` varchar(255),
    `description` varchar(500),
    `content` text,
    `date` date,
    PRIMARY KEY (`id`)
);
```

(a) MySQL

```
CREATE TABLE authors
    id serial PRIMARY KE
    first_name varchar(50),
    last_name varchar(50),
    email varchar(100),
    birthdate date,
    added timestamp
);

CREATE TABLE posts (
    id serial PRIMARY KE
    author_id integer,
    title varchar(255),
    description varchar(500)
    content text,
    date date
);
```

(b) PostgreSQL

```
class Author {
    @PrimaryKey() id: number;
    first_name: string;
    last_name: string;
    email: string;
    @Index() birthdate: Date;
    added: Date;
}

class Post {
    @PrimaryKey() id: number;
    author_id: number;
    title: string;
    description: string;
    content: string;
    date: Date;
}
```

(c) Our database system

Figure 7.1: Database structures for benchmarking

```
SELECT * FROM authors
JOIN posts ON
posts.author_id=authors.id
WHERE authors.id=
  (SELECT
    FLOOR(RAND() * 10001 + 1));
```

(a) MySQL

```
SELECT * FROM authors
JOIN posts ON
posts.author_id=authors.id
WHERE authors.id=
  (SELECT
    floor(random() * 10001 +
```

(b) PostgreSQL

```
new Author()
.where("id")
.equal(
    Math.floor(
        Math.random() * 10001) + 1)
.first();
```

(c) Our database system

Figure 7.2: Test query

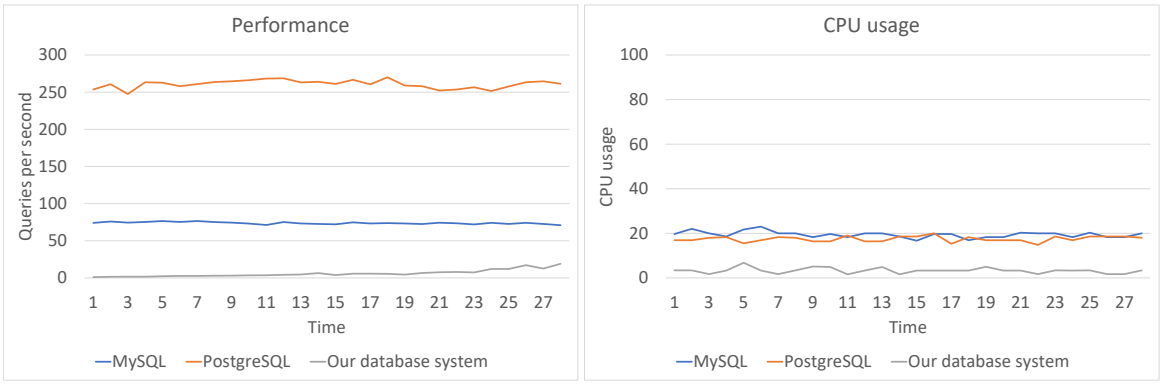
Single connected client

With a single connected client, PostgreSQL can make more than 250 queries per second and MySQL 70 queries per second. Our database system is far behind with up to 20 queries per second (see Figure 7.3a). We can see that our database system builds cache, and performance increases by time. With only a single connected client, there is no advantage of a p2p network. CPU usage with a single connected client is for MySQL and PostgreSQL same – about 20% (see Figure 7.3b). For our database system, it is only 3% due to lots of input-output operations.

20 connected clients

With 20 connected clients, PostgreSQL performance dropped to 50 queries per seconds and MySQL to 10 queries per seconds. At the end of benchmarking, our database system is more powerful than PostgreSQL and MySQL, because data gets distributed to clients (see 7.4a). Our database system also requires less processor time thanks to p2p network. Data are not obtained only from the server (as opposed to MySQL and PostgreSQL) but also

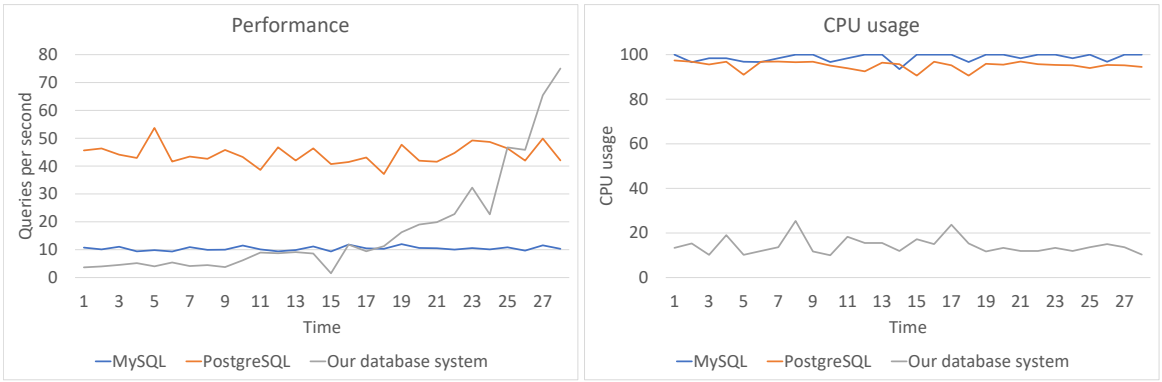
[1] https://www.npmjs.com/package/mysql
[2] https://www.npmjs.com/package/postgres



(a) Performance (b) CPU usage with

Figure 7.3: Single connected client benchmark plots

from other clients. Thanks to this, our system only using less than 20% CPU power (see Figure 7.4b).



(a) Performance (b) CPU usage with

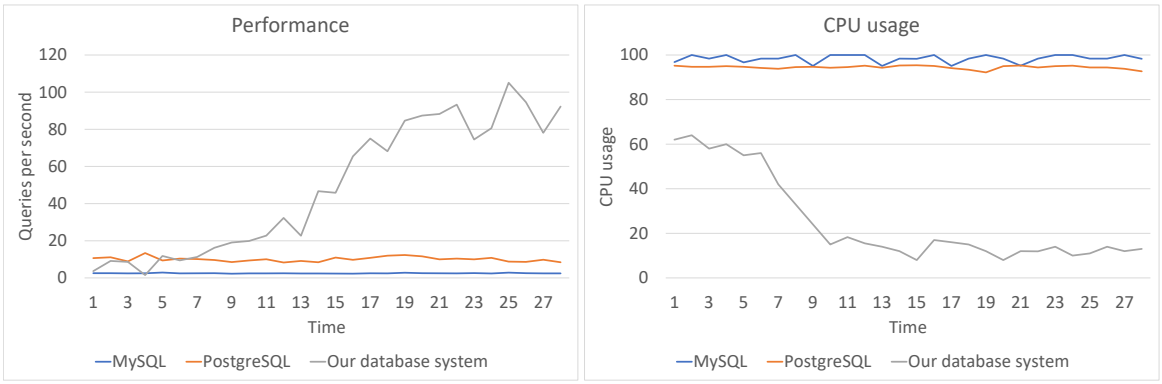
Figure 7.4: 20 connected clients benchmark plots

80 connected clients

With 80 connected clients, our database system outperforms MySQL and PostgreSQL after few seconds. Every client of our database system performs 100 queries per second. PostgreSQL client performs only 10 queries per second and MySQL only 2 queries per second (see 7.5a). CPU usage is in our database system also very low. In the beginning, when content is not distributed on the clients, the central node uses 60% of CPU, but it decreases over time (see 7.5b).

150 connected clients

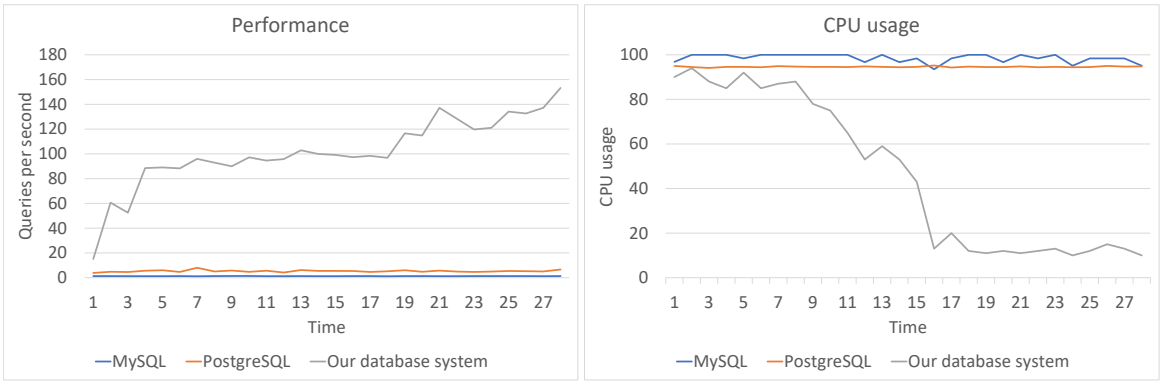
We need to change the default PostgreSQL setting for maximum connections to connects more than 100 clients to the PostgreSQL. Our system can handle 150 clients very well (opposite to MySQL and PostgreSQL). Each client connected to our database system can



(a) Performance (b) CPU usage with

Figure 7.5: 80 connected clients benchmark plots

make up to 150 queries per second, and it increases by the time (see 7.6a). For the first seconds, CPU usage on the central node is high (near 100%), but it decreases over time (see 7.6b).



(a) Performance (b) CPU usage with

Figure 7.6: 150 connected clients benchmark plots

7.1.4 Benchmark conclusion

We can see that with more than 20 connected clients, our database system became more effective than traditional ones. Thanks to IPFS and its p2p network, a content of the database is distributed on the clients. This reduces the load on the central node.

7.2 Comparing with blockchain explorers

Our blockchain exploration system solution is unique and differs from traditional blockchain explorers in many ways. See the table 7.1 to compare traditional blockchain explorers with ours.

	Our system	**Traditional blockchain explorer (Blockbook, Insight)**
Database system	Proprietary	Typically NoSQL database
Views for blocks, transactions and addresses	Yes	Yes
Complex queries support	**Yes**	No
Filtering	**Yes**	No, but filtering can be made by cycling over results
Sorting	**Yes**	No
Blockchain synchronization	Typically slower but it depends on how many Feeders are working on blockchain	**Faster**
Get block/transaction/address by hash	Slower	**Faster, but may slow down if multiple nodes are connected**
When more user is connected	**Works faster**	Works slower
Multiple coins support	Yes	Yes

Table 7.1: Comparing our system with typical blockchain explorers

Chapter 8

Conclusion

[[**cele prepisat**]] The goal of this project was to create a system that uses IPFS to explore blockchain. To achieve this, we needed to create our own decentralized and distributed database system that supports queries and indexes on top of the IPFS. Feeder and ExplorerCore communicate with this database system to store (Feeder) or retrieve data (ExplorerCore). To visualize data in GUI, ExplorerGUI can be used, and for obtaining data with RestAPI there is ExplorerAPI.

The research section of this project focuses mainly on the IPFS principles (content-based addressing, nodes linking, and others). Selected cryptocurrencies for exploring are also briefly discussed. Considerable effort was devoted to designing the whole system and creating a functional prototype.

In the next semester, we want to highly improve database system by support more operators (`like`, `groupBy`). We want to research and compare more indexing strategies and make performance testing and profiling. Also, comparison with Blockbook in the same conditions would be interesting. There is high potential by using IPFS in this system to create new use cases that no other system for exploring cryptocurrencies blockchain supports. For example, with some heuristic, we believe, that with this system, we can follow crypto coins after exchange for another crypto coins (thanks to linking between objects in the IPLD layer).

Bibliography

[1] Benet, J. IPFS - Content Addressed, Versioned, P2P File System. *CoRR*. 2014, abs/1407.3561. Available at: http://arxiv.org/abs/1407.3561.

[2] Buterin, V. et al. A next-generation smart contract and decentralized application platform. *White paper*. 2014, vol. 3, no. 37.

[3] Chacon, S. and Straub, B. *Pro git*. Apress, 2014.

[4] Chowdhury, N. *Inside blockchain, bitcoin, and cryptocurrencies*. Boca Raton: CRC Press, 2020. ISBN 978-1-138-61815-2.

[5] Dascano, M. *Digibyte: An Easy Guide to Learning the Essentials*. 1stth ed. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018. ISBN 1725629178.

[6] Davis, J. The Crypto-Currency. *The New Yorker*. october 2011. Available at: https://www.newyorker.com/magazine/2011/10/10/the-crypto-currency.

[7] Dhillon, V., Metcalf, D. and Hooper, M. *Blockchain Enabled Applications: Understand the Blockchain Ecosystem and How to Make it Work for You*. Springer, 2017.

[8] Dias, D. and Benet, J. Distributed Web Applications with IPFS, Tutorial. In: Bozzon, A., Cudre Maroux, P. and Pautasso, C., ed. *Web Engineering*. Cham: Springer International Publishing, 2016, p. 616–619. ISBN 978-3-319-38791-8.

[9] Haber, S. and Stornetta, W. S. How to time-stamp a digital document. In: Springer. *Conference on the Theory and Application of Cryptography*. 1990, p. 437–455.

[10] Jepson, C. DTB001: Decred Technical Brief. *Available at h ps://coss. io/documents/white-papers/decred. pdf Additional information available at h ps://www. decred. org*. 2015.

[11] Kalle, K. *Big data in video games*. Lappeenranta, FI, 2017. Bachalar Thesis. Lappeenranta University of Technology, School of Business and Management, Computer Science. Available at: http://lutpub.lut.fi/handle/10024/147666.

[12] Labs, P. Filecoin - A Decentralized Storage Network. 2017. Available at: https://filecoin.io/filecoin.pdf.

[13] Lamport, L. Time, clocks, and the ordering of events in a distributed system. In: *Concurrency: the Works of Leslie Lamport*. 2019, p. 179–196.

[14] Laurence, T. *Blockchain for dummies*. John Wiley & Sons, 2019.

[15] Mark Robert Henderson, S. P. *The OrbitDB Field Manual*. 2019. Available at: https://github.com/orbitdb/field-manual.

[16] Maymounkov, P. and Mazieres, D. Kademlia: A peer-to-peer information system based on the xor metric. In: Springer. *International Workshop on Peer-to-Peer Systems*. 2002, p. 53–65.

[17] Nakamoto, S. and Bitcoin, A. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*. 2008.

[18] Narayanan, A., Bonneau, J., Felten, E., Miller, A. and Goldfeder, S. *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press, 2016.

[19] Noether, S. Ring SIgnature Confidential Transactions for Monero. *IACR Cryptology ePrint Archive*. 2015, vol. 2015, p. 1098.

[20] Pick, S. and Hagopian. *A protocol and event-sourced database for decentralized user-siloed data*. 2019. Available at: https://blog.textile.io/introducing-textiles-threads-protocol/.

[21] Van Saberhagen, N. *CryptoNote v 2.0*. 2013.

[22] Wayner, P. *Digital cash: Commerce on the net*. Academic Press Professional, Inc., 1997.

[23] Wood, G. et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*. 2014, vol. 151, no. 2014, p. 1–32.