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

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 REST 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.

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 1, 2020

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[7]. Linux 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 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 Semestral project 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 7.

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

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

[3] 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.[11]

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 originally used for timestamping digital documents.[5]

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[9]

2.1 Bitcoin

Bitcoin is probably the most famous cryptocurrency. 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 gets 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][3].

```
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÷º..W
00000110   8A 4C 70 2B 6B F1 1D 5F  AC 00 00 00 00           ŠLp+kñ._¬....
```

Figure 2.2: Genesis bitcoin block

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 transactions hash is `a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d` and will be 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 bigger 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 570 bytes of data. One block can contain approximately 3 500 transactions given the 2 MB limit. This means that in DigiByte, 230 transactions can be confirmed in one second compared

[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/

to Bitcoin 4-7 transaction per second. DigiByte also has 1000:1 DigiByte to Bitcoin ratio, so for every Bitcoin there will be 1000 DigiByte.[2]

2.3 Ethereum

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).[12]

2.4 Decred

Decred is cryptocurrency build from Bitcoin. 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% hashpower attack very difficult, and making a minority fork very difficult as well.[6]

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 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. Monero hides recipient and sender addresses.[10]

2.6 Analyze of current blockchain explorers

[[vela z nich je NoSql. Ziadne filtrovanie a pod.]]

Chapter 3

IPFS

IPFS[1] stands for InterPlanetary File System and is a peer-to-peer distributed filesystem designed to make the Web faster, safer, and more open. In contrast with a standard filesystem, 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). In IPFS 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 license text, and add it to IPFS.If somebody tries to add this license 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 user try 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 is no known size limitations uploaded file or directory. There are already some big datasets hosted on IPFS such as Geocities archive[2] (704TB) or Project Apollo Archives[3] (61GB).

IPFS provides API for manipulation with objects in several levels of abstraction. Two most used APIs are Files and Graph API. The files API[4] enables users to use the File System abstraction of IPFS. Currently it has `add`, `cat`, `get` and `ls` methods. In the future, there should be MFS (Mutable File System), that would add methods `chmod`, `cp`, `mv`, `rm`, `stat` and `touch`[5]. Graph API[6] is used in lower level of abstraction for manipulating with any IPFS object and creating links between objects. There are `put` and `get` methods available.

[1] https://ipfs.io/
[2] https://ipfs.io/ipfs/QmVCjhoEFC9vwvaa8bKyJgwAByP4MXSogcyDGoz4Lkc3ox
[3] https://ipfs.io/ipfs/QmSnuWmxptJZdLJpKRarxBMS2Ju2oANVrgbr2xWbie9b2D
[4] https://github.com/ipfs/interface-js-ipfs-core/blob/master/SPEC/FILES.md
[5] https://docs.ipfs.io/guides/concepts/mfs/
[6] https://github.com/ipfs/interface-js-ipfs-core/blob/master/SPEC/DAG.md

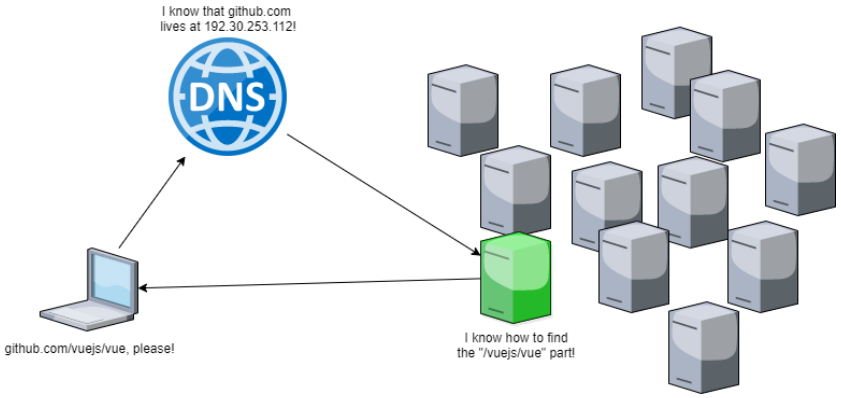


Figure 3.1: Classic web addresing

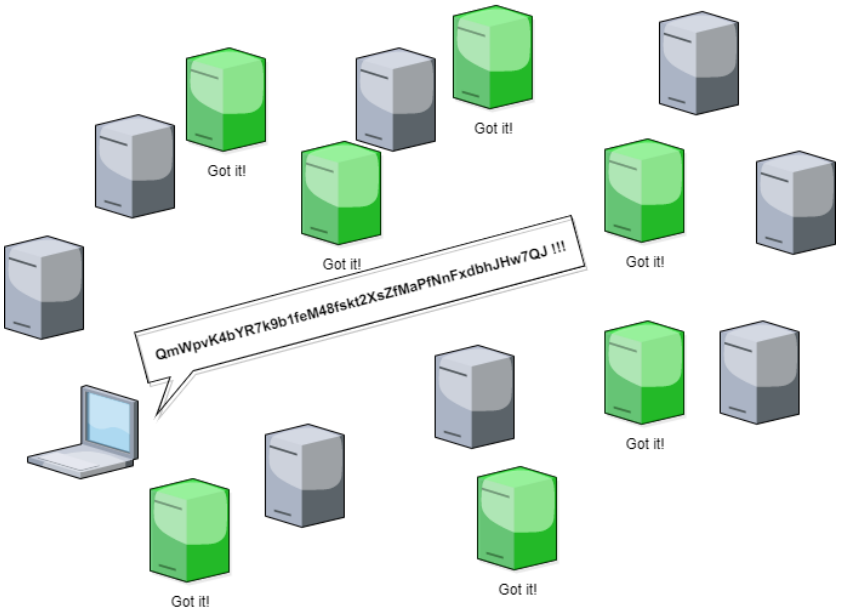


Figure 3.2: Content-based addressing

3.1 IPFS stack

We can split IPFS into layers (see Figure 3.3). *Libp2p*[7] 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 to publish and share files (or any data).[1]

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

[7] https://libp2p.io/

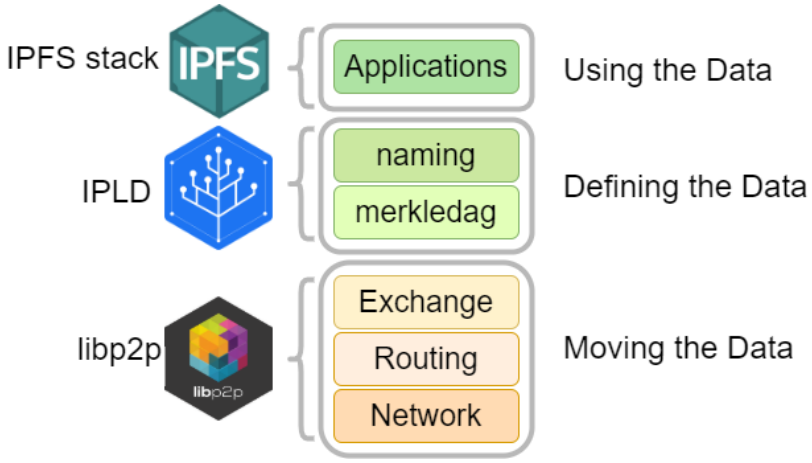


Figure 3.3: IPFS stack

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 human redable version of CID

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[8]. In the future, Kademlia might be changed easily to some other solutions that implements a simple interface for publishing and requesting data and finding a peer.[4]

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[8] format) which includes codec and multihash. Multihash is then further composed of hashtype and hash value. Let us look closer on the MIT license file, that we add to IPFS at the beginning of this chapter (see Figure 3). It's CID is `QmWpvK4bYR7k9b1feM48fskt2XsZfMaPfNnFxdbhJHw7QJ`. It can be converted to human-readable format as can be seen in Figure 3.1, thanks to multicodec table[9]. We can see that this CID is encoded in base58 format and the file was stored using protobuf[10] codec (this information is necessary to decode file correctly).

[8] https://en.wikipedia.org/wiki/Base58
[9] https://github.com/multiformats/multicodec/blob/master/table.csv
[10] https://en.wikipedia.org/wiki/Protocol_Buffers

3.4 IPFS

IPFS is the top layer from the IPFS stack. It is used for pinning objects and files, naming system (see IPNS[11]) 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 important and must not be thrown away. When garbage collection 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 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.

3.5 IPNS

Naming inside IPFS is governed by IPNS[12], the InterPlanetary 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 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. Using custom codecs for cryptocurrencies allows explorers to request blocks and transactions by its hash very fast, but there are also limitations.

- 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.

3.7 IPFS cluster

[11] https://docs.ipfs.io/guides/concepts/ipns/
[12] https://docs.ipfs.io/guides/concepts/ipns/
[13] https://en.wikipedia.org/wiki/Self-certifying_File_System
[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 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, but it is still in the alpha stage of developing, and it is not well optimized to store 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 selects 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.

4.1.2 Textile

Textile 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 provides 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 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.

4.1.3 Database system design

After some research, we concluded that currently for storing and indexing data in IPFS without large harddisk 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, 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). This approach is impossible in a decentralized system. When we update some record in our database, we can not know if somebody does not update it before (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. It two countries updates data at the same time (their updates pointing to the same previous version of the entry) they create conflict. Luckily, this specific conflict is easy to solve. We just needs to look at the previous entry and compute increment for both countries. Then update the record with final increment.

Indexes

An index is B-tree optimized for IPFS (no cycled references and node size less than 256kB). 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 keys of B-tree) and outputs a number. An index is using a comparator to search records and insert record to the right place in B-tree.
- **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 B-tree.

Table

A table contains indexes and table name. Also it implements operations like `insert`, `update` and `delete`.

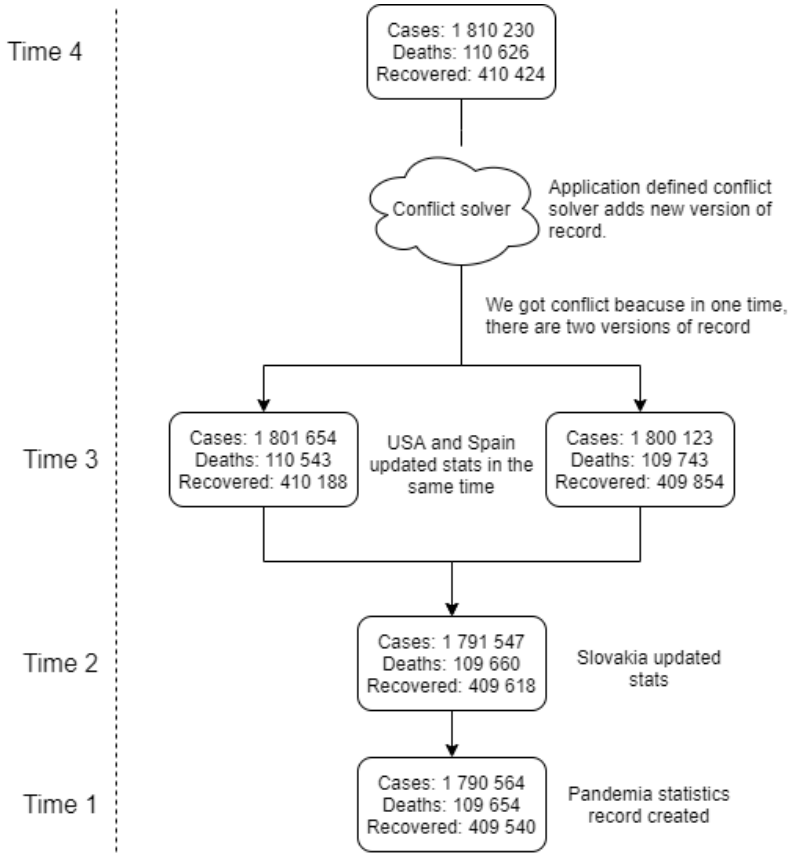


Figure 4.1: Record conflict

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 migration is done, transactions queue can continue working normally.
- `insert`, `update`, `delete` - other 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. When we receive information about the new database version, we load its root. Each database version has information from which version was created. Database versions create an append-only log called database log. Database log is an immutable, operation-based conflict-free replicated data structure (CRDT[1]) for storing database versions. Every version in the log is saved in IPFS and each points to a hash of previous version(s) forming a graph. Database logs can be forked and joined back together.[**?**]

If more than one peer publishes a new version of the database that has been created from the same database version, other peers needs to decide which version they would accept. Opposite to records conflicts, we needs 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 hash wins. This strategy is present in Figure 4.2.
- **The longest connecting time wins** - a user who 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).

Queries

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

- **conditions** - query may contains 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's conditions and union between or's a disjunction. These types of conditions are supported:
  - **equal** - record property equals to specified value,
  - **greather than** - record property is greather than specified value,
  - **less than** - record property is less than specified value,
  - **between** - record property is greather 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 returning boolean that are being applied to query results. If any filter return `false` for the query result, the query result is ignored. A query can contain several filters.
- **offset** - offset saying how many query results should be ignored from the beginning.
- **evaluator** - we use evaluator for accessing results of the query.

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

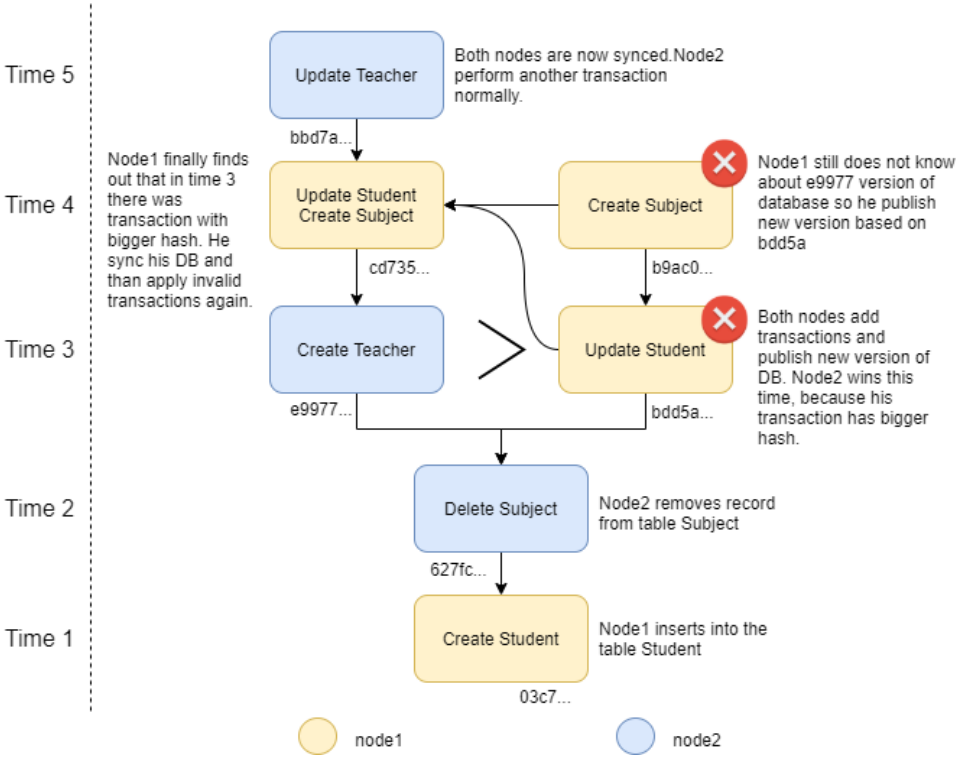


Figure 4.2: Synchronization of a database. Note that in time 4, Node1 add two transactions in one DB version.

- **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).

4.2.1 Database design

Both, Explorer and Feeder uses the same database schema. The database schema is designed in the way to optimalize 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 heigh very efficiently. Also filtering or ordering blocks by time has good performance.

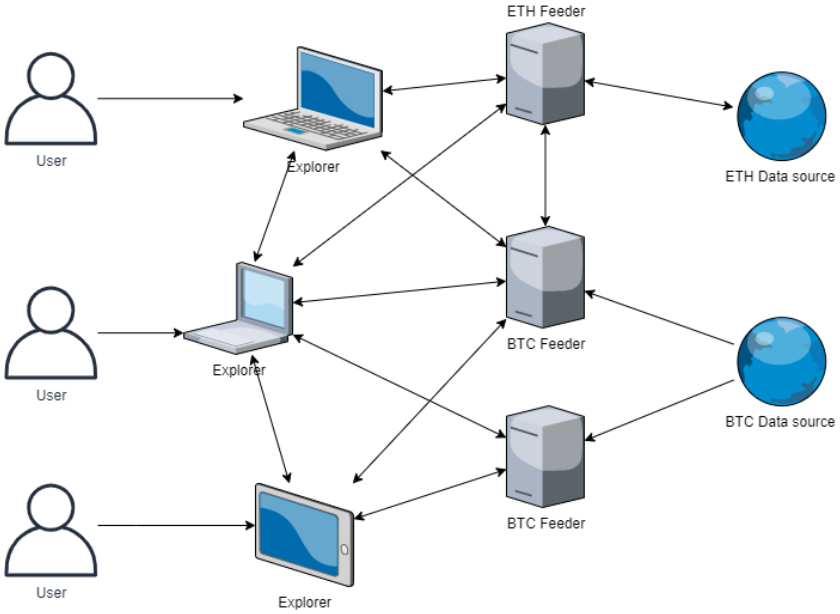


Figure 4.3: System architecture

- **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.
- **Vin** - every transactions can have multiple inputs. These inputs are stored in Vin table. Every Vin has not got any unique property, so a custom Guid[2] 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 an input can mined block reward.
- **Vout** - transactions can have multiple outputs. Every output has a foreign key to transaction and address. If an output is already spent, it has a link to the transaction where this output is used as an input.
- **Address** - Adress contains multiple columns. As primary key it uses address's 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 updated database version. This way new clients synchronize database more quickly. For optimization purposes, Feeder is performing transactions in bulks. This 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 consists of hundreds of transactions.

[2] https://en.wikipedia.org/wiki/Universally_unique_identifier

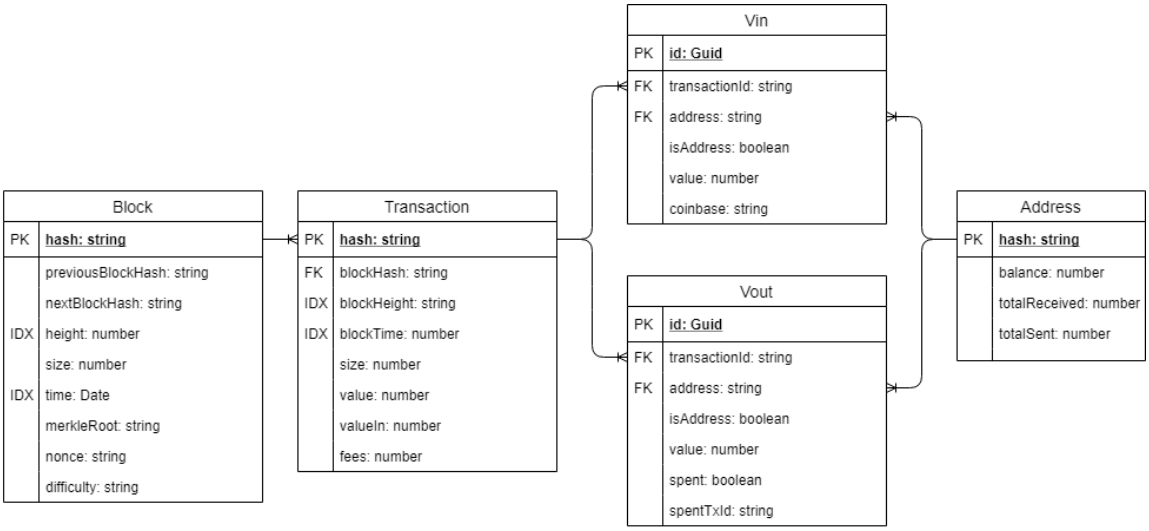


Figure 4.4: Entity-relationships diagram

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[3] or Insight[4].

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[5] as a RestAPI.

ExplorerGUI

ExplorerGUI is browser version of Explorer. It is implemented as a SPA[6] to prevent restarting connection with IPFS after each time a user visit a different page. It has separated views for block, transaction and address details. Also paginating, filtering and sorting objects by all its properties that has index is supported. Every user of ExplorerGUI keeps part of the database that he use 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 redirect a user to Block detail view

[3] https://github.com/trezor/blockbook
[4] https://insight.is
[5] https://nodejs.org/
[6] https://en.wikipedia.org/wiki/Single-page_application

where he can see all blocks transactions (see Figure 4.5c). 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 transaction which this address was part of (see Figure 4.5d). Thanks to the database design, a user can also see the history of the address.

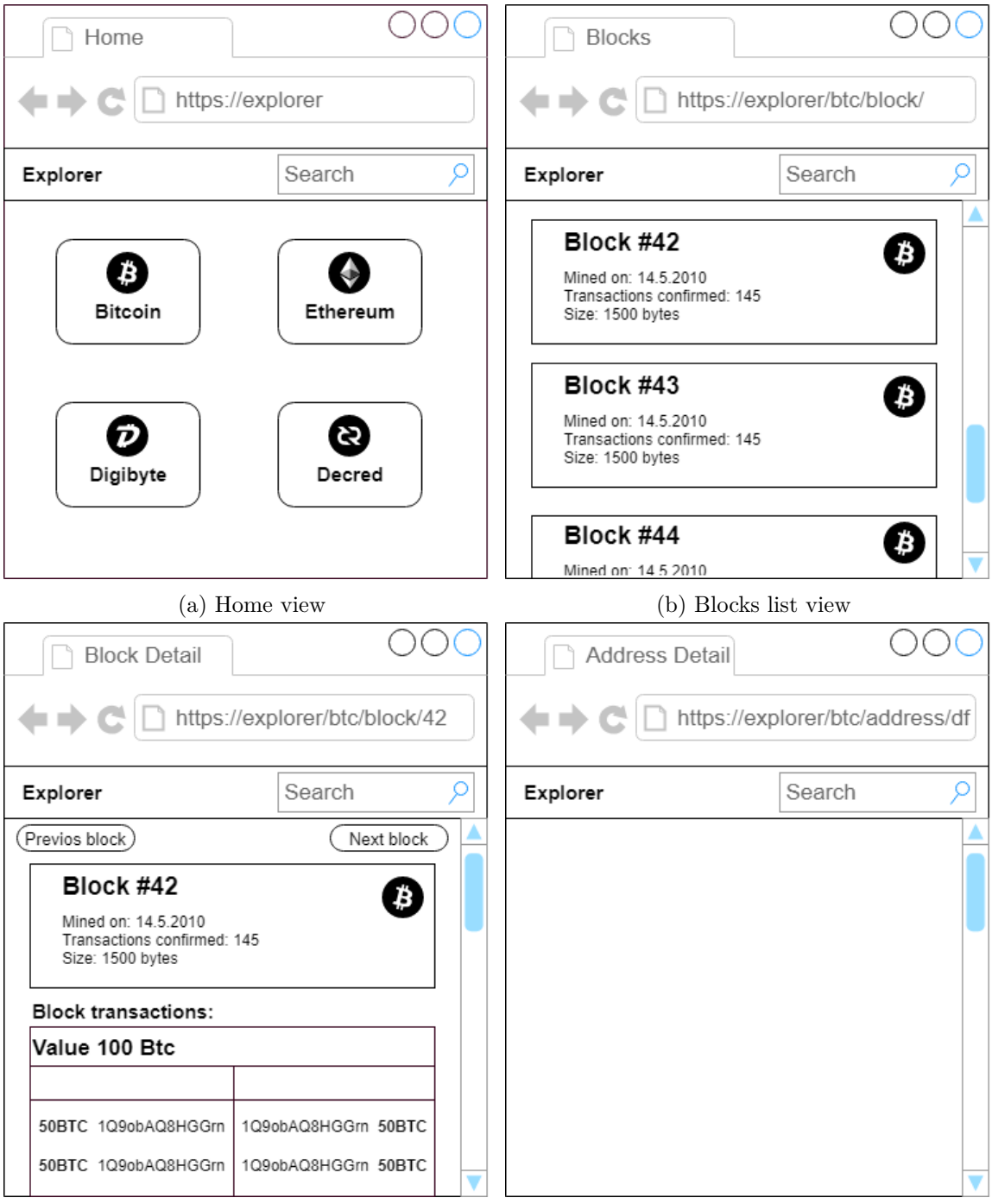
ExplorerAPI

ExplorerAPI is a server-side application that provides simple RESTapi for obtaining data. It is much less demanding on performance than ExplorerGUI and can be used for integrations with other applications. It has several endpoints:

- `GET /currency/[:currencyUnit]/tx/[:txHash]` - get transaction by hash,
- `GET /currency/[:currencyUnit]/block` - get block collection,
- `GET /currency/[:currencyUnit]/block/[:blockHeightOrHash]` - get block by height or hash,
- `GET /currency/[:currencyUnit]/address` - get address collection,
- `GET /currency/[:currencyUnit]/address/[:hash]` - get address by hash.

Every endpoint supports query parameters:

- `filter` where value of this parameter can be simple arithmetic expression like `height>50`,
- `limit` parameter for pagination with default value set to `20`,
- `orderBy` sorts results of a sequence in ascending order.
- **[[Paginate]]**



(a) Home view

(b) Blocks list view

(c) Block detail view

(d) Address view

Figure 4.5: ExplorerGUI mockups

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 choose 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).

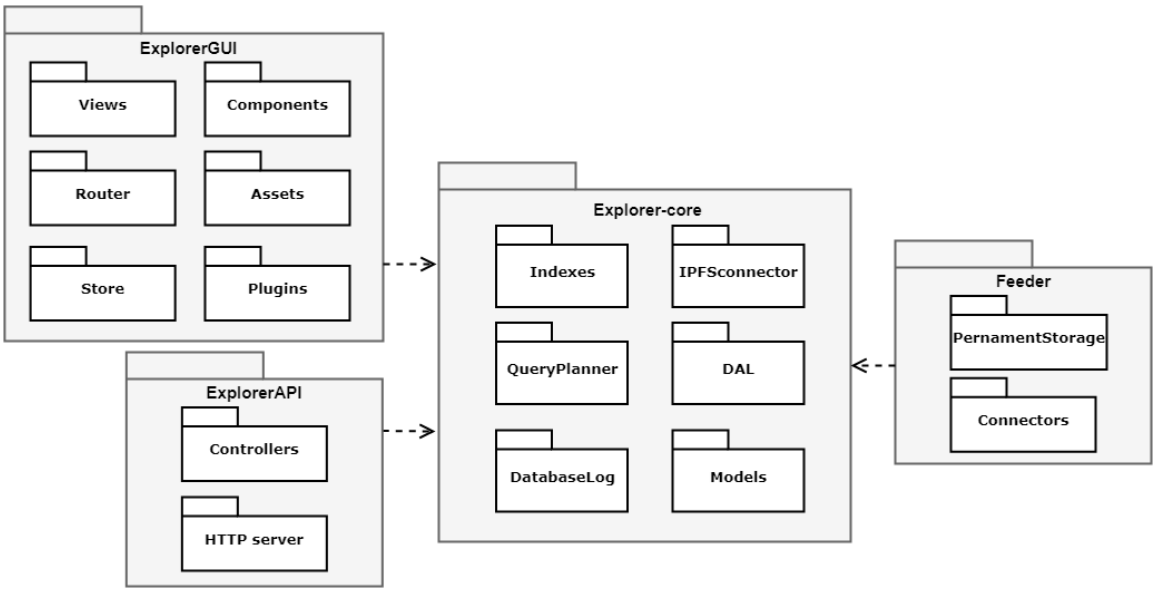


Figure 5.1: System architecture

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 a database abstract 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 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's 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 second, zero if arguments are the same, more then 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 property of the entity (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 this is not possible in IPFS. Cyclic references are not possible, 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` has now link 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 lead to better performance when executing queries. In insert queries, statistically fewer nodes needs 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 direction. In-order for `greater than` and `between`. For `less than` reversed in-order. With `equal` it does not matter.

[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

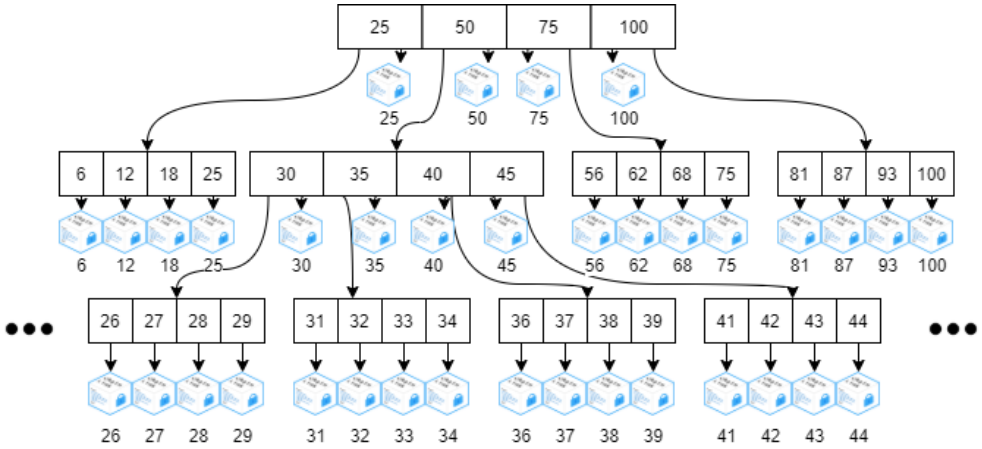


Figure 5.2: Btree indexing first 100 blocks by their height

5.1.2 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 peer's filesystem and are offered to other peers.

We can make queries with every database table model (such as model on Figure 5.4). When object inherits from `Queriable` class, it gets access to these query methods:

- `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

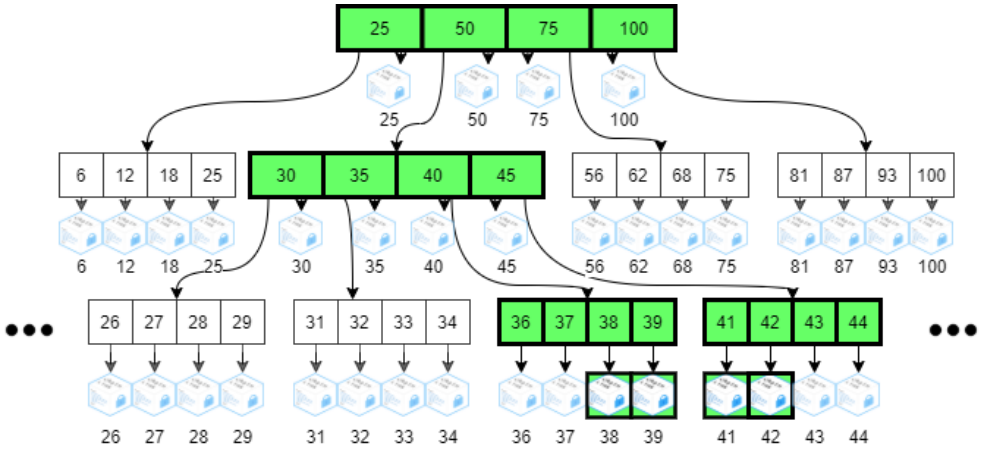


Figure 5.3: Data access for performing query that returns blocks that have a height between 38 and 42.

`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`.
- `or(childQuery)` - logical or between two queries. `childQuery` will be resolved by parent query, and its results stored in `OR-hashset` of parent query.
- `[Symbol.iterator]()` - iterator that is used in cycle `for (result of query)`.

5.1.3 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 a help of the indexes.

Tables

A database contains tables that consists of indexes. A table has interface that provide basic CRUD operations on its 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. Than update every index of the table.
- `delete` - removes entity from every table index. From now, an entity can not be find in this table.
- `select` - obtain data. If no condition is set, table uses primary key index.

Example of the class User is in Figure 5.4. 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.

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

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

Figure 5.4: Example of database table abstraction

Transactions

A database can be executing 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 will wait 100ms for more transactions to come.

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 need to be synchronized with other peers. We created Database log for that. It is an append-only log with a discrete-time. There can be only one valid database transaction in 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). **[[popisat algoritmus synchronizacie]]**

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, but Feeder can be simply expanded to support more data source such as InsightAPI or direct connection to a blockchain as a 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 insures 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
```

5.2.1 ExplorerGUI

ExplorerGUI is a single page application with a simple user interface implemented with Vue.js[5] that runs in a browser. We use browser implementation of IndexedDB[6] as a storage for IPFS. Communication with other peers is provided though WebRTC[7] and WebSockets because a webpage 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 will spawn new IPFS node.

After page with ExplorerGUI is loaded, it tries to connect to all supported blockchains. The home screen contains all enabled cryptocurrencies. Blocks list view is displayed when a user selects specific cryptocurrency. A user can filter blocks by all blocks indexes in this view. A default filter get all blocks that height is less than infinity. This query sorts blocks from the newest (highest) to the oldest (see Figure 5.5). When a user clicks on the Execute button, ExplorerGUI starts loading blocks that match the selected query by the asynchronous algorithm shown on Figure 5.6. First, we use `iterate` function (implemented in ExplorerCore 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 whole index and looking for the subtree. Then we create an array of promises[8] 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

[5] https://vuejs.org/
[6] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
[7] https://webrtc.org/
[8] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

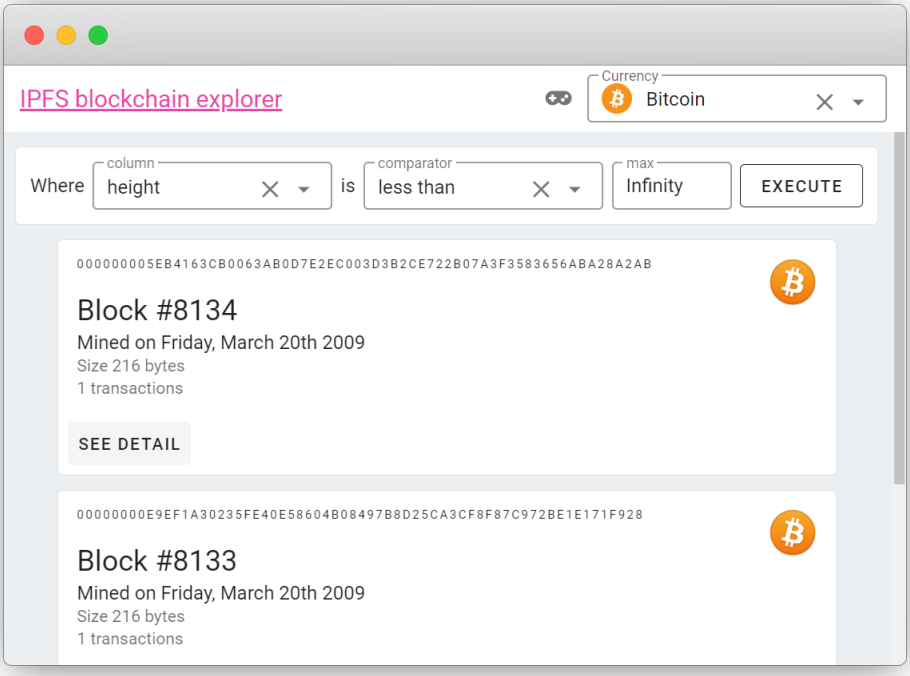


Figure 5.5: Blocks view

the order of the block that tasks 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 value. `blockPromise` is a task that loads blockchain block from IPFS and `done` is boolean, which signals if there are any more results in the subtree. Finally, task waits until blocks is 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.

5.2.2 ExplorerAPI

Node.js[9] implementation of IPFS uses filesystem to store data (see Figure **??**). On the top of ExplorerCore, there is a simple web framework Express[10], where are routes (endpoints) defined. Currently available routes are described in Section 4.2.3. Every route supports query parameters `filter` (used for filtering results) and `limit` (which reduces number of results). Pagination can be made by setting `filter` to be greater/smaller (depending on ordering) as key of the last displayed object and `limit` to page size. For example, if a user is looking at page of transactions ordered by time (ordered from the newest transactions to the oldest), the next page of transactions are first N transactions that happened before last displayed transaction (where N is page size).

[9] https://nodejs.org/
[10] http://expressjs.com/

```
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 5.6: Loading blocks from database

Rest API supports optional path param `path` that is useful for traversing objects in IPFS. If we want to get fifth transaction of the block with height 1000 one of the way is request URL `/block/998/next_block/next_block/txs/5` (get block with height 998, get next block two times, get transaction, a get fifth item from array of transactions). This approach allows the user to explore IPFS storage as graph very quickly by objects links.

Chapter 6

Benchmarking

6.1 Comparing to other database systems

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

6.1.1 MySQL

MySQL is free and open-source software under the terms of the GNU General Public License, and is also available under a variety of proprietary licenses. 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.

6.1.2 PostgreSQL

PostgreSQL also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance. It was originally 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 the default database for macOS Server, and is also available for Linux, FreeBSD, OpenBSD, and Windows.

6.1.3 Environment

We compare MySQL and PostgresSQL with our p2p database system. These database systems run virtualized in Docker containers on hardware with 4 core's processor Intel Pentium G4560 3.50GHz and 16GB of RAM. On another hardware we starts clients, that connects to these database systems a preforms queries. Than 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 6.1. 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 6.2.

```
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 KEY,
    first_name varchar(50),
    last_name varchar(50),
    email varchar(100),
    birthdate date,
    added timestamp
);

CREATE TABLE posts (
    id serial PRIMARY KEY,
    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 6.1: 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 + 1)) ;
```

(b) PostgreSQL

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

(c) Our database system

Figure 6.2: Test query

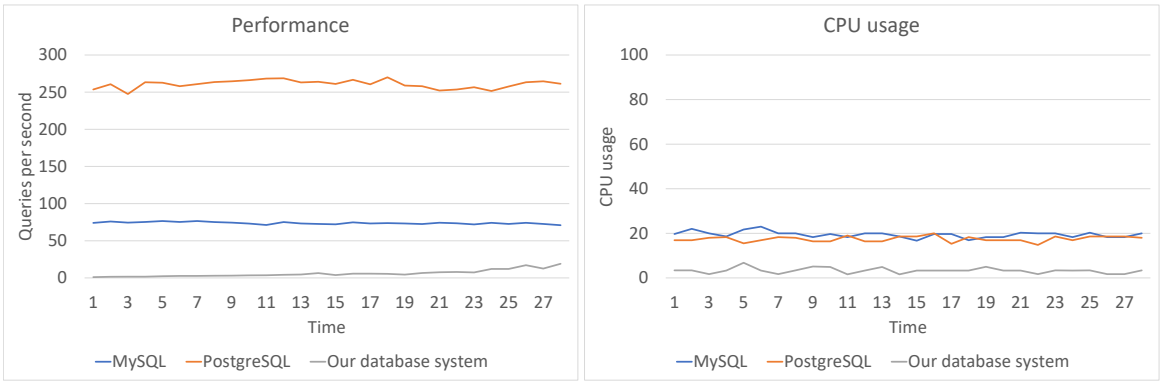
6.1.4 Single connected client

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 6.3a). 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 p2p network. CPU usage with a single connected client is for MySQL and PostgreSQL same - about 20% (see Figure 6.3b). For our database system it is only 3% due to lots of IO operations.

6.1.5 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 6.4a). 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

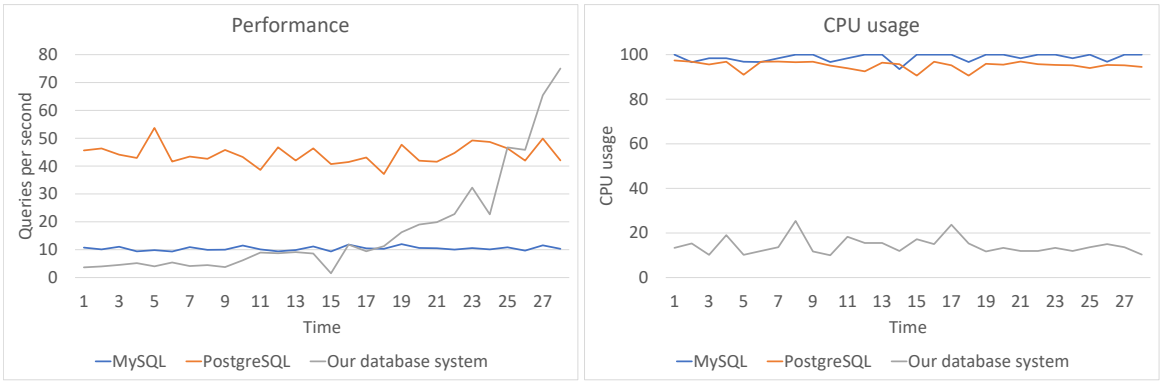
[1] https://www.npmjs.com/package/mysql
[2] https://www.npmjs.com/package/postgres



(a) Performance (b) CPU usage with

Figure 6.3: Single connected client benchmark plots

from other clients. Thanks to this our system only using less than 20% CPU power (see Figure 6.4b).



(a) Performance (b) CPU usage with

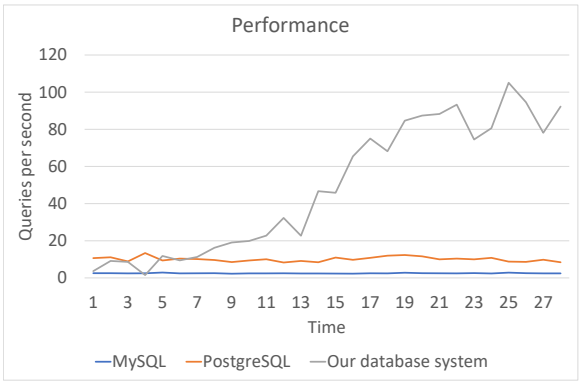
Figure 6.4: 20 connected clients benchmark plots

6.1.6 80 connected clients

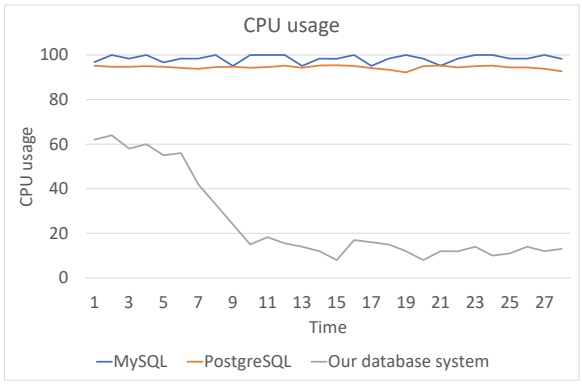
With 80 connected clients, our database system outperforms MySQL and PostgreSQL after few seconds. Every client of our database system performs 100 queries per second. PostgreSQL client performs only 10 queries per second and MySQL only 2 queries per second (see 6.5a). CPU usage is in our database system also very low. At the beginning, when content is not distributed on the clients, the central node uses 60 percents of CPU but it decreases over time (see 6.5b).

6.1.7 150 connected clients

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



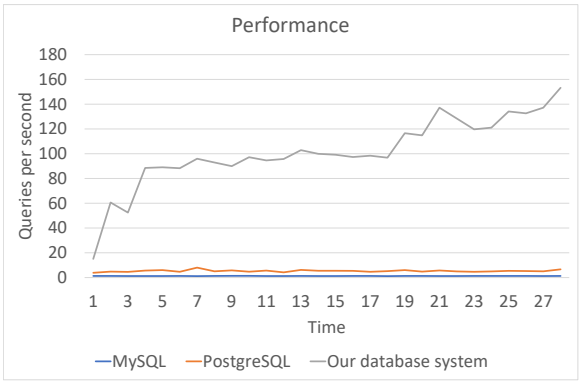
(a) Performance



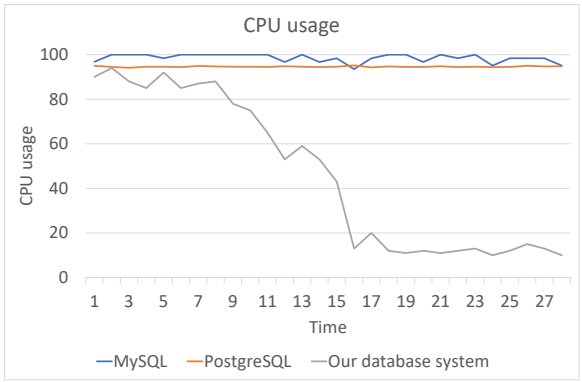
(b) CPU usage with

Figure 6.5: 80 connected clients benchmark plots

150 queries per second, and it increases by the time (see 6.6a). For the first seconds, CPU usage on the central node is high (near 100%), but it decreases over the time (see 6.6b).



(a) Performance



(b) CPU usage with

Figure 6.6: 150 connected clients benchmark plots

6.1.8 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 load on central node.

6.2 Comparing with blockchain explorers

6.2.1 Blockbook

Blockbook[3] is a blockchain indexer for Trezor Wallet[4], developed by SatoshiLabs[5]. It currently supports more than 30 coins (and the community implemented some others). For data storage Blockbook is using RocksDB[6] 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:

- **Not distributed** (client-server architecture) - problem with scaling for more users.
- **Not a 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.

[[Najprv je potrebne synchronizovat cely blockchain aby som mohol uskutocnit merania]]

6.2.2 Writing

6.2.3 Reading

6.2.4 Filtering

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

Chapter 7

Conclusion

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

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

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