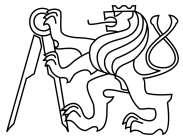
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Vizualizace a vyhledávání v distribuované databázi
Student:	Ing. arch. Šimon Steklík
Vedoucí:	Ing. Tomáš Zahradnický, Ph.D.
Studijní program:	Informatika
Studijní obor:	Znalostní inženýrství
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Seznamte se s technologií Kibana a se strukturou vedoucím zadané grafově orientované databáze (vstupní data) a dále databází, do které budou vstupní data importována (struktura a import databáze je řešen v jiné DP). V této databázi navrhněte a implementujte pomocné struktury pro snadnější vyhledávání požadovaných informací. Dále navrhněte a implementujte webovou aplikaci pro vyhledávání podle vedoucím zadaných kritérií, minimálně však: i) všechny cesty ze shluku (množiny uzlů) A do shluku B a ii) sousední shluky ke shluku A, vše s možností omezení časové značky a váhy. Aplikace bude sloužit pro zadávání vyhledávacích kritérií uživatelem a vizualizaci výsledků (grafy, diagramy) v prostředí Kibana, příp. po dohodě s vedoucím práce, v jiném vhodném vizualizačním prostředí. Ověřte funkčnost implementované aplikace a vyhodnoťte dosažené výsledky.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 16. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Diplomová práce

Vizualizace a vyhledávání v distribuované databázi

Bc. Šimon Steklík

Vedoucí práce: Ing. Tomáš Zahradnický, Ph.D.

8. května 2017

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Tomáši Zahradnickému, Ph.D. za možnost pracovat na tomto tématu. Poděkování patří i mé rodině za podporu po celou dobu studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 8. května 2017 .....

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Steklík, Šimon. *Vizualizace a vyhledávání v distribuované databázi.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací prototypu webové aplikace sloužící k prohledávání dat odvozených z distribuované databáze Blockchain. Aplikace také umožní hledání vazeb mezi těmito daty a vizualizaci výsledků.

Klíčová slova blockchain, bitcoin, hledání, vizualizace, graf, webová aplikace, Java, framework Spring

Abstract

The topic of this master's thesis is the design and implementation of a prototype of a web application which will be used for searching in data derived from Blockchain distributed database. The application will also allow its user to search for links in the data and visualize the results.

Keywords blockchain, bitcoin, search, visualization, graph, web application, Java, Spring framework

Obsah

Seznam obrázků

Úvod

V dnešní době je těžké představit si svět bez digitálních plateb. Bankovní účet mají v České republice více než čtyři pětiny lidí a podíl plateb převodem nebo kartou (zejména po příchodu bezkontaktních karet) se neustále zvyšuje. Mnoho lidí má účty i u několika bankovních institucí.

Pojem „instituce“ je zde důležitý – dnešní standardní digitální platby fungují tak, že lidé své peníze svěřují nějaké instituci – bance, družstevní záložně – a ta jim na oplátku zaručuje bezpečí jejich peněz a umožňuje (mimo jiné) snadné digitální platby. Člověk tedy směňuje (relativní) bezpečí uložených peněz a jednoduchost běžných plateb za malé riziko, že o tyto prostředky přijde (nebo s nimi aspoň nebude moci po nějakou dobu disponovat), pokud např. daná instituce zkrachuje. Je zde tedy důležitá *důvěra* klienta v jeho banku.

Banky a další instituce jako např. karetní společnosti fungují jako prostředník mezi odesílatelem a příjemcem peněz a tuto funkci samozřejmě nezastávají zadarmo – za každou transakci si (různými způsoby, třeba i nepřímo) vybírají poplatek. Tento poplatek může být u komplikovanějších transakcí – např. při převodu měn – značný.

Další důležitou vlastností dnešních digitálních plateb je *dohledatelnost* – banky u každého účtu znají jeho vlastníka a veškeré převody peněz jde tedy přiřadit ke konkrétním osobám.

Tato práce se zabývá zkoumáním vazeb v prostředí, kde takový identifikační záznam neexistuje – jde o distribuovanou databázi Blockchain. Tato databáze zastává funkci distribuované „účetní knihy“ transakcí digitální měny Bitcoin, která vznikla za účelem odstranění vnímaných nedostatků běžných měn a která řeší i tzv. double-spending problém, tj. dvojí utracení stejných prostředků. V jejím prostředí není třeba důvěřovat centrální autoritě – žádná tu neexistuje. Poplatky za transakce jsou řádově nižší než u běžných transakcí.

Není potřeba nikde poskytovat své osobní údaje nebo se jinak identifikovat – systém to nepotřebuje.

Díky těmto vlastnostem – zejména kvůli absenci potřeby se jako konkrétní osoba identifikovat – je tento způsob plateb využíván v případech, kdy existuje potřeba utajit odesílatele a/nebo příjemce platby. Tato potřeba může být naprosto legitimní, ale samozřejmě existují i případy, kdy tomu tak není. V těchto případech by se hodila možnost prohledávat data blockchainu, hledat vazby mezi jeho prvky a tyto vazby vizualizovat. A právě tímto tématem se zabývá tato diplomová práce – konkrétně vytvořením vyhledávací aplikace nad databází odvozenou z dat blockchainu.

Kapitola 1

Cíl práce

Cílem této práce je navrhnout a implementovat prototyp webové aplikace sloužící k vyhledávání v datech distribuované databáze blockchain. Tato databáze není použita přímo – pracuje se databází odvozenou z původních dat a obsahující další pomocné struktury usnadňující vyhledávání (tato odvozená databáze je předmětem jiné diplomové práce [1]).

Navrhovaná aplikace by měla umět hledat vazby mezi prvky vstupní databáze, konkrétně cesty (posloupnosti transakcí) mezi zadanými bitcoinovými adresami, resp. skupinami adres, a dále prohledávat okolí dané skupiny adres. Výsledky těchto hledání by měly být vhodným způsobem vizualizovány.

Nejprve bude popsána doména, kterou se práce zabývá, tedy distribuovaná databáze blockchain a její použití v digitální měně Bitcoin. Potom bude analyzována odvozená (vstupní) databáze, budou popsány a upřesněny požadavky na aplikaci a případy jejího užití.

V návrhové části budou popsány zvolené principy a podoba vyhledávací aplikace, budou zvoleny vhodné technologické prostředky pro práci se vstupní databází a pro prezentaci výsledků a bude navržena architektura aplikace. Podle návrhu bude poté aplikace impementována. Výsledek pak bude testován, zejména funkčnost a výkon vyhledávacích dotazů.

Kapitola 2

Blockchain

V této kapitole budou popsány principy fungování blockchainu, konkrétně blockchainu digitální měny Bitcoin. Jelikož cílem této práce je prohledávání dat blockchainu, bude se kapitola zabývat zejména jeho strukturou (bloky, transakce, adresy) a ostatní principy budou vysvětleny jen velmi stručně.

2.1 Princip

Jak již bylo uvedeno v úvodu, blockchain je distribuovaná databáze (dalo by se také říct účetní kniha nebo log) bitcoinových transakcí. Jde vlastně o chronologický řetězec záznamů o transakcích, který je sdílený všemi uzly bitcoinové sítě. Každý uzel této P2P sítě má k dispozici celý blockchain nebo jeho část a na základě informací od ostatních uzlů se nezávisle rozhoduje, zda nový záznam do blockchainu přidat nebo ne (resp. kterou verzi konfliktních záznamů přidat).

V blockchainové síti jsou 2 druhy uživatelů: *běžní uživatelé* a *těžaři* (miners). *Běžní uživatelé* vytvářejí transakce a *těžaři* je (za určitou odměnu) potvrzují a přidávají do sdílené databáze.

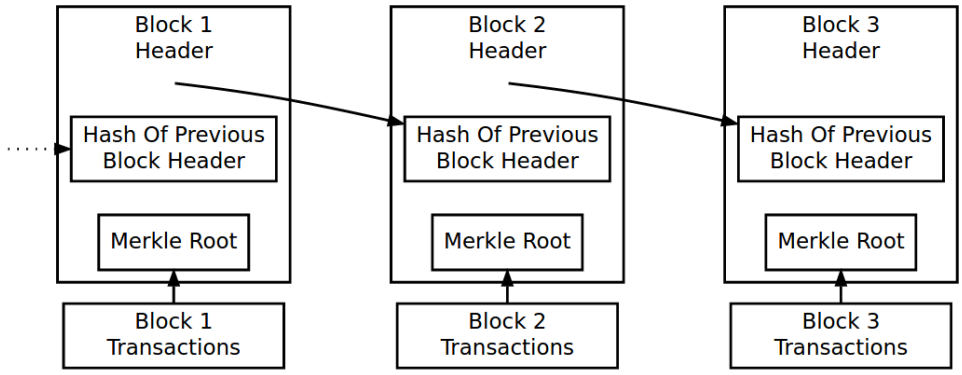
Tomuto rozdělení odpovídají i 2 druhy záznamů v blockchainu: *transakce* a *bloky*. *Transakce* jsou záznamy o jednotlivých platbách (např. *Alice posílá Bobovi 0.05 BTC*), *bloky* jsou kolekce transakcí potvrzených těžaři a přidaných do blockchainu. Jak transakce, tak bloky jsou chráněny proti následné změně a čím déle jsou součástí blockchainu, tím těžší je jejich případná změna (bude podrobněji vysvětleno v následujících sekcích) [2].

2.2 Základní prvky

2.2.1 Bloky

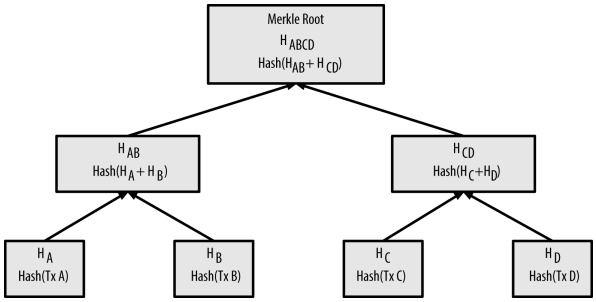
Blok je kolekce transakcí potvrzená těžaři a přidaná do sdílené databáze. Kromě samotných transakcí obsahuje *block header* - hlavičku, která daný blok identifikuje a obsahuje informace zamezující jeho pozdějším úpravám. Hlavička má následující části:

- hash hlavičky předchozího bloku – tato hash musí splňovat určité podmínky (které zde nebudeme rozebírat) a nalezení platné hashe trvá určitou dobu (to má své důvody v zabezpečení – viz níže).
- hash transakcí náležejících tomuto bloku – jde o tzv. *Merkle Root* – transakce jsou spárovány, zahashovány a výsledné hashe jsou opět párovány a hashovány tak dlouho, až zbude jen jedna hash – kořen stromu (viz obrázek 2.2) [3].
- *block height* – pořadí bloku v blockchainu. Toto číslo ale nemusí být jednoznačným identifikátorem bloku, protože může existovat více bloků se stejnou *block height* – 2 nebo více težařů vytvoří blok se stejným číslem v krátkém časovém úseku a připojí ho do blockchainu (tzv. *fork*), a dokud síť nedojde ke konsenzu (jeden blok "vyhraje"), jsou všechny svým způsobem součástí blockchainu.



Obrázek 2.1: Bloky - schéma (zdroj: [2])

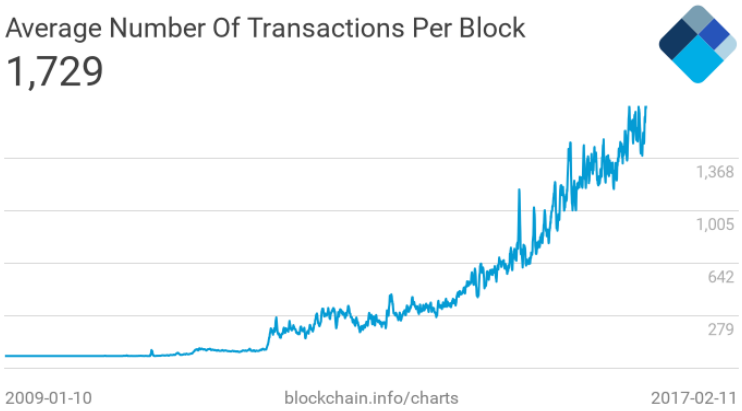
Jelikož hash každé transakce určuje celkovou hodnotu Merkle root, ta je zase součástí hlavičky bloku a hash hlavičky bloku je součástí hlavičky následujícího bloku, pro změny nějaké transakce by bylo třeba modifikovat její blok a všechny následující bloky. To je kvůli podmínkám kladeným na hash



Obrázek 2.2: Merkle tree (zdroj: [2])

hlavičky bloku (nalezení platné hodnoty trvá určitou dobu) výpočetně velmi náročné a zabraňuje to pokusům o modifikaci blockchainu.

Počet transakcí v jednom bloku se neustále zvyšuje (každý rok se přibližně zdvojnásobí), v současnosti jde o přibližně 1700 transakcí/blok [4].

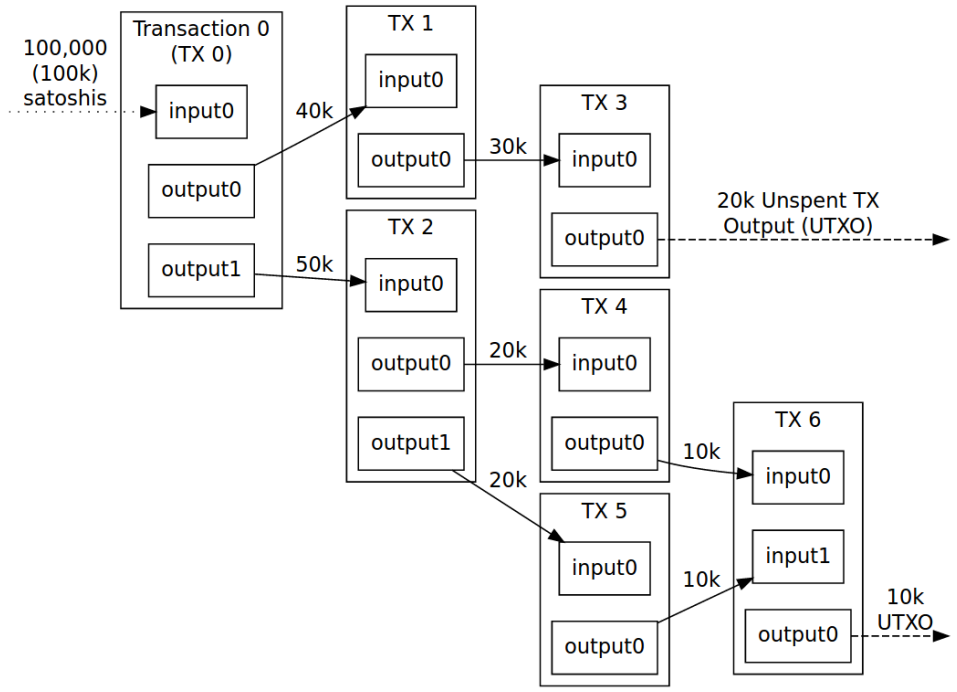


Obrázek 2.3: Vývoj počtu transakcí v jednom bloku (zdroj: [4])

2.2.2 Transakce

Bitcoinové peněženky z pohledu uživatelů vytvářejí dojem, že *transakce* je přenos Bitcoinů z peněženky na *adresu*. Ve skutečnosti Bitcoiny putují z transakce do transakce – každá transakce má n vstupů a m výstupů (počty nemusí být stejné) a výstup jedné transakce může být vstupem další transakce (viz obrázek 2.4). Každý výstup může být použit jako vstup pouze jednou, jinak dochází k tzv. *double-spend* a jedna z transakcí bude sítí odmítnuta. Zatím nepoužitý výstup transakce se nazývá *UTXO (Unspent Transaction Output)*.

Celková hodnota výstupů z transakce může být menší než hodnota vstupů – rozdíl pak slouží jako odměna pro těžaře za potvrzení transakce (díky tomu lze transakce prioritizovat – vyšší odměna vede težaře k rychlejšímu potvrzení transakce).



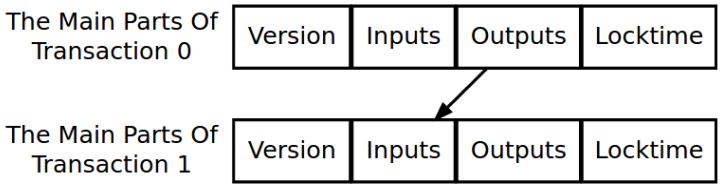
Obrázek 2.4: Řetěz transakcí (zdroj: [2])

Každý blok obsahuje jako první tzv. *Coinbase transakci* – jde o speciální transakci bez vstupů, jejímž výstupem je odměna težaře (suma odměn všech transakcí + pevná odměna za vytvoření bloku).

Všechny ostatní transakce mají následující atributy:

- alespoň jeden vstup – každý vstup obsahuje identifikátor zdrojové transakce (*txid*), index jejího *UTXO* a tzv. *signature script* (viz dále).
- alespoň jeden výstup – každý výstup má index (počítá se od nuly), hodnotu Bitcoinů, které obsahuje a tzv. *pubkey script* (viz dále).
- verzi transakce – pravidla pro validaci transakcí se občas mění, toto číslo určuje, jaká verze pravidel se má použít.

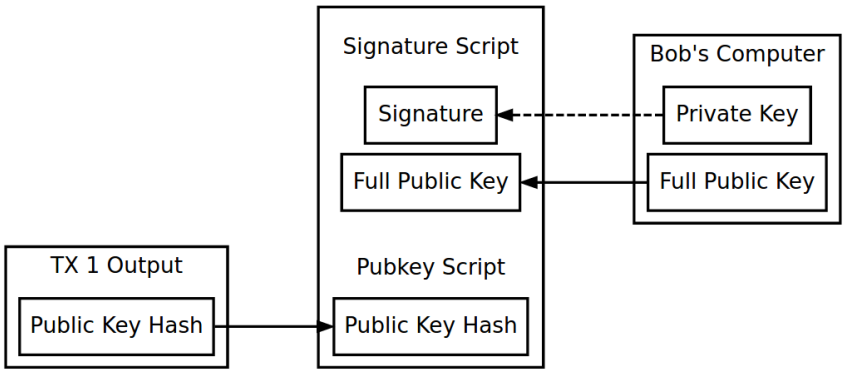
- tzv. *locktime* – čas, kdy se transakce stane validní a může být potvrzena a přidána do bloku. Toho se používá, pokud je potřeba, aby původce transakce měl možnost ji do nějakého časového intervalu odvolat.



Obrázek 2.5: Základní atributy transakce (zdroj: [2])

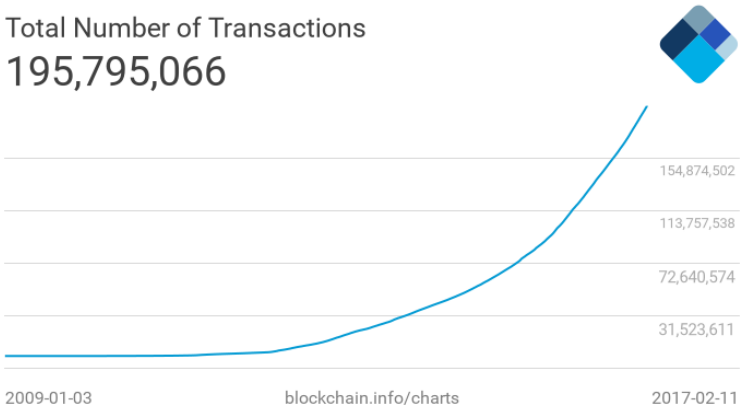
Pubkey script je řetězec znaků, který určuje, jaké podmínky musí splnit subjekt, který chce utratit daný *UTXO*. Většinou jde o veřejný klíč (resp. jeho hash), tzn. *UTXO* může utratit jen ten, kdo disponuje odpovídajícím privátním klíčem.

Signature script používá subjekt, který chce utratit *UTXO* jako důkaz, že může s danými prostředky disponovat. Obsahuje podpis transakce vygenerovaný pomocí privátního klíče odpovídajícího veřejnému klíči z *pubkey scriptu* daného *UTXO*. Díky tomu je nejen možné ověřit, že subjekt disponuje privátním klíčem, ale zároveň to znemožňuje upravit transakci při přenosu po P2P síti (podpis by se stal neplatným).



Obrázek 2.6: Pubkey script, signature script (zdroj: [2])

Celkový počet Bitcoinových transakcí je teď na hodnotě cca 200 milionů a každý rok se zvyšuje cca na dvojnásobek (což odpovídá tempu růstu velikosti bloků) [5]. Za několik let bude tedy počet transakcí v jednotkách miliard.



Obrázek 2.7: Vývoj počtu Bitcoin transakcí (zdroj: [5])

2.2.3 Adresy

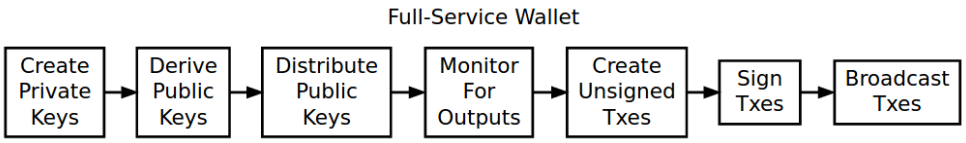
Pokud chce někdo přijmout platbu v Bitcoinech, poskytne plátci svojí *adresu* – 20-bytovou reprezentaci *pubkey scriptu*, která obsahuje verzi adresy, hash veřejného klíče a kontrolní součet. Je důležité zmínit, že adresa není nijak svázána s konkrétní osobou a že počet adres patřících jednomu subjektu není nijak omezen – jde v podstatě jen o reprezentaci veřejného klíče, a těch si každý může vytvořit, kolik chce. Naopak často platí (a doporučuje se to i v každém návodu na Bitcoinové platby), že každá nová platba pro danou osobu vede na novou adresu. Jen si pak tato osoba musí v nějaké formě vést záznam o tom, na kterých adresách se nachází její prostředky. K tomuto účelu slouží *Bitcoinové peněženky*.

2.2.4 Peněženky

Bitcoinové peněžěnky jsou prostředníky mezi běžnými uživateli a Bitcoinovou sítí. Odstiňují uživatele od implementačních detailů a umožňují používat Bitcoinové platby stejným (nebo podobným) způsobem jako on-line platby kartou či převodem na účet.

V zásadě poskytují všechny nebo některé ze služeb uvedených na obrázku 2.8 – vytvářejí privátní/veřejné klíče, vedou záznam o odpovídajících adresách a prostředcích na nich a vytvářejí transakce (odchozí platby). Tyto služby nemusí kvůli bezpečnosti poskytovat v rámci jedné aplikace. Např. jedna aplikace může sloužit k vytváření klíčů a podepisování transakcí – tato aplikace je na počítači bez připojení k internetu, aby nemohlo dojít k odcizení privátních klíčů po síti – a druhá aplikace vytváří transakce a komunikuje s P2P sítí.

Z našeho pohledu je důležité, že samotná adresa (většinou) nikoho neidentifikuje, ale peněženka (resp. data v ní uložená) už ano a můžeme poté s množinou adres z peněženky pracovat jako s jedním subjektem.



Obrázek 2.8: Peněženka – přehled činností (zdroj: [2])

Kapitola 3

Analýza

V této kapitole je popsána problematika této diplomové práce, zejména vstupní databáze (odvozená z původní distribuované databáze a zpracovávaná v jiné diplomové práci[1]). Druhá část kapitoly se zabývá popisem a zpřesněním požadavků na navrhovanou vyhledávací aplikaci a specifikací jejích případů užití.

3.1 Vstupní data

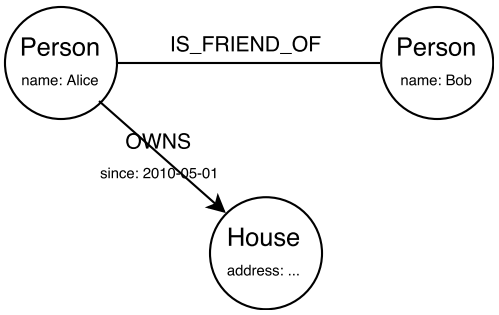
Vstupními daty této práce je grafová databáze zpracovaná v jiné diplomové práci [1]. V této sekci je databáze popsána jak po technologické stránce, tak po stránce struktury dat.

3.1.1 Grafové databáze

Grafové databáze slouží k ukládání dat v podobě grafu. Tento graf se skládá v základu ze dvou prvků:

- *uzly* – představují objekty/entity. Mohou mít přiřazené vlastnosti a typ.
- *hrany* – znázorňují vztahy mezi objekty/entitami. Mohou být orientované. Mohou mít přiřazené vlastnosti a typ (viz obrázek 3.1.

Na rozdíl od klasických relačních databází poskytují grafové databáze tzv. *bezindexovou sousednost*, tzn. každý uzel obsahuje přímé odkazy na své sousedy [6]. Díky tomu není potřeba používat index a procházení grafu je řádově rychlejší. Na druhou stranu to znamená větší velikost dat, protože každý vztah je explicitně uložen.



Obrázek 3.1: Příklad grafové struktury

Grafové databáze mají také flexibilní schéma – je možné přidávat nové typy uzlů a hran za běhu. Co je naopak složitější, je změna vlastnosti všech (nebo podmnožiny) existujících uzlů/hran [7].

3.1.2 Neo4j

Vstupní data jsou uložena v databázi *Neo4j*. Jde o nejrozšířenější grafovou databázi [8]. Mezi její hlavní vlastnosti patří [9]:

- Open source
- Napsaná v Javě
- multiplatformní - poskytuje drivery a knihovny pro všechny populární programovací jazyky
- splňuje ACID vlastnosti
- dotazovací jazyk Cypher
- teoreticky "neomezená"velikost (neomezený počet uzlů) [10]

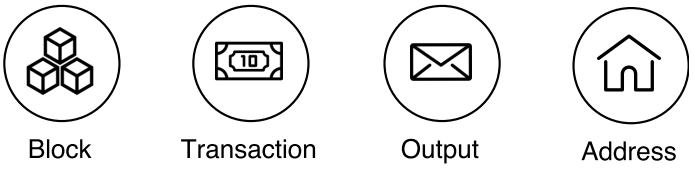
3.1.3 Struktura databáze

V této sekci bude popsána struktura vstupních dat – jednotlivé typy uzlů a hran, jejich vztahy a důležité vlastnosti. Databáze obsahuje původní data blockchainu převedená do grafové podoby a dále data z nich odvozená.

Odvozená data obsahují dvě podskupiny – první skupina slouží pro snadnější procházení grafu (např. tranzitivní hrany) a druhá obsahuje uzly a hrany představující skutečnosti, které nejsou v samotném blockchainu explicitně přítomné (např. entity – skupiny adres).

3.1.4 Původní data

Tato data představují základní prvky blockchainu (v závorce je vždy `label`, pod jakým je prvek uložen v Neo4j).



Obrázek 3.2: Původní data – použité ikony[1]

3.1.4.1 Uzly

Blok (`Block`) Blok sdružuje skupinu transakcí potvrzených ve stejný čas. Vlastnosti:

- *timestamp* – časová značka vytvoření bloku (všechny transakce tohoto bloku mají stejnou)
- *height* – index bloku (viz. sekce 2.2.1)

Transakce (`Transaction`) Odpovídá transakci v blockchainu – slouží k přesunu bitcoinů z množiny adres na jinou množinu adres. Vlastnosti:

- *txid* – identifikátor transakce (transakce zahashovaná pomocí sha256d)
- *timestamp* – časová značka potvrzení transakce
- *coinbase* – příznak značící Coinbase transakci (viz sekce 2.2.2)

Výstup transakce (`Output`) Jeden z výstupů transakce (patří k nějaké adrese). Vlastnosti:

- *n* – index výstupu transakce (viz sekce 2.2.2)
- *value* – hodnota tohoto výstupu (v BTC)
- *type* – způsob potvrzení vlastnictví tohoto výstupu (pubkey script)

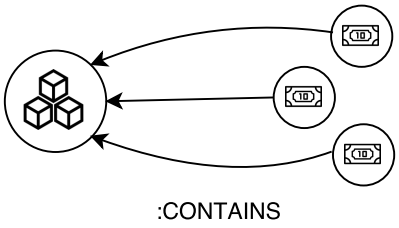
Adresa (`Address`) Adresa sdružuje výstupy transakcí (používající stejný pubkey script). Vlastnosti:

- *address* – identifikátor adresy (25-36 alfanumerických znaků)
- *balance* – suma nepoužitých výstupů transakcí náležejících k této adrese v době importu dat

[1] autor ikon: Freepik, www.flaticon.com

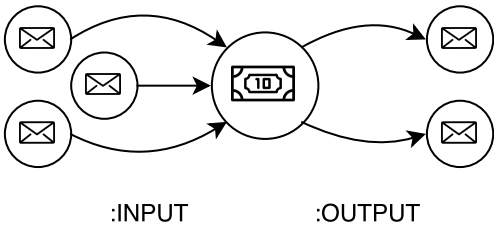
3.1.4.2 Hrany

`CONTAINS` Vztah mezi blokem a jeho transakcemi.



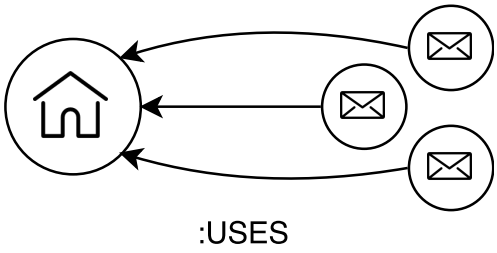
Obrázek 3.3: Schéma relace `CONTAINS`

`INPUT, OUTPUT` Vztah mezi vstupem (nepoužitý výstup předchozí transakce) a transakcí, resp. transakcí a jejím výstupem.



Obrázek 3.4: Schéma relací `INPUT`, `OUTPUT`

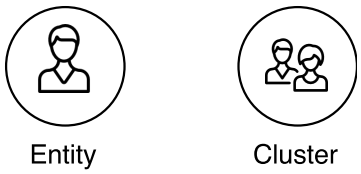
`USES` Vztah mezi výstupem transakce a adresou.



Obrázek 3.5: Schéma relace `USES`

3.1.5 Odvozená data

V této podsekci jsou popsány uzly a hrany odvozené z původních dat (v závorce je vždy `label`, pod jakým je prvek uložen v Neo4j).



Obrázek 3.6: Odvozená data – použité ikony[2]

3.1.5.1 Uzly

Entita (`Entity`) Entita sdružuje skupinu adres, u kterých se domníváme, že patří jednomu subjektu (princip sdružování adres je součástí jiné diplomové práce [1]). Vlastnosti:

- *id* – (technický) identifikátor entity
- *balance* – suma nepoužitých výstupů transakcí náležejících k adresám této entity v době importu dat

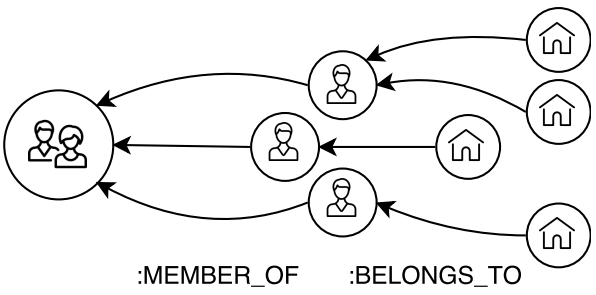
Shluk (`Cluster`) Shluk sdružuje skupinu entit, mezi kterými probíhá větší množství transakcí (princip je opět součástí jiné diplomové práce [1], jen uvedu, že jde o algoritmus *Chinese Whispers*). Vlastnosti:

- *id* – (technický) identifikátor shluku

3.1.5.2 Hrany

`BELONGS_TO` Vztah mezi adresou a její entitou.

`MEMBER_OF` Vztah mezi entitou a jejím shlukem.



Obrázek 3.7: Schéma relací `MEMBER_OF`, `BELONGS_TO`

[2]autor ikon: Freepik, www.flaticon.com

`RECEIVED` Vztah mezi transakcí a entitou – tranzitivní hrana reprezentující posloupnost *transakce→výstup→adresa→entita*, tzn. znázorňuje tok peněz z transakce do entity.

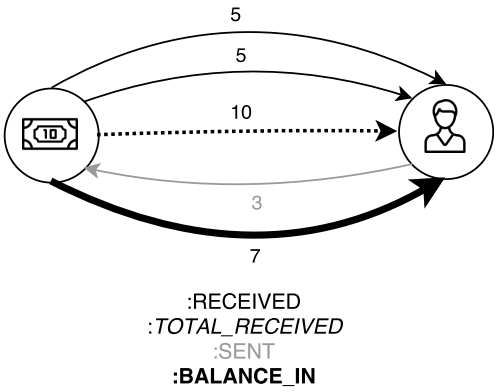
`SENT` Obdoba hrany `RECEIVED`, ale pro výstupní transakce *entita→adresa→výstup (který je vstupem další transakce)→transakce*

`TOTAL_RECEIVED` Vztah mezi transakcí a entitou – suma hran `RECEIVED` pro konkrétní entitu a transakci (viz. obrázek 3.8).

`TOTAL_SENT` Vztah mezi entitou a transakcí – suma hran `SENT` pro konkrétní entitu a transakci.

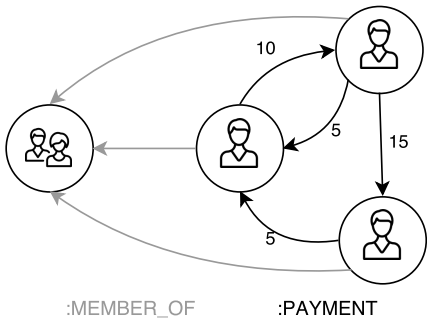
`BALANCE_OUT` Vztah mezi entitou a transakcí – rozdíl hran `TOTAL_SENT` a `TOTAL_RECEIVED` pro konkrétní entitu a transakci (entita může poslat prostředky sama sobě, proto je tato hrana potřeba – vyjadřuje celkovou výstupní bilanci z entity do transakce).

`BALANCE_IN` Vztah mezi transakcí a entitou – obdoba `BALANCE_OUT`, ale v opačném směru – vyjadřuje celkovou vstupní bilanci z transakce do entity).



Obrázek 3.8: Schéma platebních relací mezi transakcí a entitou (výstupní hrany fungují obdobně)

`PAYMENT` Hrana znázorňující platbu mezi dvěma entitami (jde vlastně o posloupnost *entita→transakce→entita* přes hrany `BALANCE_IN` a `BALANCE_OUT`). Entity, mezi kterými existuje větší množství těchto hran, tvoří dohromady shluk (`Cluster`, viz obrázek 3.9).



Obrázek 3.9: Schéma hrany `PAYMENT`

3.2 Požadavky

Obsahem této sekce je popis funkčních požadavků na vyvíjenou aplikaci vycházejícíze zadání diplomové práce. Dále zde budou obecně nastíněny způsoby, jak dosáhnout splnění těchto požadavků.

3.2.1 Funkční požadavky

Ze zadání vyplývají v zásadě 2 základní funkční požadavky:

1. *Najít všechny cesty ze shluku (množiny uzlů) A do shluku B.*

2. *Najít sousední shluky ke shluku A.*

Obojí s možností omezení časové značky a váhy.

Shlukem v tomto kontextu se rozumí množina bitcoinových adres. Odpovídá typu `Entity` ze vstupní databáze – ve zbytku práce bude slovo „entita“ používáno v tomto významu[3] (pokud z kontextu nevyplyne jinak). *Cesta* je posloupnost transakcí mezi entitami.

Časovou značkou se rozumí doba provedení transakce.

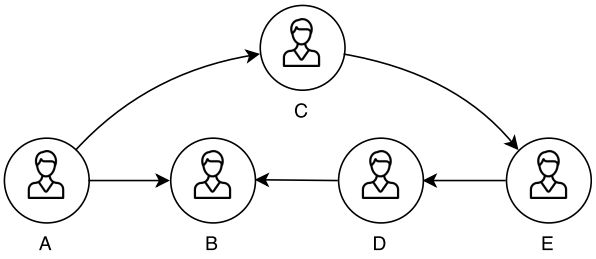
Váha hrany na cestě je vlastně hodnota transakce mezi entitami, resp. hodnota výstupu transakce náležejícího cílové entitě (transakce může mít několik výstupů a některé z nich nemusí náležet do hledané cesty).

[3]Ve vstupní databázi se také vyskytuje typ uzlu `Cluster`, pokud budu mluvit o „shluku“, myslím tím právě tento typ uzlu

3.2.2 Hledání cest mezi dvěma entitami

Protože *cesta* je posloupnost transakcí, můžeme pro její hledání použít hrany `PAYMENT` propojující entity – tyto hrany přesně odpovídají námi hledané relaci. Bylo by samozřejmě možné hledat cestu i pomocí hran a uzlů, z kterých `PAYMENT` tranzitivně vzniká (viz. 3.1.5), ale tyto původní vztahy je vždy možné z hrany `PAYMENT` odvodit. Je tedy výhodnější použít první možnost jak z hlediska snadnější implementace, tak z hlediska výkonu (průchod grafem je rychlejší).

Směr cesty Za zvážení stojí, co je považováno za validní cestu z hlediska jejího směru – na obrázku 3.10 je situace, ve které je cílem najít cesty z entity A do entity E. Validní je určitě cesta $A \rightarrow C \rightarrow E$. Ale co cesta $A \rightarrow B \rightarrow D \rightarrow E$? Transakce se sbíhají v entitě B, tzn. žádné bitcoiny patřící entitě A se touto cestou k entitě E nemohly dostat. Na druhou stranu informace, že obě entity (možná) posílaly bitcoiny stejnému příjemci (entita B) může být zajímavá.



Obrázek 3.10: Cesty mezi entitami

Došel jsem k závěru, že pro tento problém neexistuje jedno „správné“ řešení – vždy záleží na kontextu konkrétního hledání. Řešením v tomto případě je umožnit uživateli aplikace, aby sám specifikoval, jestli ho zajímají cesty orientované v určitém směru, nebo na směru (toku bitcoinů) nezáleží.

Časová značka Jelikož hrana `PAYMENT` odpovídá konkrétní transakci, obsahuje i její časovou značku (atribut `timestamp`). Jde tedy o to umožnit uživateli aplikace specifikovat časové rozmezí, které ho zajímá, a omezit průchod grafem jen na hrany, jejichž časová značka náleží do tohoto rozmezí.

Váha (hodnota) hrany Jak už bylo řečeno, hrana `PAYMENT` obsahuje i hodnotu bitcoinů, převedených v rámci odpovídající transakce mezi dvěma entitami. Je tedy možné hledat převody od určité minimální hodnoty, příp. převody mající přesně konkrétní hodnotu.

Hloubka hledání Dalším parametrem, který ovlivňuje hledání, je jeho hloubka – maximální délka cesty, která bude zahrnuta ve výsledcích. Tento parametr

má dvojí význam: z business pohledu mohou uživatele zajímat pouze cesty do určité délky (ať už z důvodu snadnější interpretovatelnosti, kvůli velkému množství výsledků nebo z jiného důvodu). Významným způsobem to ale také ovlivní výkon aplikace (čas hledání), protože s každým krokem cesty roste okolí entity exponenciálně. Bude proto třeba provést měření času pro různé hloubky hledání a omezit maximální hloubku na vhodnou hodnotu.

3.2.3 Hledání okolí – „sousední“ entity

Hledání okolních entit funguje v zásadě velmi podobně jako hledání cest mezi dvěma entitami, jen máme pouze jeden výchozí bod a výsledkem jsou cesty do všech okolních entit. Velký vliv zde bude mít nastavení hloubky hledání. Omezení výsledků na cesty v určitém časovém období a s určitou hodnotou funguje obdobně jako při hledání cest mezi dvěma entitami.

3.2.4 Shluky (clustery)

Kromě entit obsahuje vstupní databáze také *clustery* – množiny entit, u kterých existuje mnoho vzájemných transakcí. V aplikaci by tedy mělo být možné pro danou entitu zobrazit její cluster a příp. i všechny ostatní entity a transakce, které do daného clusteru patří.

3.2.5 Souhrnné informace o hledaných objektech

V předchozím popisu požadavků bylo řešeno zobrazení prvků blockchainu (a odvozených prvků) a jejich vztahů, pravděpodobně v podobě grafu. Pro dobré porozumění datům je ale potřeba poskytnout uživateli i nějaké souhrnné informace o jednotlivých prvcích – např. z grafu se jen velmi těžko vyčte, jakými prostředky disponuje v určitém časovém okamžiku konkrétní adresa nebo jaké je pořadí transakcí dané entity. Proto pro jednotlivé typy prvků bude možnost zobrazit následující informace:

- Entita
  - zůstatek (součet zůstatků všech jejích adres)
  - počet adres
  - seznam adres
  - počet transakcí
  - seznam transakcí
  - graf vývoje zůstatku
- Adresa
  - zůstatek
  - počet transakcí
  - seznam transakcí
  - graf vývoje zůstatku
- Cluster

  - počet entit
  - seznam entit
  - počet transakcí
  - seznam transakcí
- Transakce
  - časová značka
  - seznam vstupů
  - seznam výstupů

3.2.6 Obohacení o informace mimo blockchain

Blockchain obsahuje informace o transakcích a adresách a umožňuje z nich odvodit existenci entit a clusterů. Co už ale neumožňuje je identifikovat jen na základě jeho dat konkrétní osoby nebo skupiny osob – bitcoinový blockchain (na rozdíl od klasických bankovních systémů) sám o sobě neobsahuje žádný registr uživatelů. Vlastnictví bitcoinů je dáno pouze schopností prokázat vlastnictví privátního klíče náležejícího z nepoužitému výstupu transakce.

To ale neznamená, že ke konkrétním adresám neexistují informace umožňující alespoň přibližnou identifikaci vlastníka. Na různých fórech, ve veřejných chatroomech i jinde na internetu lidé často zveřejňují své bitcoinové adresy, a pokud jsou tyto zdroje veřejně přístupné, je možné možné díky tomu přiřadit k adrese např. konkrétního uživatele fóra. Nejde tedy o přímou identifikaci konkrétní osoby, ale je jistě jednodušší identifikovat osobu z uživatele fóra než z několika adres a transakcí na blockchainu.

V této práci bude řešeno získání a import dat z fóra bitcointalk.org, konkrétně podpisy (signature) uživatelů – tyto podpisy často obsahují adresu s žádostí o zaslání bitcoinů. Určitě nejde o nejlepší zdroj pro získávání informací o adresách, ale poslouží jako ukázka využití tohoto principu a zároveň budou uložení dat a aplikace navrženy takovým způsobem, aby bylo možné budoucí přidávání dalších zdrojů identifikačních dat.

3.2.7 Uživatelské rozhraní

Blockchain je jednolitá grafová struktura s řádově miliardami uzlů, jejíž procházení je kontinuální proces – lze začít v libovolném bodě a přes množinu hran procházet graf prakticky neomezeně dlouho. Takový způsob procházení a vyhledávání by měla umožnit i navrhovaná aplikace – mělo by být možné rozvíjet výsledek prvotního dotazu; při vyhledání cest mězi dvěma entitami by mělo být možné ihned jednoduše hledat a zobrazit okolí některé z entit na této cestě, zobrazit její adresy apod.

Uživatelské rozhraní jsem se proto rozhodl členit na 3 základní části:

- *vyhledávací formuláře* – formuláře umožňující prvotní hledání pomocí známých adres, entit apod., které zároveň umožní určit další parametry hledání (časový rámec, směr cest, ...).
- *interaktivní graf* – zobrazuje výsledky vyhledávání ve formě grafu uzlů a hran, umožňuje s výsledky interagovat a spouštět další dotazy (přes kontextovou nabídku nebo obdobně).
- *informační panel* – zobrazuje souhrnné informace o vyhledávání a o vybraných prvcích grafu (informace uvedené v sekci 3.2.5).

3.2.8 Nefunkční požadavky

Zde budou popsány požadavky na aplikaci, které nesouvisí s jejími funkcemi.

Počítač bez připojení k internetu Aplikace bude provozována na jednom počítači bez připojení k internetu – z toho vyplývá např. nemožnost použití aplikačních rozhraní (API) třetích stran.

Použití technologie Kibana Ze zadání vyplývá možnost použití technologie *Kibana*[4], nicméně tato technologie se používá jako front-end pro data uložená v databázi *ElasticSearch*[5] a s jinými databázemi (prakticky) pracovat neumí. *ElasticSearch* je pro účely procházení grafu méně vhodný než *Neo4j* (vstupní databáze). Zároveň by toto obnášelo převod dat mezi databázemi a v neposlední řadě je vytváření vlastních (nestandardních) prvků uživatelského rozhraní v technologii *Kibana* poměrně náročné (a interaktivní graf uvedený v předchozí sekci mezi nestandardní prvky určitě patří). Ze všech těchto důvodů bude aplikace vyvíjena nezávisle na technologii *Kibana* (bude mít vlastní back-end i front-end).

3.3 Případy užití

V této sekci budou popsány případy užití, se kterými se počítá při návrhu aplikace. Tyto případy užití vycházejí z funkčních požadavků z předchozí kapitoly.

UC1. *Vyhledání adresy/adres*

1. Uživatel zadá jednu nebo více adres do formuláře.
2. Tyto adresy budou zobrazeny v grafu.
3. Detail adresy bude zobrazen v informačním panelu.

[4] `https://www.elastic.co/products/kibana`
[5] `https://www.elastic.co/products/elasticsearch`

UC2. *Vytvoření entity ze zadaných adres* – pro potřeby hledání vazeb na ostatní entity (adresy mohly být získany např. z dat bitcoinové peněženky)

1. Uživatel zadá jednu nebo více adres do formuláře zároveň zadá jméno nové entity.
2. Nová entita a její adresy budou zobrazeny v grafu.
3. Od této chvíle je možné jméno entity použít při hledání kdekoliv, kde je požadován identifikátor entity.

UC3. *Vyhledání všech cest mezi dvěma entitami*

1. Uživatel zadá identifikátory dvou entit.
2. Může zvolit některou z omezujících podmínek:
   - směr cesty (odchozí/příchozí transakce nebo obojí)
   - minimální/maximální hodnota transakce
   - časový úsek do kterého má transakce spadat
   - hloubku hledání
3. Nalezené cesty budou zobrazeny v grafu.
4. Detail první zadané entity bude zobrazen v informačním panelu.

UC4. *Vyhledání nejkratší cesty mezi dvěma entitami* – jde o obdobu předchozího případu (postup a omezující podmínky jsou stejné), jen výsledkem hledání bude pouze nejkratší cesta.

UC5. *Vyhledání okolí entity*

1. Uživatel zadá identifikátor entity.
2. Může zvolit některou z omezujících podmínek (viz UC3.
3. Entita a její okolí (transakce a entity) budou zobrazeny v grafu.
4. Detail entity bude zobrazen v informačním panelu.

UC6. *Zobrazení detailu transakce*

1. Uživatel vybere transakci v grafu nebo se proklikne z detailu entity/adresy/clusteru.
2. V informačním panelu se zobrazí informace o transakci uvedené v sekci 3.2.5. Zároveň budou zobrazeny odkazy na entity/adresy účastnící se transakce.

UC7. *Zobrazení detailu adresy*

1. Uživatel vybere adresu v grafu nebo se proklikne z detailu entity/transakce.

2. V informačním panelu se zobrazí informace o adrese uvedené v sekci 3.2.5. Zároveň budou zobrazeny odkazy na transakce, kterých se tato adresa účastnila.

UC8. *Zobrazení detailu entity*

1. Uživatel vybere entitu v grafu nebo se proklikne z detailu adresy/transakce/clusteru.
2. V informačním panelu se zobrazí informace o entitě uvedené v sekci 3.2.5. Zároveň budou zobrazeny odkazy na adresy, které přísluší této entitě a transakce, kterých se entita účastnila.

UC9. *Zobrazení detailu clusteru*

1. Uživatel vybere cluster v grafu nebo se proklikne z detailu entity.
2. V informačním panelu se zobrazí informace o clusteru uvedené v sekci 3.2.5. Zároveň budou zobrazeny odkazy na entity, které přísluší tomuto clusteru a transakce, které mezi sebou tyto entity provedly.

UC10. *Graf – vyhledání okolí entity* – přímo v grafu bude možné spustit hledání okolí entity pomocí kontextové nabídky u každé entity.

UC11. *Graf – zobrazení adres entity* – v grafu bude možné zobrazit adresy konkrétní entity pomocí kontextové nabídky.

UC12. *Graf – zobrazení clusteru entity* – v grafu bude možné zobrazit cluster konkrétní entity pomocí kontextové nabídky.

UC13. *Graf – zobrazení entit a transakcí clusteru* – v grafu bude možné zobrazit entity a transakce konkrétního clusteru pomocí kontextové nabídky.

UC14. *Graf – rozklik hran* `PAYMENT` *na transakce* – v grafu bude možné pomocí kontextové nabídky „rozkliknout“ `PAYMENT` hranu (*entita*→*entita*) na posloupnost *adresa*→*transakce*→*adresa*.

UC15. *Filtrování zobrazených prvků grafu* – uživatel bude mít možnost zobrazit/skrýt libovolný typ prvků v grafu (např. skrýt transakce nebo clustery).

Kapitola 4

Návrh

V této kapitole bude popsán postup návrhu vyhledávací aplikace. Návrh bude vycházet z požadavků a případů užití uvedených v předchozí kapitole a na jejich základě budou zvoleny vhodné technologické prostředky a architektura aplikace. Dále bude podrobněji specifikován způsob a postup obohacení dat z původní distribuované databáze o nové informace a také bude popsán návrh uživatelského rozhraní.

4.1 Technologie – back-end

Navrhovaným programem je webová aplikace nad danou vstupní databází, z technologického hlediska jde tedy zejména o volbu implementačního jazyka aplikace (a příp. použitého frameworku) a dále o způsob práce s databází. Kromě samotné aplikace práce řeší také získávání dodatečných informací o adresách ze vstupní databáze – i to bude součástí této sekce.

4.1.1 Volba programovacího jazyka

V současné době je možné pro vývoj webových aplikací použít většinu populárních programovacích jazyků – mnoho z nich poskytuje podpůrné nástroje a frameworky usnadňující vývoj webových aplikací. Patří mezi ně např. Java (Spring[6]), Javascript (Node.js[7] + jeden z mnoha javascriptových frameworků pro front-end), PHP (Symfony[8], Laravel[9]), Python (Django[10]), Ruby (Ruby on Rails[11] a další.

[6] https://spring.io/
[7] https://nodejs.org/
[8] http://symfony.com
[9] https://laravel.com/
[10] https://www.djangoproject.com/
[11] http://rubyonrails.org/

Navrhovaná aplikace by se jistě dala napsat v každém z nich, já jsem se ale na základě osobních zkušeností rozhodoval zejména mezi dvěma jazyky - *PHP* a *Javou*.

PHP PHP (původně ve významu *Personal Home Page*[11], dnes *PHP: Hypertext Preprocessor*) je skriptovací dynamicky typovaný (od verze 7 volitelně i staticky typovaný) programovací jazyk používaný převážně právě pro tvorbu webových aplikací.

Aplikace v něm napsané většinou fungují tak, že na popředí je webový server (Apache, Nginx), který pro každý příchozí požadavek spustí samostatný skript. To má výhodu v nezávislosti každého požadavku – pokud požadavek *x* způsobí chybu, nijak to přímo neovlivňuje vyřízení požadavku *x + 1*. Na druhou stranu to znamená opakovanou inicializaci všech služeb a připojení k databázi a dalším vnějším prostředkům, což vede k pomalejšímu vyřizování požadavků.

Pro PHP existuje značné množství webových frameworků, nejrozšířenějšími jsou *Symfony* (robustní, modulární framework) a *Laravel* (jednodušší, na rychlost a přístupnost vývoje zaměřený) [12].

Z hlediska práce s Neo4j je podstatné zmínit, že neexistuje oficiální PHP knihovna, pouze komunitní knihovny rozdílné kvality[13].

Java Java je kompilovaný, staticky typovaný, objektově orientovaný jazyk. Má široké možnosti využití – je používaná pro vývoj desktopových, mobilních a webových aplikací nebo i vestavěných systémů. Webové aplikace v ní napsané většinou fungují jako dlouhodobě běžící procesy obsluhující jednotlivé požadavky. To má výhodu ve větší rychlosti odezvy aplikace, ale sdílení prostředků mezi požadavky je přece jen větší než v případě PHP.

Ohledně webových frameworků je situace jednodušší než u PHP – výrazně nejpoužívanější je framework *Spring*[14].

Databáze Neo4j je napsaná v Javě, proto nepřekvapí, že poskytuje pro Javu několik oficiálních knihoven (mimo jiné i integraci do frameworku Spring)[15].

Zvolený jazyk Po zvážení uvedených skutečností jsem se nakonec rozhodl využít pro navrhovanou aplikaci programovací jazyk Java. Programy napsané v Javě jsou (ve srovnání s PHP) většinou robustnější a lépe rozšiřitelné. Roli při rozhodování hrála i existence oficiálních knihoven pro práci s Neo4j.

4.1.2 Volba frameworku

Protože navrhovaným programem je z obecného hlediska celkem standardní webová aplikace, jeví se vhodným krokem použití některého z frameworků pro tyto aplikace.

Jak už bylo řečeno, v ekosystému jazyku Java je nejrozšířenějším frameworkem Spring. Kromě toho, že je nejrozšířenější, poskytuje také snadný způsob práce s Neo4j pomocí projektu *Spring Data Neo4j*[12]. V neposlední řadě už mám s tímto frameworkem zkušenosti z předchozího studia. Proto jsem se rozhodl právě pro tento framework, konkrétně jeho verzi *Spring Web MVC*.

Standardní verze Springu používá konfiguraci pomocí XML souborů (hlavně soubor `web.xml`) a nastavování nového projektu není zrovna přímočaré, Spring ale poskytuje také autokonfigurační nástroj *Spring Boot*[13], který využívá principu „convention over configuration“ a který automatizuje a výrazně zjednodušuje prvotní nastavení projektu[16]. Toto základní nastavení je ale samozřejmě možné později změnit nebo rozšířit (a u navrhované aplikace tomu tak bude). Pro vývoj této aplikace jsem se rozhodl Spring Boot použít.

4.1.3 Práce se vstupní databází

Neo4j a Spring umožňují přistupovat k datům z databáze několika způsoby. Každý z nich má své výhody a nevýhody a hodí se pro jiný typ interakce s databází. Těmito způsoby jsou (každý z nich je zprostředkován konkrétní knihovnou):

- *Neo4j Java Driver* – přístup přes dotazy v CQL (Cypher Query Language)
- *Neo4j-OGM* – objektově-grafové mapování (obdoba ORM pro relační databáze)
- *Spring Data Neo4j* – integrace dat z Neo4j do frameworku Spring
- *Embedded Java API* – přístup v datům přes interní API databáze

V následujících odstavcích budou tyto způsoby popsány podrobněji a zároveň bude zhodnoceno, do jaké míry se hodí pro použití v navrhované aplikaci.

[12] `https://projects.spring.io/spring-data-neo4j/`
[13] `https://projects.spring.io/spring-boot/`

4.1.3.1 Neo4j Java Driver

Neo4j Java Driver je základním způsobem, jak pracovat s Neo4j databází. Umožňuje vytvářet transakce, posílat dotazy v Cypheru a zpracovávat výsledky. Jde o poměrně nízkoúrovňový typ přístupu k datům – vrácené výsledky jsou ve formě jednoduchých datových typů `Node`, `Relationship` apod. a pokud je potřeba výsledky mapovat na doménové objekty, toto mapování si musí programátor vytvořit sám. Použití může vypadat např. takto:

```
Driver driver = GraphDatabase.driver("bolt://localhost:7687",
                AuthTokens.basic( "neo4j", "neo4j"));
Session session = driver.session();

StatementResult result = session.run(
        "MATCH (a:Address) WHERE a.address = {address} " +
        "RETURN *",
        parameters("address", "113Uax5WXDehoHoRfnaR8XtsjsQavsSN9v"));

session.close();
driver.close();
```

4.1.3.2 Neo4j-OGM

Neo4j-OGM je inspirováno ORM frameworky (v Javě např. Hibernate[14], pro PHP Doctrine[15]). Převážně pomocí anotací mapuje uzly a hrany grafu na doménové entity dané aplikace. Mapování entity `Address` by mohlo vypadat např. takto:

```
@NodeEntity
public class Address {

        @GraphId
        private Long id;
        private String address;

        @Relationship(type = "BELONGS_TO", direction = "OUTGOING")
        private Entity entity;
}
```

Výhodou je právě snadné mapování mezi grafem a entitami (za cenu mírného zpomalení aplikace). Pokud je ale potřeba sestavit složitější dotaz nebo (hlavně) dotaz, jehož návratovou hodnotou je něco jiného než namapované entity (např. výsledek nějaké agregační funkce) , je stejně třeba sáhnout po

[14] `http://hibernate.org/`

[15] `http://www.doctrine-project.org/`

vlastním mapování. Existuje také názor, že ORM/OGM knihovny vedou člověka k mimoděčnému „ohýbání“ návrhu aplikace tak, aby bylo možné využít automatického mapování a nebylo třeba psát specializované dotazy.

4.1.3.3 Spring data Neo4j

Spring data staví na předchozí OGM knihovně a přidávají prvky pro integraci do frameworku Spring, hlavně tzv. `Repository` třídy (resp. interface), které sdružují databázové dotazy pro konkrétní entitu. Takto by mohl vypadat `Repository` interface pro entitu `Address`:

```
interface AddressRepository extends GraphRepository<Address> {

        // derived finder
        Address findByAddress(String address);

        @Query("MATCH (e:Entity)<-[b:BELONGS_TO]-(address)
                WHERE id(address)={address} return e")
        Entity getEntity(@Param("address") Address address);
}
```

Jde o interface a nikoliv o třídu – *Spring data* na pozadí vytvoří implementující třídu. Zajímavý je způsob odvození implementace metody `findByAddress` – díky dodrženým konvencím v pojmenování metody knihovna *Spring data* pozná, že má jít o hledání podle atributu `address`. Takovýto druh „magie“ bych čekal spíš v javascriptových nebo PHP knihovnách a v Javě mě poměrně překvapil.

4.1.3.4 Embedded Java API

Databáze Neo4j je napsaná v Javě a umožňuje využívat aplikační rozhraní používané interně samotnou databází. Toho se primárně využívá pro tvorbu různých pluginů a obecně pro rozšiřování funkcí databáze, ale je možné tímto způsobem i databázi procházet a vyhledávat v ní:

```
GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
GraphDatabaseService db = dbFactory.newEmbeddedDatabase("\data\Neo4jDB.db");
Transaction tx = db.beginTx();

ResourceIterator<Node> addresses = db.findNodes(Label.label("Address"));
while (addresses.hasNext()) {
        Node address = addresses.next();
        System.out.print(address.getProperty("address");
}
tx.success()
```

Jak je vidět, jde o přístup na nižší úrovni než všechny předchozí a psaní složitějších vyhledávacích dotazů by bylo poměrně náročné. Kromě toho je potřeba zmínit, že oproti ostatním popsaným způsobům jde o „embedded“ přístup – program nekomunikuje s oddělenou databází, ale pracuje přímo s jejími datovými soubory. Kvůli tomu musí být databáze vypnutá a tudíž není možné tento způsob kombinovat s ostatními popsanými způsoby přístupu.

4.1.3.5 Zvolený způsob přístupu k databázi

Pro potřeby navrhované aplikace se zjevně příliš nehodí *Embedded Java API* – jde o příliš nízkoúrovňový přístup a nemožnost ho kombinovat s ostatními přístupy by byla značně limitující.

V návrhu se kloním k použití knihovny *Spring data* v kombinaci s přístupem přes *Neo4j Java Driver*. První způsob (mapování na doménové entity) se hodí pro tvorbu grafu provázaných entit, který bude poté zobrazen uživateli na frontendu pro základní orientaci ve vazbách dat (viz. 3.2.7). Druhý způsob (pokládání Cypher dotazů a přímá práce s výsledky) bude naopak použit pro získávání sumarizačních a agregačních dat, pokud si uživatel vyžádá detailnější informace o konkrétních entitách, adresách apod.

4.2 Technologie – front-end

Na straně front-endu je potřeba vyřešit dvě základní otázky. Za prvé – jakými prostředky implementovat samotné rozhraní? A za druhé – jak implementovat navrhovaný interaktivní graf? Odpověďmi na tyto otázky se bude zabývat tato sekce.

4.2.1 Rozhraní aplikace

Základní rozhraní aplikace je ve své podstatě jedna stránka rozdělená na 3 sekce – vyhledávací panel, interaktivní graf a panel s informacemi o nalezených prvcích grafu. Základní rozvržení bude proto implementováno pomocí JSTL šablony (standardní šablonovací komponenta frameworku Spring).

Obsluha formulářů a dynamické načítání a zobrazování dat budou řešeny pomocí javascriptu a knihovny *jQuery*[16]. Domnívám se, že pro potřeby aplikace není třeba (při kvalitním návrhu) využívat nejnovější front-endové knihovny a frameworky jako *React.js* nebo *Vue.js*. Zároveň přiznávám, že s danými knihovnami nemám osobní zkušenost, což má na mé rozhodnutí je nepoužít určitě vliv.

[16] `https://jquery.com/`

Stylování aplikace bude řešeno kombinací vlastních CSS stylů s některými prvky frameworku *Bootstrap*[17].

4.2.2 Interaktivní graf

Navrhovaný graf má sloužit k zobrazování vztahů mezi entitami, adresami, transakcemi a dalšími prvky vstupní databáze. Má umožňovat iterativně prozkoumávat prostor grafu zejména ve vazbě na entity (skupiny adres) – hledat okolí dané entity, zobrazovat její adresy a jejich vazby na transakce apod. Z technologického hlediska by měl umožňovat:

- *dynamicky přidávat a odebírat uzly/hrany* – při hledání okolí uzlu (které ještě není v grafu) bude třeba přidávat nové uzly a hrany, při skrývání určitých typů prvků bude zase třeba uzly a hrany odebírat.
- *automatické rozvržení grafu* – po načtení dat grafu je třeba ho zobrazit v podobě, která je srozumitelná a přehledná.
- *úpravy rozvržení grafu uživatelem* – uživatel musí mít možnost upravit polohu jednotlivých uzlů grafu, např. pro vizuální oddělení konkrétních vazeb/uzlů od zbytku (obecně pro libovolné vizuální organizování dat grafu)
- *stylování uzlů a hran* – kvůli přehlednosti je třeba jasně vizuálně odlišit jednotlivé typy uzlů a hran
- *kontextová nabídka* – graf musí umožňovat nad jeho jednotlivými uzly/hranami provádět různé akce (nejspíš ve formě kontextové nabídky) – např. nabídka u uzlu `Entita` by měla umožňovat zobrazit k entitě přiřazené adresy, zobrazit její cluster nebo hledat okolní entity

Na základě rešerše existujících řešení splňujících uvedené požadavky jsem k podrobnějšímu zkoumání vybral 4 javascriptové knihovny: *KeyLines*, *D3.js*, *Sigma.js* a *Cytoscape.js*.

4.2.2.1 KeyLines

KeyLines[18] je knihovna pro vizualizaci grafově orientovaných dat. Obsahuje konektor přímo pro práci s Neo4j a splňuje všechny požadavky uvedené výše. Umožňuje také shlukovat jednotlivé uzly do jednoho nebo vybírat z několika typů automatického rozvržení[17]. Její nevýhodou je, že je placená (a není levná).

[17] `http://getbootstrap.com/`
[18] `https://cambridge-intelligence.com/keylines/`

4.2.2.2 D3.js

D3.js[19] je obecná vizualizační knihovna. Také splňuje všechny výše uvedené požadavky a je určitě nejuniverzálnější z uvažovaných řešení. To je ale zároveň (pro účely navrhované aplikace) její největší problém – jelikož nejde přímo o knihovnu pro vizualizaci grafů, bylo by třeba se při implementaci pohybovat na nižší úrovni abstrakce, což by mohlo značně prodlužovat dobu vývoje. Na druhou stranu by bylo možné implementovat všechny požadované vlastnosti přesně.

4.2.2.3 Sigma.js

Sigma.js[20] je asi nejrozšířenější JS knihovna na kreslení grafů. Umožňuje přidávání a odebírání prvků grafu, několik druhů automatického rozvržení i manuální úpravy rozvržení. Možnosti úpravy vzhledu grafu nejsou příliš rozsáhlé, ale pro účely navrhované aplikace by nejspíš stačily. Přímá možnost zobrazení kontextové nabídky zde není – bylo by nejspíš potřeba využít některé z knihoven sloužících pro tento účel (např. *jQuery contextMenu*[21].

4.2.2.4 Cytoscape.js

Cytoscape.js[22] je knihovna pro vizualizaci a analýzu grafů. Splňuje všechny požadavky uvedené výše. Možnosti stylování grafu jsou výrazně širší než u knihovny *sigma.js* – je možné používat jakési pseudo-CSS selektory. Práce s událostni grafu je také jednodušší (stejné selektory je možné používat i pro filtraci eventů), knihovna obsahuje i plugin pro práci s kontextovou nabídkou. Kromě požadovaných vlastností umí také některé grafové algoritmy (např. DFS, Dijsktra a další).

4.2.2.5 Zvolená knihovna

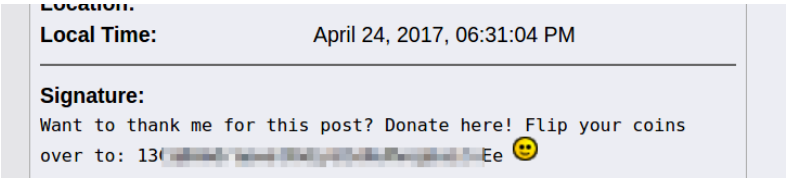
Po zvážení všech uvedených možností jsem se rozhodl použít knihovnu *cytoscape.js* – jde o knihovnu přímo určenou pro vizualizaci grafů a zároveň poskytuje mnoho možností úprav a rozšíření funkcí a snadné stylování grafu. Jde tak o dobrý kompromis robustnosti a zároveň jednoduchosti vývoje.

4.2.3 Identifikace adres a entit pomocí externích dat

Součástí této práce má být i získání a import dat z fóra bitcointalk.org, konkrétně získání adresy obsažené v podpisu (signature) některých uživatelů tohoto fóra. Jak bylo uvedeno v sekci 3.2.6, uživatelé často do svého podpisu

[19] `https://d3js.org/`
[20] `http://sigmajs.org/`
[21] `https://github.com/swisnl/jQuery-contextMenu`
[22] `http://js.cytoscape.org/`

přidají bitcoinovou adresu v naději, že jim čtenáři jejich příspěvků pošlou na tuto adresu peníze (viz obrázek 4.1).



Obrázek 4.1: Ukázka podpisu z internetového fóra

Profily uživatelů jsou veřejně přístupné, pro každého uživatele jde tedy o posloupnost těchto kroků:

1. Načíst stránku s profilem.
2. Extrahovat podpis uživatele.
3. Pokud podpis obsahuje adresu, uložit tuto informaci ve strojově zpracovatelném formátu.

4.2.3.1 Technologie

Extrahování a import externích dat je nezávislé na samotné aplikaci, je proto možné použít rozdílné technologie. Jelikož mám z minulosti zkušenost s frameworkem *Scrapy*[23], rozhodl jsem se použít pravě tento nástroj. *Scrapy* je framework pro extrahování informací z webových stránek napsaný v jazyce Python. Je možné pomocí něj snadno definovat, jaká data a odkud se mají stahovat a v jakém formátu se mají uložit.

Data budou extrahována pomocí tohoto nástroje ve formátu CSV. Následně budou importována do vstupní databáze pomocí Cypher příkazu `LOAD CSV`.

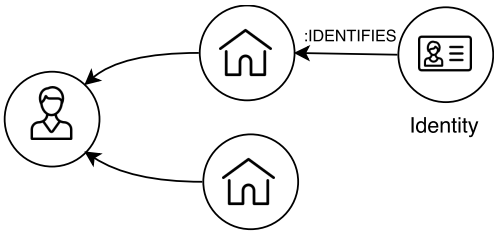
4.2.3.2 Datové struktury

Návrh počítá s budoucím rozšířením množství importovaných informací zejména co do počtu jejich zdrojů, je třeba s tím tedy počítat i při návrhu datových struktur pro jejich uložení.

Každá adresa ve vstupní databázi může být identifikována pomocí 0 - n zdrojů, navrhuji tedy vytvořit typ (label) uzlu určený pro uložení informace z konkrétního zdroje a odpovídající adresu k němu přiřadit pomocí identifikační hrany. V návrhu pro tyto účely používám typ uzlu `Identity` a typ hrany `IDENTIFIES` (viz obr. 4.2). Entita je v tomto modelu identifikována identitami

[23] `https://scrapy.org/`

svých adres (entita samotná není součástí původní distribuované databáze a není proto možné k ní přímo přiřadit externí informace).



Obrázek 4.2: Identita – vztah s adresou a entitou

Každý uzel `Identity` bude mít následující atributy:

- *adresa* – bitcoinová adresa, kterou tento uzel identifikuje
- *označení zdroje* – identifikace zdroje informace (např. „bitcointalk.org“)
- *url* – přesná url adresa, na které se nachází tato konkrétní informace
- *další atributy podle zdroje* – každý zdroj může obsahovat jiné informace – z *bitcointalk* fóra se dá např. získat *podpis*, *uživatelské jméno* a *id uživatele*.

4.3 Architektura

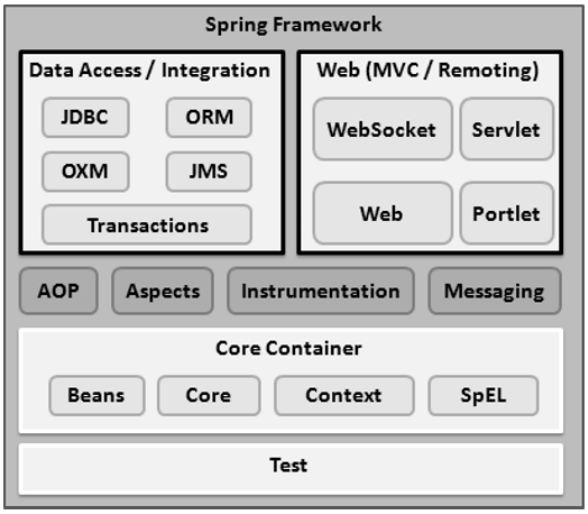
V této sekci bude popsán návrh architektury aplikace – způsob spolupráce tříd, struktura balíčků apod. Tato architektura je samozřejmě silně ovlivněna zvolenými technologiemi, které poskytují určité prostředky k řešení problémů a počítají s nějakým přístupem k návrhu. Proto v první části bude stručně popsáno fungování frameworku *Spring* (při použití konfiguračního nástroje *Spring Boot*).

4.3.1 Fungování frameworku Spring

Spring je framework pro vývoj aplikací v programovacím jazyku Java – není omezen jen na vývoj webových aplikací, i když pro tento účel je používán nejčastěji. Je možné ho použít pro vývoj širokého spektra zejména enterprise aplikací – Spring si klade za cíl poskytovat „kostru“ aplikace a nástroje pro řešení problémů typických pro většinu aplikací, zejména:

- *Dependency injection, IoC container* – správa závislostí jednotlivých tříd, snadná změna závislé třídy, testovatelnost.
- *Modularita* – jednotlivé části poskytované funkčnosti je možné používat nezávisle (podle potřeby dané aplikace), viz obrázek. 4.3

- *Spring MVC* – kompletní MVC (Model-View-Controller) framework pro tvorbu webových aplikací.
- *Podpora JDBC, JPA a dalších technologií pro persistenci dat*
- *Spring Data* – sjednocení způsobů přístupu k různým datovým zdrojům (relační / objektové / grafové databáze apod.).
- *Konfigurace pomocí anotací* – velkou část konfigurace je možné přesunout z XML a podobných souborů do anotací v kódu přímo u konfigurovaných prvků. Tato vlastnost je je ještě zvýrazněna při použití konfigurátoru *Spring Boot*.



Obrázek 4.3: Architektura frameworku Spring (zdroj: [18])

IoC container *Spring IoC container* (nebo také Spring container / Service container) řeší závislosti tříd a v podstatě „skládá“ dohromady celou aplikaci z tzv. *Spring beans*, což jsou jednotlivé instance tříd, pro které je definováno (pomocí anotací nebo jinak), že jsou spravovány IoC containerem. Tyto třídy mohou být pomocí IoC containeru jednoduše skládány do rozsáhlých funkčních celků. Uvedu příklad:

```
@Service
public class Neo4jService {
        ...
}

@Repository
```

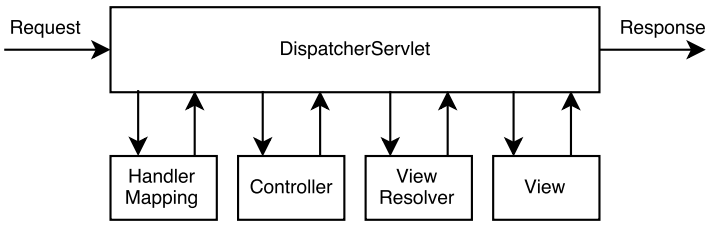
```
public class EntityRepository {

        @Autowired
        protected Neo4jService neo4jService;
        ...
}
```

V tomto příkladu je nejprve vytvořen *Spring bean* `Neo4jService` pomocí anotace `@Service` a poté je jako závislost použit ve třídě `EntityRepository`, která je sama *Spring beanem* (díky anotaci `@Repository`) a může být tedy použita stejným způsobem v dalších třídách. Důležité je, že není třeba řešit, jak se závislá služba `Neo4jService` do `EntityRepository` dostane (není třeba ji nikde inicializovat) – o to se postará IoC container.

Spring MVC Spring obsahuje jako svou součást MVC framework. MVC je návrhový vzor pro tvorbu aplikací, který rozděluje aplikaci do tří komponent, které jsou na sobě pokud možno zcela nezávislé:

- *Model* – data a business logika s kterými aplikace pracuje.
- *View* – reprezentace dat pro prezentaci uživateli.
- *Controller* – řadič, který přijímá požadavky od uživatele, předává je *modelu* a informuje *view* o změnách.

Stejně jako většina ostatních MVC frameworků používá *Spring MVC* návrhový vzor spíše volně. Jeho základním prvkem je tzv. *DispatcherServlet*, který přijímá všechny požadavky a vrací odpovědi. Jednotlivé fáze zpracování požadavku deleguje na další části frameworku – *Handler Mapping* rozhoduje o tom, který *Controller* se použije, ten pak požadavek předá business vrstvě aplikace, připraví data a předá zpět jméno *view*, který se má použít. *DispatcherServlet* pak pomocí komponenty *View Resolver* daný *view* vyhledá a předá mu data. Výsledná reprezentace je pak vrácena jako odpověď na požadavek (celý proces je znázorněn na obrázku 4.4).



Obrázek 4.4: Spring MVC DispatcherServlet

Spring Data *Spring data* umožňuje (víceméně) jednotný způsob práce s persistentními daty při použití různých úložišť. Pracuje na principu mapování

objektového modelu na jiný model (relační, grafový, ...). Ukázka použití pro Neo4j byla uvedena v sekci 4.1.3.3.

4.3.2 Architektura samotné aplikace

Tato sekce popisuje architekturu navrhované aplikace. Ta vychází z funkčních i nefunkčních požadavků a z použitých technologií.

Základní vrstvy aplikace jsou reprezentovány třemi typy tříd:

- *Controller* – přijímá požadavky od uživatele a překládá je do business funkcí.
- *Service* – třída obstarávající business logiku aplikace.
- *Repository* – řeší čtení a zápis konkrétních dat z/do persistentního úložiště (databáze Neo4j).

Kromě nich jsou základním prvkem aplikace *modelové třídy*, reprezentující data řešené domény, zejména uzly a hrany grafu a také souhrnné informace o některých doménových objektech (např. detail entity obsahující zároveň informace o jejích transakcích a adresách).

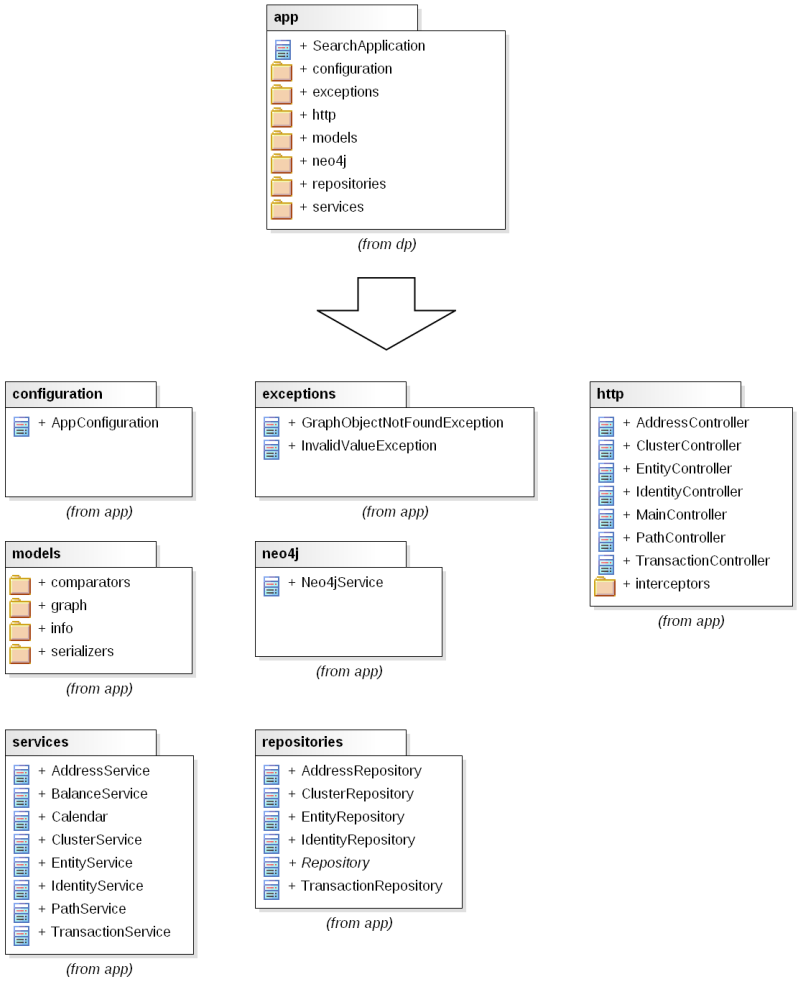
Základní rozvržení balíčků aplikace je vidět na obrázku 4.5 – v dalších podsekcích popíšu funkce jednotlivých balíčků a uvedu princip jejich spolupráce při vyřizování příchozího požadavku.

4.3.2.1 Základní rozdělení na balíčky

Hlavní třídou aplikace je `BlockchainApplication`, ve které se inicializuje Spring framework, jeho IoC container a DispatcherServlet pro vyřizování příchozích požadavků. Toto je jediná třída v rootovském balíčku aplikace. Další balíčky jsou popsány níže.

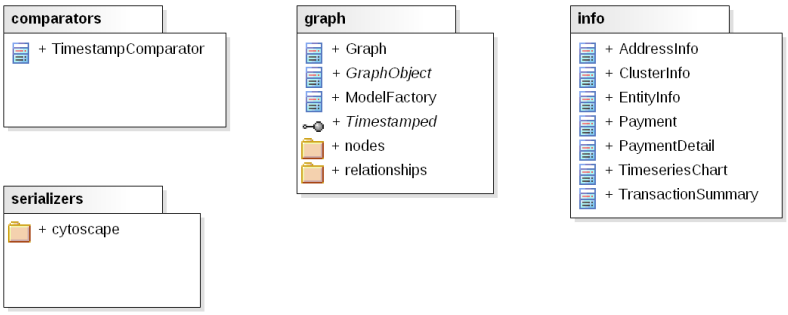
Http V tomto balíčku se nacházejí všechny controllery aplikace. Každý controller má na starosti požadavky týkající se konkrétního doménového objektu (tzn. existuje např. `EntityController`, `AddressController` atd.) – toto členění zachovávají i další balíčky aplikace. Kromě controllerů je obsahem ještě balíček tzv. *interceptorů* – jde o služby, které zpracovávají příchozí požadavek ještě před přijetím v controlleru (je např. možné k požadavku přidat nějaké informace nebo ho rovnou odmítnout).

Models Balíček Models sdružuje všechny modelové třídy. Obsahuje další čtyři balíčky:



Obrázek 4.5: Základní organizace balíčků

- *Graph* – obsahuje modely potřebné pro vykreslení a práci s grafem. Jde jednak o model grafu samotného a pak modely všech možných uzlů (`Transaction`, `Address` atd.) a hran (`BELONGS_TO`, `PAYMENT` atd.) grafu.
- *Info* – obsahuje třídy reprezentující souhrnné informace (detaily) vybraných objektů (např. `EntityInfo` nebo `ClusterInfo`).
- *Serializers* – třídy sloužící pro serializaci grafových modelů do reprezentace používané na frontendu pro kreslení grafu (každá třída má svůj serializér).
- *Comparators* – obsahuje třídy pro porovnávání/řazení modelů (určitě bude třeba např. řazení podle časové značky).



Obrázek 4.6: Organizace modelových tříd

Services V tomto balíčku se nacházejí všechny *service* třídy. Ty mají na starosti business logiku aplikace, např. vytváření souhrnných informací o entitách, výpočet zůstatku adresy apod.

Repositories Zde jsou sdruženy všechny *repository* třídy, řešící čtení a zápis dat do persistentního úložiště.

Neo4j Tento balíček obsahuje třídu starající se o správu spojení do databáze Neo4j (tato třída je pak využívána *repository* třídami).

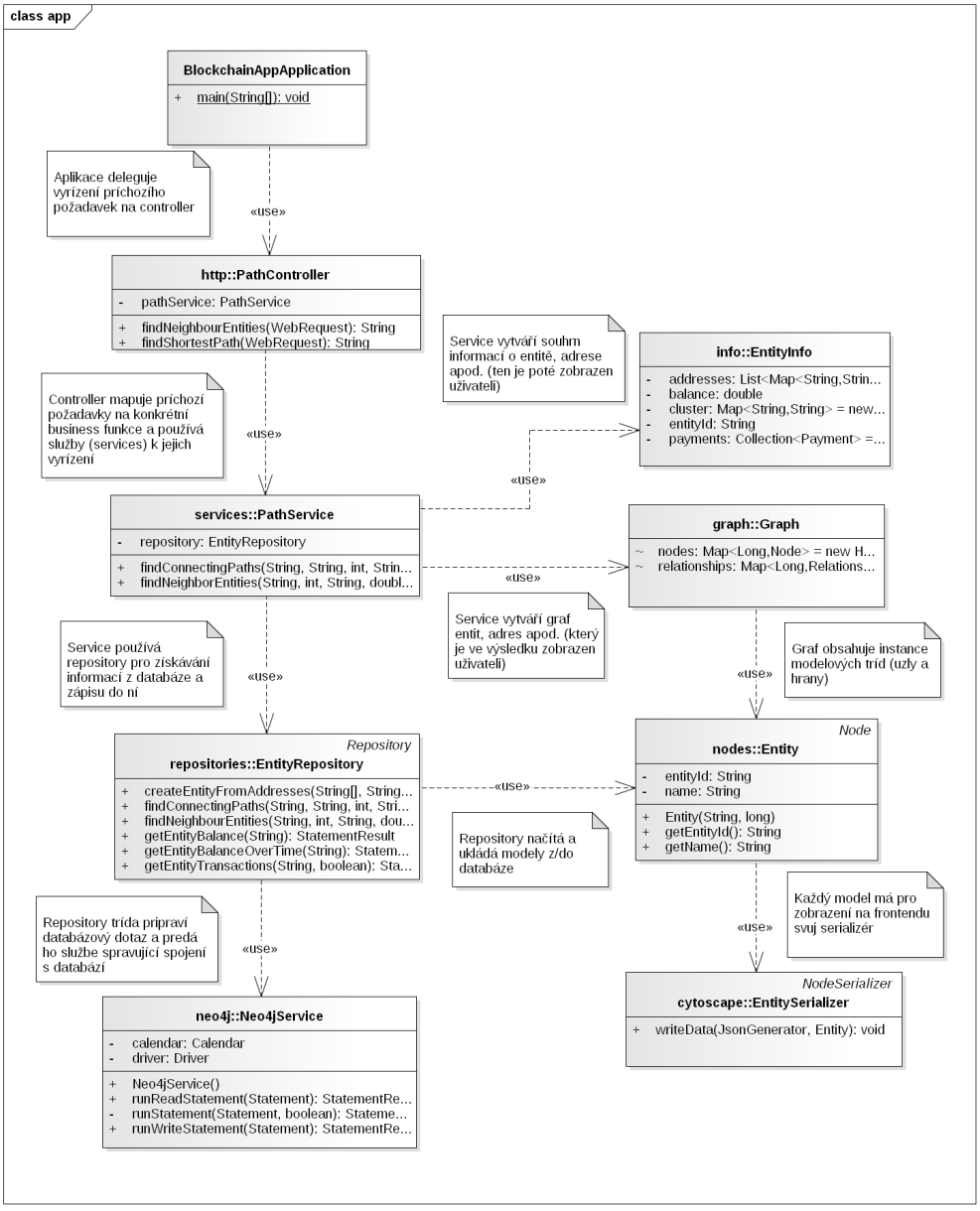
Configuration V tomto balíčku se nachází konfigurační třída aplikace – Spring Boot používá místo XML konfiguračních souborů soubor tzv. *sane defaults* – nastavení, která fungují pro většinu aplikací, ale pokud je třeba nějaké nastavení změnit, používá se k tomu konfiguračních tříd (samozřejmě je možné využít i klasickou XML konfiguraci).

Exceptions Tento balíček sdružuje výjimky (exceptions) používané aplikací.

4.3.3 Spolupráce tříd

Jednotlivé typy tříd (controllery, služby, repository třídy, ...) tvoří vrstvy aplikace, které spolupracují za zpracování příchozího požadavku. Základním kamenem aplikace jsou doménové objekty – proto jsou (jak je zjevné z popisu balíčků) controllery, služby apod. často navázané na konkrétní doménový objekt/model (např. pro model `Entity` existuje `EntityController` a `EntityService`). Existují ale i controllery a služby, které nepřísluší přímo konkrétnímu modelu nebo používají modelů více.

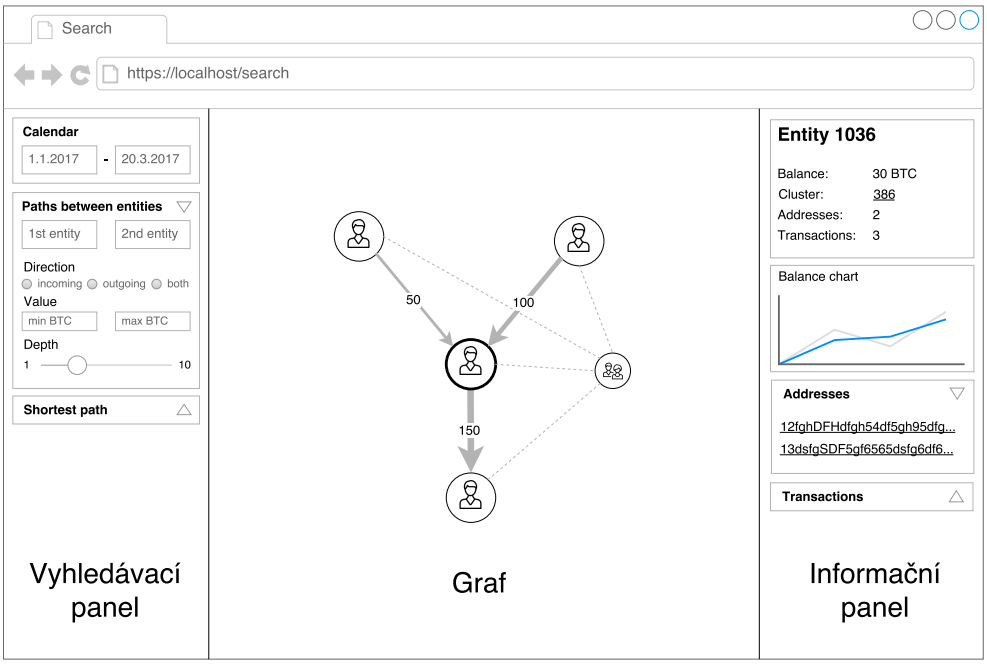
Na obrázku 4.7 je na příkladu výpočtu cest (plateb) mezi entitami ukázána spolupráce jednotlivých tříd s vysvětlení jejich vazeb.



Obrázek 4.7: Spolupráce tříd

4.4 Uživatelské rozhraní

V této sekci je popsán návrh uživatelského rozhraní. Ten v základu vychází z případů užití a rozvíjí základní členění popsané v sekci 3.2.7. Toto základní členění je zobrazeno na obrázku 4.8 – *vyhledávací panel* je vlevo, největší prostor uprostřed zabírá *interaktivní graf* a vpravo je *informační panel*.



Obrázek 4.8: Přehled uživatelského rozhraní

4.4.0.1 Vyhledávací panel

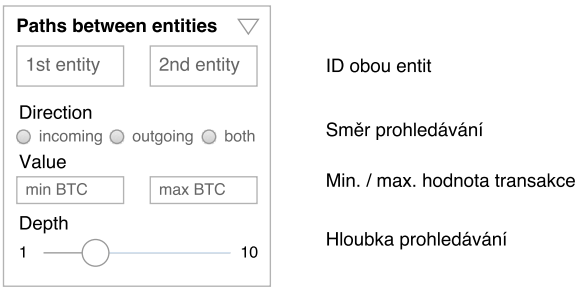
Tento panel slouží k zadávání vyhledávacích kritérií pro prvotní hledání (v další fázi je možné toto vyhledávání rozvíjet pomocí „proklikávání se“ interaktivním grafem nebo pomocí odkazů v informačním panelu). Obsahuje *kalendář* s možností výběru počátečního a koncového data (všechny vyhledávací dotazy na hledání cest / okolí entit budou poté omezeny na transakce v tomto období) a dále *vyhledávací formuláře*. Tyto formuláře umožňují hledat:

- adresy a k nim náležející entity
- všechny cesty mezi entitami
- nejkratší cestu mezi entitami
- okolí entity – entity spojené transakčními hranami s vybranou entitou (do zadané hloubky)

Při hledání cesty je možné zadat doplňující kritéria (směr, hloubku a další),

která jsou popsaná v sekci 3.3 (konkrétně v případu užití UC3.) a jejich rozvržení v návrhu uživatelského rozhraní je vidět na obrázku 4.9.

Kromě vyhledávání je v tomto panelu také možné vytvořit ze zadaných adres novou entitu – toho se dá využít ve chvíli, kdy uživatel disponuje informací (ze zdrojů mimo blockchain) o tom, že konkrétní množina adres patří jedné osobě.



Obrázek 4.9: Vyhledávací formulář

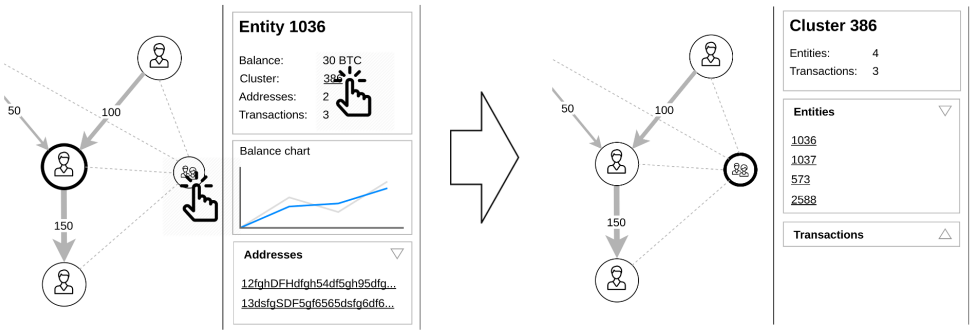
4.4.0.2 Interaktivní graf

Graf má v základu dvě funkce: zaprvé slouží k zobrazení prvků ze vstupní databáze a důležitých vazeb mezi nimi, a zadruhé umožňuje interakci s těmito prvky. Je možné tyto prvky v grafu přesouvat, skrývat/zobrazovat typy prvků a u důležitých prvků grafu (konkrétně clusterů, entit, adres a transakcí) je možné zobrazit jejich detail. Tento detail je možné zobrazit také pomocí odkazu v souvisejícím detailu jiného prvku (např. z detailu entity se dá „prokliknout“ a detail jejího clusteru). Oba způsoby jsou vidět na obrázku 4.10.

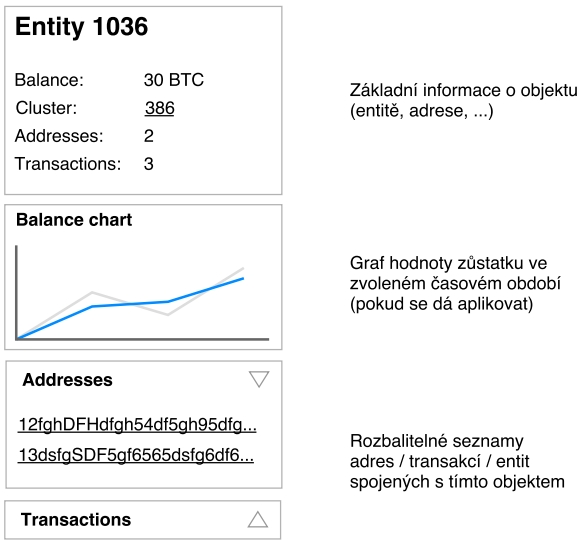
Dále je možné pomocí s některými prvky grafu interagovat pomocí kontextové nabídky, např. hledat okolí dané entity (všechny možnosti jsou popsány v Případech užití, konkrétně UC10. až UC14.).

4.4.0.3 Informační panel

Tento panel slouží k zobrazení detailu entity, adresy apod. Jeho základní členění je vidět na obrázku 4.11. Důležitou vlastností informačního panelu je provázanost detailu entity, adresy apod. se všemi souvisejícími prvky, tzn. z detailu entity je možné se „prokliknout“ na detail její libovolné adresy a z této adresy zase na její libovolnou transakci (nebo zpět na entitu). To umožňuje plynulejší prohledávání dat blockchainu, než kdyby bylo třeba používat pouze vyhledávací formuláře.



Obrázek 4.10: Navigace v grafu / info panelu



Obrázek 4.11: Informační panel

Kapitola 5

Implementace

Předchozí kapitoly se zabývaly popisem vstupních dat, požadavků na aplikaci (kapitola Analýza), výběrem technologií, návrhem architektury a uživatelského rozhraní aplikace (kapitola Návrh). V této kapitole bude řešena samotná implementace. V první části budou popsány změny proti návrhu, které vyplynuly z detailnějšího pohledu na problematiku v průběhu implementace. Dále bude podrobněji popsána implementace některých prvků, a nakonec bude uveden diagram nasazení aplikace.

5.1 Změny proti návrhu

V této sekci budou uvedeny některé podstatnější změny proti návrhu. Tyto změny nemění funkce aplikace, jde spíše o způsob jejich implementace.

5.1.1 Nepoužití knihovny Spring data Neo4j

V návrhu byly popsáno několik způsobů přístupu k datům v databázi Neo4j (sekce 4.1.3) a nakonec byla zvolena kombinace knihoven *Spring data Neo4j* (přístup pomocí objektově-grafového mapování) a *Neo4j Java Driver* (nízkoúrovňový přístup přes Cypher dotazy).

V průběhu implementace se ale ukázalo, že načítání dat pomocí objektově-grafového mapování není pro potřeby aplikace příliš vhodné. Aplikace potřebuje data v zásadě pro dva účely – zaprvé pro zobrazení grafu entit, adres apod. a vazeb mezi nimi, a zadruhé pro zobrazení souhrnných informací o konkrétním prvku grafu (detail entity, clusteru, ...).

V prvním případě je graf výsledkem hledání cesty/cest z jednoho uzlu grafu do dalších přes proměnlivé množství jiných uzlů a hran různých typů. Aby toto fungovalo v prostředí objektově-relačního mapování, je třeba hodně manuální práce s výsledky dotazů, což jde proti principu automatického mapování a

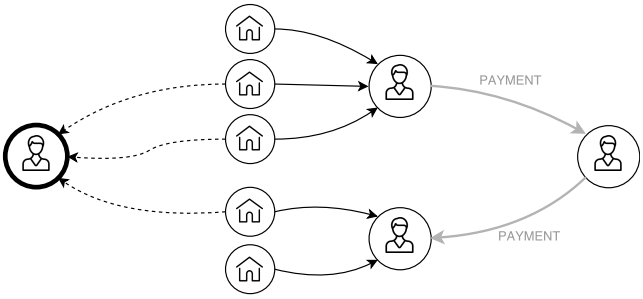
ve výsledku to není o mnoho jednodušší než přímá práce s výsledky Cypher dotazů.

Rozhodl jsem se proto použít pro tento účel knihovnu *Neo4j Java Driver*. Tato knihovna je použita i pro potřeby získání souhrnných informací o konkrétním prvku grafu (zde už v souladu s návrhem).

5.1.2 Manuálně přidané entity

V aplikace je možné zadat skupinu adres a vytvořit z nich novou entitu. Tato entita pak funguje stejně jako ostatní existující entity – je možné ji použít při hledání cest, okolí apod. V návrhu se počítalo s tím, že i implementačně půjde o stejný typ uzlu jako ostatní entity. To se při implementaci ukázalo jako ne příliš vhodné řešení.

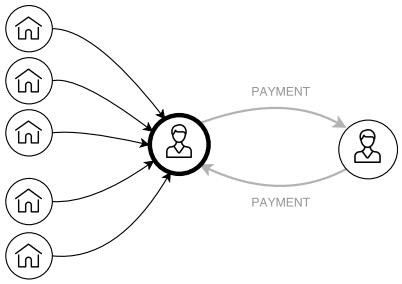
Na obrázku 5.1 je ukázka situace při přidání nové entity. Každá z adres nové entity už patří některé existující entitě. Je možné, že všechny patří právě jedné entitě, ale může se také stát, že každá z adres bude patřit jiné entitě. Otázka je, jak s v takové situaci zachovat, protože adresa logicky nemůže patřit více entitám.



Obrázek 5.1: Situace při manuálním vytvoření entity

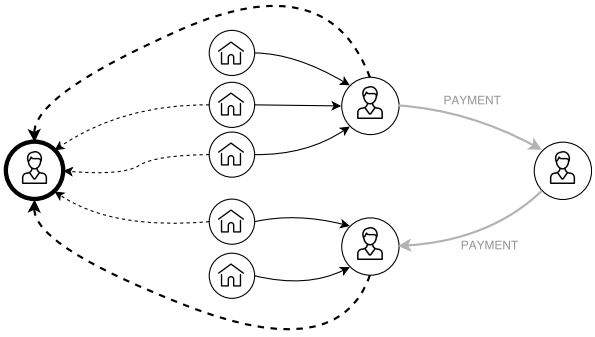
Je možné všechny entity, jejichž adresy jsou v množině zadané uživatelem sloučit do jedné (viz. obrázek 5.2). V tom případě je třeba i přepočítat (sloučit, příp. vypustit) všechny hrany vedoucí do původních entit (`PAYMENT` , `SENT`, `RECEIVED` a další). Také se tím ztratí informace o tom, které adresy byly nové entitě přiřazeny manuálně a které v důsledku sloučení – vlastně se ztratí celá informace o původních entitách. Bylo by tudíž poměrně náročné tuto manuální entitu případně smazat a obnovit původní (nesloučené) entity.

Rozhodl jsem se proto modelovat tuto situaci jinak – manuálně vytvořená entita je je uzel jiného typu (`MANUAL_ENTITY`) než standardní (vygenerované) entity a je spojena hranami `MANUAL_BELONGS_TO` s přiřazenými adresami a hranami `ENTITY_BELONGS_TO` s entitami, kterým tyto adresy patří (princip



Obrázek 5.2: Sloučení entit

je zobrazen na obrázku 5.3). Místo přímého sloučení je tedy použito sloučení pomocí vazby. Tento způsob poněkud zesložiťuje Cypher dotazy na okolí a cesty mezi entitami (je třeba počítat s tím, že zadaná entita může být typu `MANUAL_ENTITY`), na druhou stranu se díky tomu neztrácí žádná informace a není třeba přepočítávat vazby mezi existujícími entitami.



Obrázek 5.3: Sloučení entit pomocí vazby

5.2 Vybrané části implementace

5.2.1 Specifika dotazovacího jazyka Cypher

Dotazovací jazyk Cypher je specialitou databáze Neo4j. Umožňuje vytvářet grafové dotazy, které jsou vizuálně velmi přehledné (podle dokumentace mají připomínat ascii-art [19]). Některé jeho vlastnosti ale překvapí – někdy to souvisí s odlišností oproti SQL, jindy s faktem, že se vyhledává nad grafem.

5.2.1.1 Graf vs. tabulka

Základním rozporem, se kterým je třeba se při práci s Cypherem (a obecně s Neo4j) vyrovnat, je fakt, že ačkoliv se pracuje s grafem, výsledky dotazů

jsou ve formě tabulky – např. při tomto jednoduchém hledání:

```
MATCH (e1:Entity {id:1})-[p:PAYMENT*0..2]-(e2:Entity) return *
```

(hledá se okolí entity přes 0-2 hrany `PAYMENT`) není výsledkem množina uzlů `Entity` a hran `PAYMENT` , ale všechny *cesty* z počáteční entity přes max. 2 hrany daného typu (každá cesta na jednom řádku). Tímto nechci říct, že by to mělo být jinak, jen je potřeba na to pamatovat – i při malém množství uzlů je možné ve výsledcích získat řádově větší počet řádků, pokud mezi danými uzly existuje velký počet hran.

5.2.1.2 Hrana vs. kolekce hran

U dotazu z předchozí sekce (`MATCH (e1:Entity id:1)-[p:PAYMENT*0..2] -(e2:Entity) return *`) může hodnota `p` v konkrétním výsledku být buď prázdná, nebo obsahuje jednu hranu, nebo *kolekci* 2 hran. Problém nastává, pokud je třeba aplikovat na tyto hrany nějakou podmínku. Např. podmínka:

```
WHERE p.timestamp > 10000000
```

bude fungovat pro jednu hranu `p`, ale selže, pokud je aplikována na kolekci hran. Naopak podmínka:

```
WHERE all(o IN p WHERE o.timestamp > 10000000)
```

funguje pro kolekci hran, ale selže pro jednu hranu. V případech, kdy není jisté, jestli výsledkem bude jedna hrana nebo jejich kolekce, je třeba toto obcházet pomocí konstrukce `WITH` a funkce `collect` preventivním „zabalením“ výsledku do kolekce. Funkční dotaz potom vypadá takto:

```
MATCH (e1:Entity {id:1})-[p:PAYMENT*0..2]-(e2:Entity)
WITH collect(p) AS ps, e1, p, e2
WHERE all(p IN ps WHERE p.timestamp > 10000000)
RETURN e1, p, e2
```

5.2.1.3 Automatické grupování

V předchozí sekci zmíněná konstrukce `WITH collect(...)` má ještě jednu důležitou vlastnost: výsledek vnitřní funkce `collect` je grupován podle toho, jaké další proměnné se vyskytnou v konstrukci `WITH` – pokud by druhý řádek předchozího dotazu byl pouze `WITH collect(p) AS ps` (bez `e1`, `e2`), výsledkem by byla kolekce všech hran `p` bez ohledu na to, ke které entitě patří.

5.2.2 Transformace výsledků dotazů na modelové třídy

Jak již bylo zmíněno, pro komunikaci s databází se používá knihovna *Neo4j Java Driver*, která se stará pouze o vykonání Cypher dotazu a vrátí výsledek ve formě obecných tříd `StatementResult`, `Record`, `Value` apod. Transformaci těchto výsledků na doménové objekty je tedy třeba implementovat.

5.2.2.1 ModelFactory

V aplikaci k tomu existuje třída `ModelFactory`, která umí z libovolného výsledku dotazu (resp. jeho části) vytvořit odpovídající instanci modelové třídy (např. `Transaction`, `BelongsToRelationship` apod.). Základní metoda pro tuto transformaci vypadá takto:

```
public static GraphObject createGraphObject(Value value) {
    if (value.hasType(TYPE_SYSTEM.NODE()))
        return createNode(value.asNode());
    else if (value.hasType(TYPE_SYSTEM.RELATIONSHIP()))
        return createRelationship(value.asRelationship());
    else
        throw new RuntimeException("Value has unsupported type");
}
```

Ve výše uvedených metodách `createNode`, resp. `createRelationship` je volána konkrétní metoda pro vytvoření instance daného typu, např. takto vypadá metoda pro vytvoření instance `Transaction`:

```
public static Transaction createTransaction(Node transactionNode) {
    long id = transactionNode.id();
    long timestamp = transactionNode.get("timestamp").asLong();
    String txid = transactionNode.get("txid").asString();
    return new Transaction(txid, timestamp, id);
}
```

5.2.2.2 Graph

Kromě třídy `ModelFactory` je v aplikaci ještě třída `Graph`, která reprezentuje celý graf výsledku dotazu. Instance této třídy načte celý výsledek dotazu a (pomocí `ModelFactory`) jej přetvoří do množiny uzlů a hran a jako celek je poté serializována pro zobrazení na frontendu.

5.2.3 Coinbase transakce

Coinbase transakce jsou speciálním typem transakcí – tyto transakce nemají žádný vstup, mají pouze výstup (více v sekci 2.2.2). Pro implementovanou

aplikaci je to důležité, protože všechny ostatní transakce se dají vyjádřit jako relace *vstupní entita*→`PAYMENT` *hrana*→*výstupní entita*. Protože coinbase transakce nemají vstup, nemohou mít ani vstupní entitu. Tento fakt ovlivňuje zejména výpočet zůstatku entity (jak současného zůstatku, tak průběhu zůstatku v čase) – nestačí totiž zpracovat s entitou spojené `PAYMENT` hrany, je třeba načíst i všechny coinbase transakce jejích adres. Cypher dotaz pro průběh zůstatku v čase pak vypadá takto (pro zobrazení grafu zůstatku je třeba provést ještě další zpracování vrácených dat):

```
MATCH (e:Entity {id: 1}) OPTIONAL MATCH (e)-[rel:PAYMENT]-()
OPTIONAL MATCH (e)<-[b_in:BALANCE_IN]-(t:Transaction {coinbase: true})
WITH collect(rel) AS rels, collect(b_in) AS b_ins, e, rel, b_in
WHERE all(r IN rels WHERE r.timestamp >= 1000000)
AND all(b IN b_ins WHERE b.timestamp >= 1000000)
RETURN rel, startnode(rel) = e AS outgoing, b_in
```

Jelikož coinbase transakce tvoří jen zlomek z celkového počtu transakcí, trvalo mi poměrně dlouho uvědomit si, proč můj původní výpočet zůstatku nefunguje. U výpočtu zůstatku adresy toto není problém, protože u ní se pracuje přímo s výstupy transakcí (a coinbase transakce jsou tak rovnou zahrnuty).

5.2.4 Transformace grafu pro frontend

V sekci 5.2.2 byla popsána třída `Graph`, reprezentující množinu uzlů a hran grafu. Frontend používá pro vykreslení grafu knihovnu *Cytoscape* (viz. sekce 4.2.2). Tato knihovna očekává elementy grafu v určitém formátu, je tedy třeba graf do této podoby transformovat.

Knihovna *Cytoscape* pracuje s daty ve formátu JSON a očekává elementy grafu v tomto tvaru:

```
elements: [
    {
        group: 'nodes',        // uzel
        data: {
            id: 'n1',
            timestamp: 101321549,
            ...
        }
    },
    {
        group: 'edges',        // hrana
        data: {
            id: 'n1',
            source: 'n1',
```

```
            target: 'n2',
            ...
        }
    }
]
```

U každého elementu je tedy nejdřív určeno, zda se jedná o uzel nebo hranu (atribut `group`), a poté atribut `data` obsahuje samotná data elementu (pro upřesnění – element může obsahovat i další atributy, ale ty nejsou pro vysvětlení principu důležité).

Aplikace pro JSON serializaci používá knihovnu *Jackson*, která je součástí frameworku Spring. Tato knihovna umožňuje pomocí anotací specifikovat, jakým způsobem se má z modelu vytvořit jeho JSON reprezentace. Jelikož v tomto případě je modelové třídy a jejich JSON reprezentace poměrně rozdílné, rozhodl jsem se použít serializační třídy, kde je možné přesně definovat formát vytvořeného JSONu. Pro každou modelovou třídu existuje odpovídající serializační třída. Použitá serializační třída se u modelu specifikuje pomocí anotace:

```
@JsonSerialize(using = TransactionSerializer.class)
public class Transaction extends Node {
    ...
}
```

Reprezentace jednotlivých modelů se liší v tom, zda jde o uzel nebo hranu, a poté v tom, jaká data modelu JSON obsahuje. Proto existují dvě základní serializační třídy `NodeSerializer` a `RelationshipSerializer`, které obsahují základní kostru výsledného JSONu. Od nich poté dědí serializační třídy jednotlivých modelů, které specifikují obsah atributu `data`:

```
public abstract class NodeSerializer<T> extends Serializer<T>{
    @Override
    public void serialize(T item, JsonGenerator jgen,
            SerializerProvider provider) throws IOException,
            JsonProcessingException {
        jgen.writeStartObject();
        jgen.writeStringField("group", "nodes");

        jgen.writeObjectFieldStart("data"); // atribut 'data'
        writeData(jgen, item); // tuto metodu implementuje dědící třída
        jgen.writeEndObject();

        jgen.writeEndObject();
```

```
    }
}

public class TransactionSerializer extends NodeSerializer<Transaction> {
    @Override
    public void writeData(JsonGenerator jgen,
            Transaction transaction) throws IOException {
        jgen.writeStringField("type", "transaction");
        jgen.writeStringField("id",
            String.valueOf(transaction.getId()));
        jgen.writeStringField("txid", transaction.getTxid());
        jgen.writeStringField("timestamp",
            String.valueOf(transaction.getTxid()));
    }
}
```

5.2.5 Filtrování podle času

Jednou z funkcí aplikace je i filtrování hledaných výsledků podle času – je možné specifikovat počáteční a koncové datum a pro všechna hledání cest mezi entitami a okolí entit pak budou brány v potaz jen transakce patřící do vymezeného období.

K tomuto účelu je implementována služba `Calendar`, která všem dalším částem backendu aplikace zprostředkovává informaci o tom, jaké časové období je pro vyhledávání relevantní. Přístup k této službě potřebují zejména *repository* třídy – pokud je specifikováno časové období, je třeba podle toho upravit odpovídající databázové dotazy (základní třída `Repository` pro tento účel obsahuje metodu `getCalendarCondition`).

Na frontendu existuje odpovídající objekt `Calendar`, který spravuje nastavení časového období uživatelem a přidává tuto informaci do všech požadavků odesílaných backendu, resp. tyto požadavky odesílá objekt `Queries`, který má k datům kalendáře přístup:

```
function Queries(calendar) {
    this.calendar = calendar;
}
...
Queries.prototype.neighbors = function(entityId, depth, direction,
        minValue, maxValue, callback) {
    $.post("neighbours", {
        id: entityId,
        depth: depth,
        direction: direction,
        minValue: minValue || -1,
        maxValue: maxValue || -1,
```

```
        // zde se přidávají data kalendáře
        fromDate: this.calendar.getFromDate(),
        toDate: this.calendar.getToDate()
    }, callback);
};
```

Nastavení backendové služby `Calendar` funguje pomocí tzv. *interceptoru*. Tímto termínem se ve frameworku Spring označuje komponenta, která dostane přístup k příchozímu požadavku ještě předtím, než je předán odpovídajícímu controlleru (v jiných frameworcích se stejnému principu říká např. middleware, filter aj.).

Aplikace obsahuje `CalendarInterceptor`, který prozkoumá data požadavku a pokud obsahují informaci o časovém období, nastaví podle toho službu `Calendar`. Díky tomu je zajištěno, že při začátku zpracování požadavku controllerem (a návaznými službami) je `Calendar` už nastaven a zároveň není třeba jeho inicializaci řešit v controlleru. Každý interceptor je třeba v aplikaci registrovat – v případě použití konfigurátoru Spring Boot pomocí tzv. *konfigurační třídy*, která je v tomto případě velmi jednoduchá:

```
@org.springframework.context.annotation.Configuration
public class AppConfiguration extends WebMvcConfigurerAdapter {
    @Autowired
    CalendarInterceptor calendarInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        super.addInterceptors(registry);
        registry.addInterceptor(calendarInterceptor);
    }
}
```

5.2.6 Implementace uživatelského rozhraní

Implementace uživatelského rozhraní proběhla přesně podle návrhu. Knihovna Cytoscape se ukázala jako vhodně zvolený nástroj pro vizualizaci grafu, zejména díky kvalitní podpoře reakcí na události v grafu a také díky širokým možnostem stylování prvků grafu.

Výsledná podoba uživatelského rozhraní je vidět v příloze B.

5.2.7 Získávání dat z fóra Bitcointalk

Jak bylo popsáno v sekci 4.2.3, profily uživatelů z fóra bitcointalk.org někdy obsahují bitcoinové adresy, je tedy možné k těmto adresám v databázi přiřadit informace o uživatelích z fóra.

Skript na získání těchto informací je napsán s pomocí frameworku *Scrapy* v programovacím jazyku Python (více viz. uvedená sekce Návrhu). V principu funguje tak, že postupně prochází profily uživatelů (v rozmezí zadaném při spuštění skriptu) a hledá v nich sekci *Signature*. V této sekci pak zkouší najít slovo, které by mohlo být bitcoinovou adresou a pokud ho najde, informace o uživateli se uloží do CSV souboru. Url profilu je jednoduše odvoditelná – má vždy tvar

```
bitcointalk.org/index.php?action=profile;u=x
```

a liší se jen hodnotou parametru `u`, který označuje *id* uživatele. Hledání adresy v podpisu uživatele také není nijak komplikované, testuje se jen několik pravidel (začátek slova, minimální/maximální délka)[20] pomocí této funkce:

```
def get_address(self, signature):
    words = signature.split(' ')
    for word in words:
        possible_addresses = []
        if word.find("1") is not -1:
            possible_addresses.append(word[word.find("1"):])
        if word.find("3") is not -1:
            possible_addresses.append(word[word.find("3"):])

        for possible_address in possible_addresses:
            if self.ADDRESS_MIN_LENGHT <= len(possible_address)
                    <= self.ADDRESS_MAX_LENGHT:
                return possible_address
    return None
```

Stahování jednotlivých profilů je třeba dávkovat – testováním jsem zjistil, že fórum umožní stažení 35 profilů za minutu, na každý další odpoví status kódem 503 (not available). Je tedy potřeba po stažení každých 35 profilů čekat do konce dané minuty.

Výsledný CSV soubor může být importován do Neo4j pomocí příkazu `LOAD CSV` (každý řádek v CSV souboru bude importován jako uzel typu `Identity`). Napojení importovaných identit a odpovídající adresy se provede tímto Cypher příkazem:

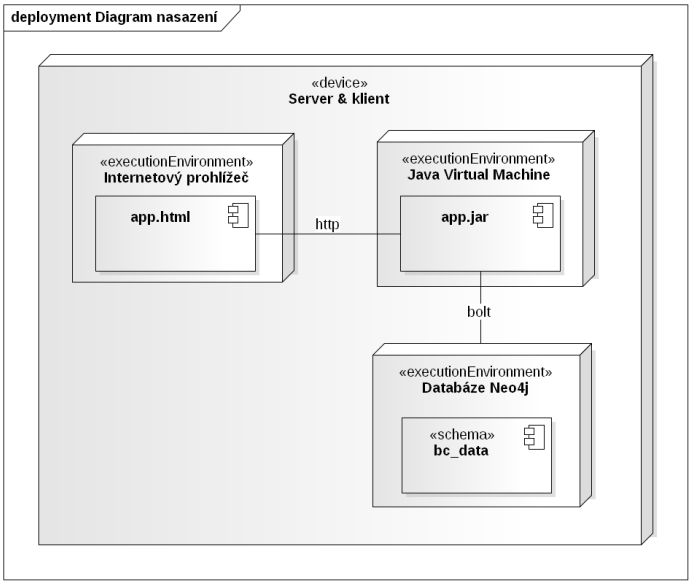
```
MATCH (ei:Identity), (a:Address) WHERE ei.address = a.address
MERGE (a)<-[ir:IDENTIFIES]-(ei)
RETURN count(ir) AS linkedCount
```

Tyto příkazy není třeba zadávat ručně – práce obsahuje skript, kterému se pouze předá vstupní soubor a připojení k databázi. Samotný import a napojení adres pak bude provedeno automaticky. Přesný postup je v příloze C.

5.3 Diagram nasazení

Obrázek 5.4 ukazuje nasazení aplikace. Jedním z nefunkčních požadavků je, aby aplikace včetně uživatelského rozhraní běžela na jednom stroji bez připojení k internetu.

Na tomto počítači je tak spuštěna jak samotná vyhledávací aplikace (jako tzv. *fat jar*, který kromě aplikace obsahuje i všechny její závislosti a také vestavěný Tomcat aplikační server), tak databáze Neo4j, a zároveň i internetový prohlížeč, přes který je aplikace přístupná.



Obrázek 5.4: Diagram nasazení

Kapitola 6

Testování

Tato kapitola se zabývá testováním aplikace. V první části se řeší měření výkonu aplikace (konkrétně vyhledávacích dotazů), druhá část popisuje funkční testy (testy zkoumající funkčnost aplikace jako celku) přítomné v aplikaci.

6.1 Měření výkonu

Aplikace umožňuje v rámci hledání cest mezi entitami a okolí entit specifikovat hloubku prohledávání (tzn. maximální počet transakcí na cestě mezi entitami). Jelikož prohledávanými daty je graf, jehož struktura není pravidelná, není možné přímo z hloubky hledání odvodit počet procházených uzlů/hran ani dobu hledání. Rozhodl jsem se proto prověřit vlastnosti grafu experimentálně.

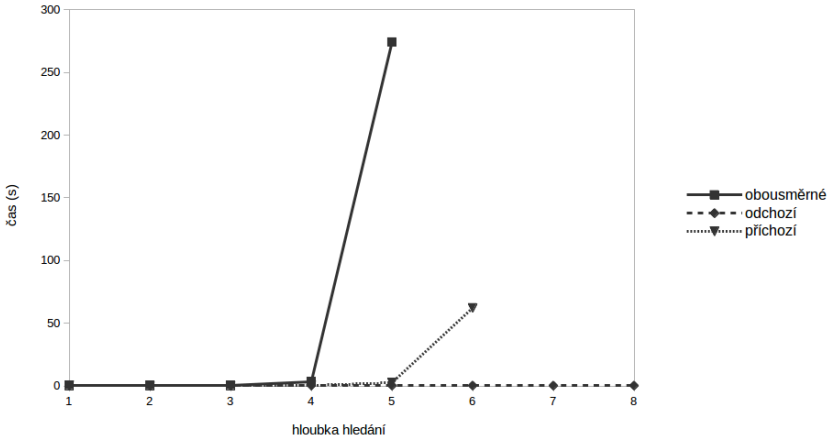
6.1.1 Hledání cest mezi entitami

Hledání všech cest mezi dvěma entitami je potenciálně velmi náročná operace. Pro ověření, jak dlouho může takové hledání trvat, je třeba vybrat vhodné entity, mezi kterými existuje velké množství cest. Postupoval jsem tak, že jsem v databáze hledal clustery (shluky entit s mnoha vzájemnými platbami) s velkým množstvím entit a zároveň velkým poměrným množstvím plateb mezi těmito entitami. Toho jsem dosáhl pomocí tohoto Cypher dotazu:

```
match (c:Cluster)<-[m:MEMBER_OF]-(e:Entity)-[p:PAYMENT]->()
with count(distinct m) as ms, count(distinct p) as ps, c
return c, ms, ps, 1.0 * ps / ms as ratio order by ms desc limit 20
```

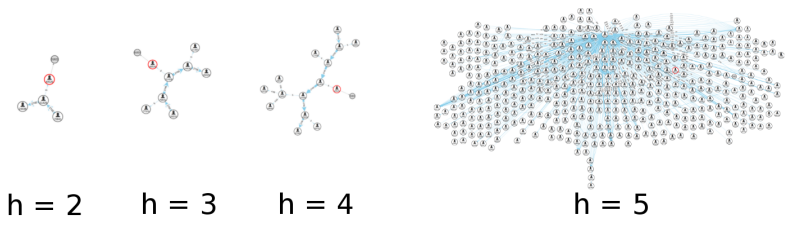
Na základě tohoto dotazu jsem vybral cluster s 125 entitami, mezi kterými existovalo 253 plateb. V tomto clusteru jsem poté (manuálně) vybral několik párů entit, mezi kterými existovalo mnoho cest. Pro tyto páry jsem poté zkoušel hledat cesty s postupně se zvyšující maximální délkou a měřil čas

hledání. Toto měření jsem prováděl zvlášť pro hledání odchozích, příchozích a libovolných plateb.



Obrázek 6.1: Cesty mezi entitami - závislost času hledání na jeho hloubce

Pro všechny entity dopadlo toto měření obdobně, jako je vidět v grafu na obrázku 6.1 – při malých hloubkách hledání trvá méně než vteřinu, a poté najednou prudce zpomalí (pro obousměrné cesty toto zpomalení přichází dříve než pro jednosměrné). Příčinou tohoto zpomalení je výskyt entity s velkým množstvím plateb na hledané cestě. Dobře je to vidět na obrázku 6.2 – do hloubky 4 roste okolí entity poměrně pomalu a pak se objeví entita se výrazně nadprůměrným počtem plateb a počet prvků grafu (a s ním i čas hledání) prudce vzroste.

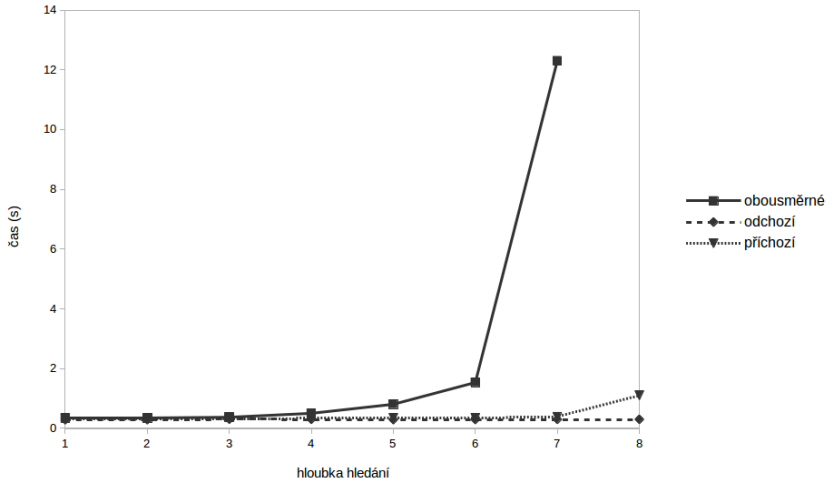


Obrázek 6.2: Příklad okolí entity v závislosti na hloubce h

6.1.2 Hledání okolí entity

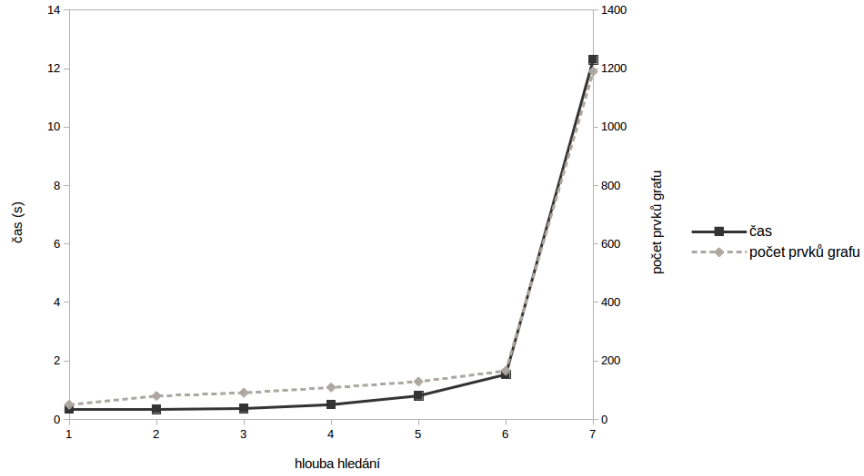
Měření času hledání okolí entity probíhalo obdobným způsobem jako u hledání cest – použil jsem stejný cluster a v něm vybral entity s velkým množstvím

plateb. Jak je vidět na obrázku 6.3, průběh je také obdobný jako v předchozím případě (ze stejných důvodů).



Obrázek 6.3: Okolí entity - závislost času hledání na jeho hloubce

Zdá se tedy, že čas hledání je závislý na velikosti výsledku hledání. Rozhodl jsem se tuto závislost ověřit, a kromě času jsem změřil i počet prvků grafu (uzlů a hran) pro jednotlivé hloubky. Výsledek je vidět na obrázku 6.4 – čas hledání je přímo úměrný velikosti výsledku.



Obrázek 6.4: Okolí entity - závislost času a velikosti výsledku hledání na jeho hloubce

6.1.3 Aplikace výsledků měření

Měření byla provedena zejména pro zjištění vhodné maximální hloubky hledání. Na jejich základě jsem se rozhodl zvolit jako maximální hloubku $h = 7$. Při větší hloubce bylo měření většinou třeba přerušit, protože trvalo neúměrně dlouhou dobu, příp. počet vrácených výsledků znemožňoval efektivní práci s nimi. Tato hloubka platí samozřejmě pro jeden dotaz – výsledný graf je možné opakovaně rozšiřovat hledáním okolí entit (pomocí kontextové nabídky).

6.2 Funkční testy

Aplikace obsahuje funkční testy backendu – tyto testy zkoumají aplikaci jako celek a každý test zkoumá nějakou její konkrétní funkci. Protože jde o webovou aplikaci, jednotlivé funkce mají podobu příchozích požadavků. Každý test tak vypadá takto:

1. Vytvoří se požadavek, který prověřuje zkoumanou funkčnost a tento požadavek se předá aplikaci.
2. Aplikace tento požadavek běžným způsobem vyřídí (že jde o test nehraje roli) a vrátí odpověď.
3. U této odpovědi se zkoumá její správnost., příp. se zkoumá obsah persistentního úložiště nebo další změny v aplikaci, pokud je vyřízení požadavku mohlo vyvolat.

Aplikace obsahuje testy všech hlavních controllerů. Konkrétně v implementované aplikaci existuje např. test, zkoumající výsledek požadavku na nalezení nejkratší cesty mezi dvěma entitami, který vypadá takto:

```
@Test
public void testShortestPath() throws Exception {
    mvc.perform(post("/shortestPath")
        .param("id1", "835")
        .param("id2", "133")
        .param("depth", "3")
        .param("direction", "both")
        .param("minValue", "-1")
        .param("maxValue", "-1")
        .param("shortestOnly", "true")
    )
        .andExpect(status().isOk())
        .andExpect(jsonPath("$", hasSize(5)))
        .andExpect(jsonPath("$[0].data.entityId", is("133")))
```

```
        .andExpect(jsonPath("$[3].data.entityId", is("835")))
        .andExpect(jsonPath("$[4].data.entityId", is("29")))
    ;
}
```

Proměnná `mvc` na začátku testu je instancí třídy `MockMvc` – tuto třídu poskytuje framework Spring a jejím účelem je simulace příchozích požadavků do aplikace. Ve skutečnosti tedy nejsou požadavky posílány např. na adresu `http://localhost/shortestPath` – adresa slouží pouze pro namapování požadavku na správný controller. Všechny ostatní vrstvy aplikace a fungují stejně jako u skutečných požadavků.

Kromě testovacího frameworku, který je součástí Springu, pracují testy ještě s knihovnou *Hamcrest*[24], která poskytuje tzv. *matchers*, což jsou funkce sloužící k vyhodnocení nějaké podmínky (v příkladu výše jde o volání `hasSize(5)` nebo `is("385")`).

Protože aplikace během funkčního testování pracuje stejně jako v běžném provozu, používá i databázi. Na přiloženém CD je připravená testovací databáze – testy počítají s přítomností konkrétních dat z této databáze. Jde o verzi vstupní databáze této práce, která ale obsahuje jen několik set prvních bloků blockchainu, aby bylo nad čím testovat. Zároveň bylo pro potřeby testů provedeno několik menších úprav dat (např. úprava adresy tak, aby odpovídala vloženému uzlu `Identity` a aby bylo možné testovat tuto vazbu).

Údaje pro připojení do databáze jsou uložené ve standardním souboru `application.properties`. Aby bylo možné použít pro testování jinou databázi, obsahují testovací balíčky vlastní soubor `application.properties`, jehož hodnoty jsou při testování nadřazeny hodnotám z původního souboru.

[24] `http://hamcrest.org`

Závěr

Cílem této práce bylo navrhnout a implementovat prototyp webové aplikace pro hledání v datech distribuované databáze blockchain, resp. v datech z ní odvozené databáze. Nejdříve bylo třeba prozkoumat a popsat fungování této databáze a na základě zadání zformulovat funkční i nefunkční požadavky na aplikaci. Jako základní prvek interakce s prohledávanými daty byl zvolen interaktivní graf, který by měl umožňovat jednak zobrazení vazeb v datech a jednak pokládání doplňujících dotazů (a obecně co nejplynulejší prohledávání grafu).

Poté byly zvoleny vhodné technologie a architektura aplikace – na backendu programovací jazyk Java a framework Spring, na frontendu zejména knihovna Cytoscape pro zobrazení a manipulaci s interaktivním grafem. Obě volby se ukázaly jako správné a během vývoje tyto nástroje nezpůsobovaly žádné problémy.

Při implementaci bylo třeba přehodnotit některá návrhová i technologická rozhodnutí – ukázalo se například, že přístup k databázi Neo4j pomocí Cypher dotazů je vhodnější než původně v části aplikace plánované použití objektově-grafového mapování (OGM). Při testování výkonu vyhledávacích dotazů se ukázalo, že jejich výkon značně (víc než jsem předpokládal) závisí na konkrétním místě prohledávání grafu, protože v něm existují entity (skupiny adres), které provedly řádově víc transakcí než ostatní – pokud se při procházení grafu „narazí“ na takovou entitu, rychlost procházení pak řádově klesne.

Existuje samozřejmě velký prostor pro rozšíření nebo vylepšení stávajících funkcí, např. zlepšení zmíněné kolísavé rychlosti prohledávání grafu. Aplikace také neobsahuje žádnou hlubší analýzu nalezených vazeb a pracuje se pouze s daty, která jsou „jistá“ (dala by se např. počítat míra jistoty, s jakou peníze poslané jednou entitou došly přes několik transakcí jiné entitě). Aplikace také jako prototyp neobsahuje práci s uživateli (možnost přihlášení, uložení vytvo-

řených entit apod.). V zásadě se ale podařilo funkční i nefunkční požadavky na aplikaci splnit.

Literatura

[1] Křeček, M.: *Vizualizace a vyhledávání v distribuované databázi.* Diplomová práce, České vysoké učení technické v Praze, 2017.

[2] Bitcoin Developer Guide. [online], [cit. 2017-04-14]. Dostupné z: `https://bitcoin.org/en/developer-guide`

[3] Mastering Bitcoin, Chapter 7. The Blockchain. [online], [cit. 2017-04-14]. Dostupné z: `http://chimera.labs.oreilly.com/books/1234000001802/ch07.html`

[4] Bitcoin - Average Number Of Transactions Per Block. [online], [cit. 2017-04-14]. Dostupné z: `https://blockchain.info/charts/n-transactions-per-block?timespan=all`

[5] Bitcoin - Total Number of Transactions. [online], [cit. 2017-04-14]. Dostupné z: `https://blockchain.info/charts/n-transactions-total?timespan=all`

[6] Ramba, J.: Grafová terminologie a dostupné technologie. [online], [cit. 2017-04-15]. Dostupné z: `https://www.zdrojak.cz/clanky/grafova-terminologie-a-dostupne-technologie/`

[7] Svoboda, M.: Graph Databases, Neo4j, Cypher. [online], [cit. 2017-04-15]. Dostupné z: `http://www.ksi.mff.cuni.cz/~svoboda/courses/2015-2-MIE-PDB/lectures/Lecture-09-Graph-Databases-Neo4j-Cypher.pdf`

[8] DB-Engines Ranking of Graph DBMS. [online], [cit. 2017-04-14]. Dostupné z: `https://db-engines.com/en/ranking/graph+dbms`

[9] Neo4j: The World's Leading Graph Database. [online], [cit. 2017-04-14]. Dostupné z: `https://neo4j.com/product/`

[10] Rathle, P.: Official Release: 3 Essentials of Neo4j 3.0, from Scale to Productivity & Deployment. [online], [cit. 2017-04-14]. Dostupné z: https://neo4j.com/blog/neo4j-3-0-massive-scale-developer-productivity/

[11] History of PHP. [online], [cit. 2017-04-19]. Dostupné z: http://php.net/manual/en/history.php.php

[12] Language Framework Popularity: A Look at PHP. [online], [cit. 2017-04-19]. Dostupné z: http://redmonk.com/fryan/2016/11/01/language-framework-popularity-a-look-at-php/

[13] Using Neo4j from PHP. [online], [cit. 2017-04-19]. Dostupné z: https://neo4j.com/developer/php/

[14] Shelajev, O.: Java Web Frameworks Index: February 2017. [online], [cit. 2017-04-19]. Dostupné z: https://zeroturnaround.com/rebellabs/java-web-frameworks-index-by-rebellabs/

[15] Using Neo4j from Java. [online], [cit. 2017-04-19]. Dostupné z: https://neo4j.com/developer/java/

[16] Webb, P.: Spring Boot – Simplifying Spring for Everyone. [online], [cit. 2017-04-20]. Dostupné z: https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone

[17] Visualizing a Neo4j Graph Database. [online], [cit. 2017-04-20]. Dostupné z: https://cambridge-intelligence.com/keylines/neo4j/

[18] Spring Framework - Architecture. [online], [cit. 2017-04-26]. Dostupné z: https://www.tutorialspoint.com/spring/spring_architecture.htm

[19] Intro to Cypher. [online], [cit. 2017-04-30]. Dostupné z: https://neo4j.com/developer/cypher-query-language/

[20] Bitcoin Address. [online], [cit. 2017-04-30]. Dostupné z: https://en.bitcoin.it/wiki/Address

Příloha A

Seznam použitých zkratek

RDBMS Relational database management system

BTC Bitcoin

UTXO Unspent Transaction Output

API Application Programming Interface

XML eXtensible Markup Language

CQL Cypher Query Language

JSTL JavaServer Pages Standard Tag Library

DFS Depth first search

VSC Comma-separated values

MVC Model-View-Controller

Příloha **B**

Popis uživatelského rozhraní

V této kapitole bude popsáno výsledné uživatelské rozhraní aplikace. Toto rozhraní vychází z návrhu v sekci 4.4 (a poměrně přesně mu odpovídá).

B.1 Základní rozvržení

Na obrázku B.1 je vidět základní rozvržení uživatelského rozhraní, které je rozděleno do tří sekcí:

- vyhledávací formuláře
- interaktivní graf
- informační panel

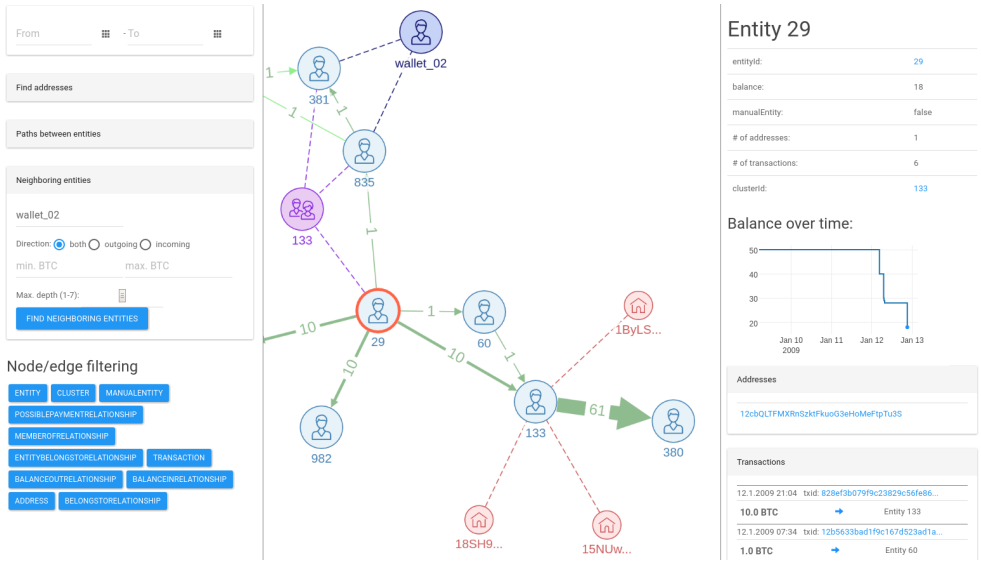
B.2 Vyhledávací formuláře

V levém panelu se nachází vyhledávací formuláře. Zde je možné hledat adresy, vytvářet z nalezených adres nové entity, hledat cesty (posloupnosti transakcí) mezi entitami, hledat okolí entit a také filtrovat zobrazené uzly v grafu. Hledání cest a okolí entit je možné časově omezit pomocí kalendáře v horní části panelu.

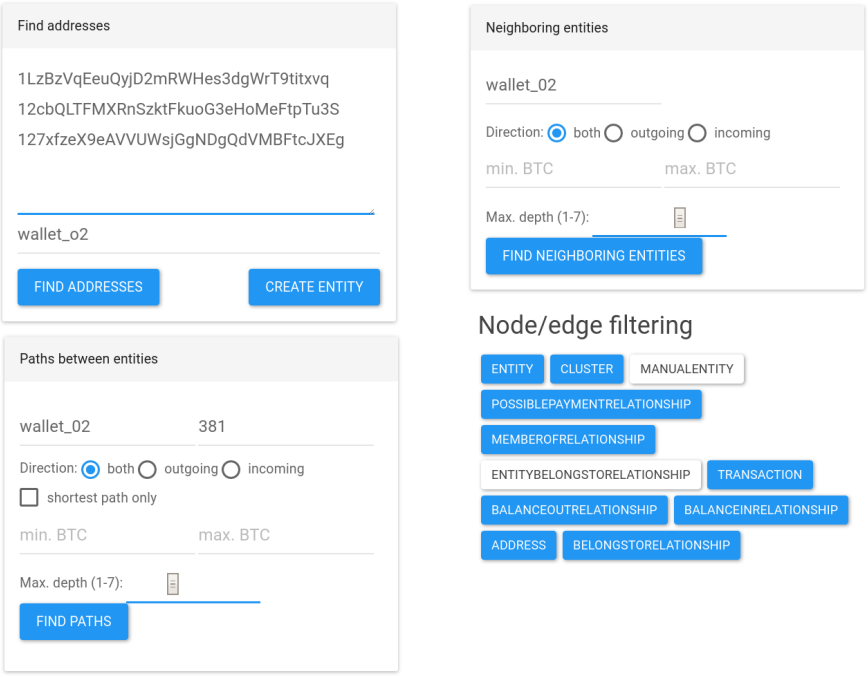
B.2.1 Hledání adres a vytváření nových entit

V tomto formuláři je možné zadat jednu nebo více adres (každou na nový řádek) a poté je tlačítkem *Find addresses*) vyhledat a zobrazit v grafu společně s jejich entitami a příp. platbami, které mezi těmito entitami proběhly.

Druhou možností je vyplnit k adresám ještě jméno nové entity a pomocí tlačítka *Create entity* tuto entitu uložit. Takto vytvořenou entitu je možné použít při hledání stejným způsobem jako ostatní entity (rozdíly mezi stan-



Obrázek B.1: Základní rozvržení uživatelského rozhraní



Obrázek B.2: Vyhledávací formuláře a filtrování

dardními a manuálně přidanými entitami ale existují, viz. sekce 5.1.2 v Implementaci).

B.2.2 Hledání cest mezi entitami

Tento formulář slouží pro hledání cest mezi dvěma entitami. Je třeba zadat identifikátory obou entit a (nepovinně) některá z dalších upřesňujících kritérií:

- směr cesty (příchozí/odchozí/libovolné transakce)
- hledání pouze nejkratší cesty
- omezení minimální/maximální hodnoty transakce na cestě
- maximální délka cesty

B.2.3 Hledání okolí entity

Pomocí tohoto formuláře je možné hledat okolní entity k zadané entitě. Je třeba zadat identifikátor entity a volitelně některá z upřesňujících kritérií. Ta jsou stejná jako v předchozím případě, jen zde není volba pro hledání nejkratší cesty.

B.2.4 Filtrování zobrazených uzlů grafu

Pomocí těchto přepínačů je možné skrýt/zobrazit konkrétní typ uzlu v grafu. Přepínače pracují s aktuálně zobrazenými uzly, tzn. pokud např. po skrytí transakcí bude do grafu přidána nová transakce, pak tato transakce nebude skryta.

B.3 Interaktivní graf

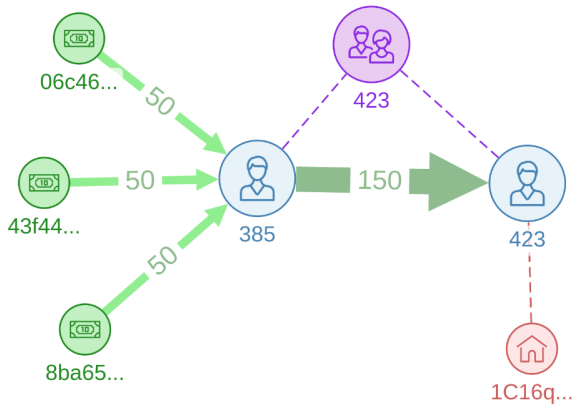
V prostřední části je graf, který slouží pro orientaci ve vazbách mezi prvky a k interakci s nimi.

V grafu je možné volně přesouvat všechny uzly i skrývat konkrétní uzly/hrany. Po výběru některého z uzlů se ve vedlejším informačním panelu zobrazí souhrnné informace o tomto prvku. Funguje to i obráceně – pokud se uživatel v informačním panelu „proklikne“ např. z detailu entity na některou její adresu, bude tato adresa zobrazena v grafu.

Obrázek B.3: Typy uzlů: transakce, adresa, entita, manuální entita, cluster

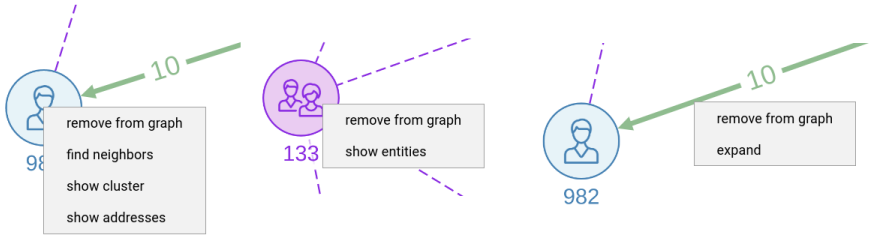
Na obr. B.3 jsou zobrazeny uzly, se kterými se v grafu pracuje – jde o transakce, adresy, standardní entity (skupiny adres), clustery (skupiny entit) a manuálně přidané entity.

Graf obsahuje dva základní typy hran – plnou čarou s šipkou a hodnotou jsou značeny toky peněz (mezi entitami/adresami), přerušovanou čarou je značena příslušnost jednoho uzlu k jinému (např. entity jsou touto čarou spojeny se svým clusterem). Na obrázku B.4 je vidět toto základní rozdělení: tři hrany z transakcí do entity (jde o coinbase transakce) a jedna platba mezi entitami jsou plnou čarou, příslušnost entit ke clusteru a adresy k entitě je značena přerušovanou čarou.



Obrázek B.4: Typy hran

Každý uzel a hrana grafu má také kontextovou nabídku, která se zobrazí po kliknutí pravým tlačítkem myši na daný prvek. Nabídka umožňuje u každého prvku jeho skrytí, u entity, clusteru a platební hrany mezi entitami umožňuje navíc další činnosti (např. u entity hledání okolí) – tyto činnosti jsou vidět na obrázku B.5.



Obrázek B.5: Kontextová nabídka

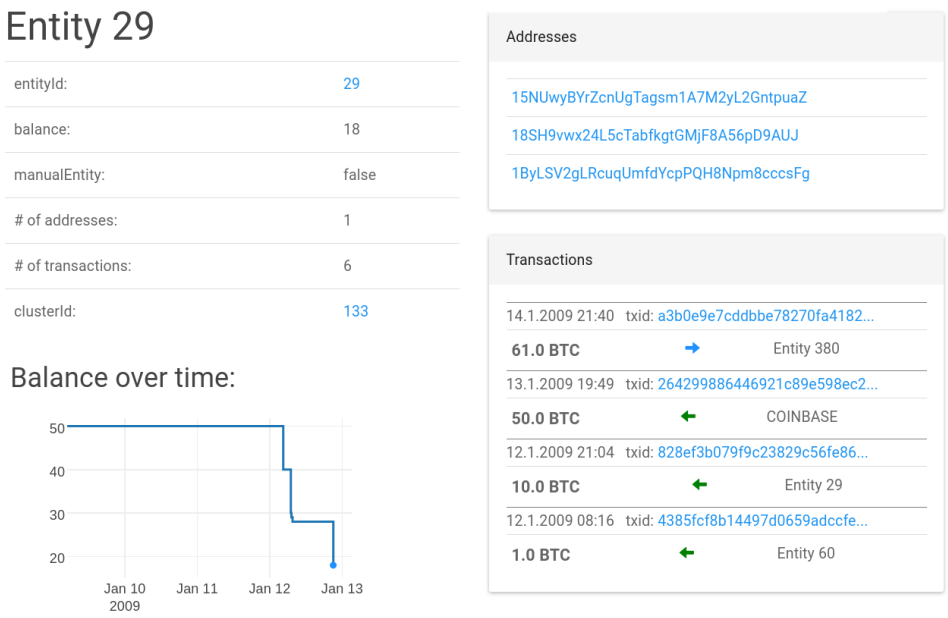
B.4 Informační panel

V tomto panelu jsou zobrazeny souhrnné informace o vybraném uzlu grafu. Souhrn u každého typu uzlu obsahuje trochu jiné informace:

- Transakce
  - základní informace (txid, datum a čas transakce, zda jde o coinbase transakci)
  - seznam vstupů a jejich hodnot (adresy)
  - seznam výstupů a jejich hodnot (adresy)
- Adresa
  - základní informace (adresa, odpovídající entita, zůstatek, počet transakcí)
  - graf průběhu zůstatku za zvolené období
  - seznam transakcí
- Entita
  - základní informace (zůstatek, počet adres a transakcí, odpovídající cluster)
  - graf průběhu zůstatku za zvolené období
  - seznam adres
  - seznam transakcí
- Manuální entita
  - základní informace (zůstatek, počet entit, adres a transakcí)
  - graf průběhu zůstatku za zvolené období
  - seznam entit
  - seznam adres
  - seznam transakcí
- Cluster
  - základní informace (počet entit a transakcí mezi nimi)
  - seznam entit
  - seznam transakcí

Souhrny jednotlivých prvků jsou vzájemně provázané – např. z entity je možné se odkazem dostat na její libovolnou adresu nebo transakci, z transakce na její vstupní/ výstupní adresy apod.

Příklad zobrazených informací pro entitu s odkazy na související prvky je na obrázku B.6.



Obrázek B.6: Souhrnné informace o entitě

Příloha C

Návod na import externích informací o adresách

Přiložené CD obsahuje ve složce `src/identity_import` 2 skripty:

- `scraper.py` – slouží k získání adres a odpovídajících uživatelů z fóra bitcoitalk.org
- `importer.py` – slouží k importu získaných informací do databáze a napojení těchto informací na existující adresy

Oba skripty vyžadují Python 3.4, `scraper.py` potřebuje balíček `scrapy`, `importer.py` potřebuje balíček `neo4j` (`pip install neo4j scrapy`).

C.1 Použití scraper.py

```
scrapy runspider scraper.py -o {výstupní soubor} -a first={počáteční
id} -a last={koncové id}
```

- *výstupní soubor* – název výstupního souboru (je třeba zadat název s příponou .csv, aby výsledný soubor fungoval s následujícím skriptem)
- *počáteční id* – id prvního uživatele, který bude skriptem zpracováván
- *koncové id* – id posledního uživatele, který bude skriptem zpracováván

C.2 Použití importer.py

```
python importer.py {vstupní soubor} {login} {heslo} {port}
```

- *vstupní soubor* – cesta k vstupnímu CSV souboru (který je výstupem předchozího skriptu)
- *login* – uživatel pro připojení k Neo4j
- *heslo* – heslo pro připojení k Neo4j
- *port* – port, na kterém běží Neo4j

Skript předpokládá, že Neo4j běží na adrese `http://localhost:{port}`, data budou importována do aktivní databáze.

Příloha D

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src` ..... zdrojové kódy
  - `app` ..... zdrojový kód vyhledávací aplikace
  - `identity_import` .. skripty pro získání a import identifikačních údajů
- `resources` ..... pomocné soubory
  - `test_db.zip` ..... testovací Neo4j databáze
- `thesis` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `src` ..... zdrojová forma práce ve formátu LaTeX