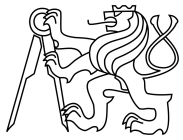
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Vizualizace a vyhledávání v distribuované databázi
Student:	Bc. Martin Křeček
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. Zmapujte vhodné dostupné databáze pro uložení objemných grafových struktur (vstupní databáze) a vyberte po dohodě s vedoucím práce jednu z nich. Navrhněte databázové schéma pro tuto databázi vhodné pro uložení veškeré informace obsažené ve vstupní databázi a proveďte její import do zvolené databáze. Při návrhu uvažujte situaci, ve které je jeden či více uzlů v grafu sloučeno a je chápáno jako shluk uzlů, a dále fakt, že data v databázi mohou obsahovat časovou značku, hrany jsou orientované a mohou mít svoji váhu. Nad zvolenou databází navrhněte a implementujte vyhledávání podle vedoucím zadaných kritérií, minimálně však: i) všechny cesty ze shluku A do shluku B a ii) sousední shluky ke shluku A, vše s možností omezení časové značky a váhy. Navrhněte a implementujte webovou aplikaci pro zadávání vyhledávacích kritérií uživatelem a vizualizaci výsledků v prostředí Kibana.

Seznam odborné literatury

Dodá vedoucí práce.

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

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 1. února 2017

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

Diplomová práce

Vizualizace a vyhledávání v distribuované databázi

Bc. Martin Křeček

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

5. května 2017

Poděkování

Děkuji vedoucímu této práce, Ing. Tomáši Zahradnickému, Ph.D., za vstřícnost při domluvě tématu a za pomoc při tvorbě práce. Rovněž bych rád poděkoval Ing. Dušanu Mondekovi za konzultace při finalizaci práce.

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 5. května 2017 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Martin Křeček. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

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

Abstrakt

Práce se zabývá analýzou distribuované databáze blockchain, která je použita jako základní technologie pro kryptoměnu Bitcoin. Cílem je importovat data z této databáze do databáze vhodné pro analytické dotazy a implementovat aplikaci, která poskytne možnost získat vhled do importovaných dat a vyvodit z nich užitečné informace nebo znalosti.

Klíčová slova distribuovaná databáze, grafová databáze, kryptoměna, shlukování, Neo4j, Java

Abstract

The aim of this thesis is to analyze the distributed database blockchain, which is used as an underlying technology for the Bitcoin cryptocurrency. The goal is to import data from this database to a database suitable for analytical queries and to implement an application, that provides the possibility to gain an insight into the imported data and deduce useful information or knowledge from it.

Keywords distributed database, graph database, cryptocurrency, clustering, Neo4j, Java

Obsah

Seznam obrázků

Úvod

V prostředí internetu je jedním ze zásadních problémů důvěra mezi jeho uživateli, kterou nelze zajistit konvenčními metodami, na které jsou lidé zvyklí v běžném „offline“ životě. Místo toho je třeba přicházet s postupy, které zajistí potřebnou úroveň důvěryhodnosti v kyberprostoru.

Z tohoto důvodu vznikla i technologie, kterou se tato práce zabývá. Je jí blockchain, což je distribuovaná databáze sloužící jejím uživatelům k uchování ověřitelných informací bez centrální důvěryhodné autority. S pomocí blockchainu lze ověřit určitá fakta na internetu, čímž je zajištěna důvěra mezi uživateli bez nutnosti jejich osobního setkání nebo účasti třetí strany. Blockchain je založen na ověřitelnosti a nezměnitelnosti záznamů, díky čemuž není nutné se spoléhat na důvěryhodnou třetí stranu, když spolu chtějí uživatelé důvěryhodně interagovat.

Databáze v blockchainu však může být značně nepřehledná a není snadné z ní získat užitečné informace. Pro zjištění, jaký je aktuální stav některých informací v této databázi je nutné spojit mnoho kusů dat, které jsou na první první pohled nečitelné a samostatně z nich není mnoho patrné. Pokud se ovšem podaří tato data správně pospojovat, lze získat užitečné znalosti. Právě touto analýzou dat z databáze za účelem získání souvislých informací se zabývá tato práce.

Kapitola 1

Cíl práce

Cílem této práce je seznámit se s technologií distribuované databáze blockchain, zmapovat vhodné dostupné databáze pro uložení v ní obsažených grafových struktur a následně tyto struktury importovat. Dále je cílem implementovat nad importovanými daty takové předzpracování, které umožní následně vizualizovat a analyzovat obsah databáze se zaměřením na shluky uzlů a hledání cesty mezi nimi, včetně filtrování podle časové značky.

V navazující práci [1] je pak, po dohodě s vedoucím práce, řešena implementace webové aplikace pro vizualizaci a analýzu výstupů algoritmů vytvořených v rámci této práce.

Kapitola 2

Blockchain

Zdrojovou databází, ze které jsou pro tuto práci získávána data k analýze, je blockchain . Lze ji popsat jako druh distribuované databáze, která obsahuje chronologicky uspořádané záznamy. Konkrétně zde jde o záznamy pro digitální měnu Bitcoin, která používá blockchain ke svému fungování.

V této kapitole jsou nastíněny principy této distribuované databáze a nejdůležitější termíny, které jsou používány dále v textu. Z detailů fungování jsou vybrány ty informace, které jsou podstatné vzhledem k analýzám prováděným v rámci této práce. Některé další technické podrobnosti, které nejsou významné pro získávání potřebných informací, jsou zmíněny jen ve zjednodušené formě nebo vynechány.

2.1 Principy

Základním stavebním kamenem blockchainu jsou jednotlivé bloky, které jsou navzájem provázány a tvoří tedy souvislý „řetěz“ (odtud název z angličtiny – blockchain – tedy „řetěz bloků“).

Digitální měna Bitcoin využívá tyto bloky k postupnému ukládání záznamů, pomocí kterých je distribuovaně udržována „pravda“ o proběhlých platbách. Těmito záznamy jsou transakce ovlivňující pohyb měnových prostředků (*bitcoinů* – BTC), kterým je věnována sekce 2.3.

Každý uzel, který je přes internet připojen do Bitcoin sítě, si udržuje svou vlastní lokální kopii[1] blockchainu a je zcela „rovnocenným“ účastníkem komunikace. Není zde žádný centrální „řídící“ uzel ani uzly se speciálními vyhrazenými funkcemi, které by byly kritické pro chod systému. Takováto architektura sítě je označována jako *peer-to-peer* (též zkráceně *P2P*) a zajišťuje, že systém jako celek není závislý na konkrétních uzlech, které by měly speciální funkce nebo výsadní postavení. Tím je chráněn jeden ze základních principů Bitcoinu,

[1] Ne všechny uzly potřebují zcela kompletní data, některým k fungování stačí jejich podmnožina.

kterým je jeho decentralizovanost vylučující kontrolu nad záznamy ze strany jakékoliv centrální autority [2].

Výše zmiňovaná „pravda“, která je v blockchainu udržována, tedy vychází z konsenzu uzlů připojených do Bitcoin sítě. Každá operace musí být potvrzena a zařazena do „řetězu bloků“ procesem těžby, kterému se věnuje sekce 2.4. Tento proces vyžaduje určitý výpočetní výkon a zaručuje, že potvrzení „pravdy“ je založeno de facto na práci, jež byla pro její přijetí v konsenzu vykonána. Nutnou podmínkou pro fungování tohoto systému je tedy to, že žádný účastník v síti nezíská více než 50 % výpočetního výkonu celé sítě, protože to by mu umožnilo měnit záznamy v blockchainu [3].

Provázanost bloků, která je spolu s jejich strukturou detailněji popsána v podsekci 2.1.1, zajišťuje jejich neměnnost. Záznam, který je do blockchainu jednou zařazen, prakticky nelze změnit. Přesněji řečeno, teoreticky to možné je, avšak bylo by to výpočetně příliš náročné.

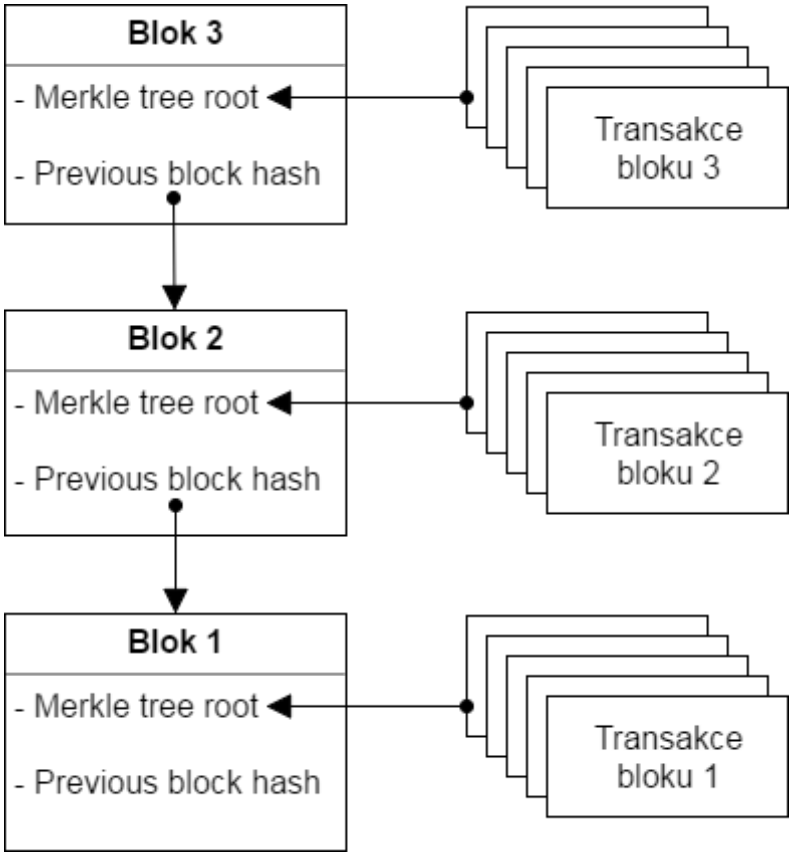
2.1.1 Bloky

Blok je ve své podstatě sdružení několika transakcí, které je potřeba potvrdit a přidat do blockchainu. Transakcemi se zabývá sekce 2.3 níže, zde je popsána struktura bloků a princip, jakým jsou mezi sebou provázány.

Jeden blok se skládá z hlavičky a seznamu transakcí, které jsou do něj zařazeny. Identifikátorem bloku je jeho *hash*, resp. hash jeho hlavičky. Tento identifikátor není přímo v bloku uložen, ale lze ho kdykoliv spočítat z údajů v hlavičce. Pro účely pochopení základního principu jsou z hlavičky bloku podstatné následující položky:

- Previous block hash – hash předchozího bloku, tj. bloku, se kterým bude tento blok svázán
- Merkle tree root – kořen stromu hashů transakcí v bloku, vysvětlen níže v podsekci 2.1.2
- Nonce – pole používané pro těžbu, která je rozebrána v sekci 2.4

Hash předchozího bloku je právě tím údajem, který spojuje jednotlivé bloky do provázaného „řetězu“. Tím, že hlavička tohoto bloku obsahuje hash bloku, který mu předcházel, je zajištěno, že potvrzené bloky jsou nadále neměnné. Vazba mezi bloky je ilustrována na obrázku 2.1.



Obrázek 2.1: Vzájemná provázanost bloků

Úplným „začátkem“ celého blockchainu je tzv. *Genesis block*, který je uložený přímo ve zdrojových kódech Bitcoinu. Jeho hash je

```
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

a všechny validní bloky v blockchainu jsou jeho „potomky“, takže postupný průchod bloků pomocí *previous block hash* směrem do historie by vždy skončil u tohoto *genesis* bloku.

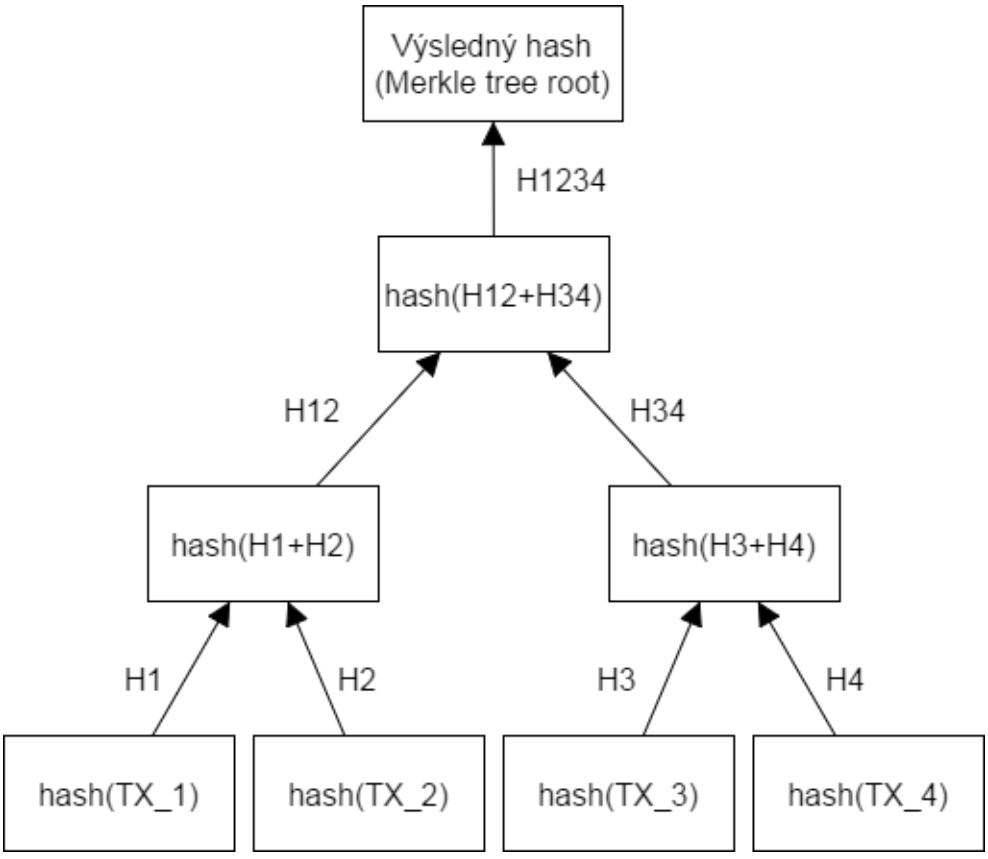
Pokud by se někdo pokusil změnit blok, který byl již v minulosti potvrzen a jsou na něj navázány další bloky, musel by pak také pozměnit všechny bloky následující po změněném, aby zajistil, že jejich hlavičky obsahují správné hashe předchozího bloku. Tento proces je však výpočetně náročný, jak je popsáno v kapitole 2.4, takže čím více následujících bloků postupně navazuje na daný blok, tím těžší je jakkoliv ho pozměnit. Uzly v Bitcoin síti tak vyjadřují potvrzení bloku implicitně tím, že jeho hash použijí v dalším vytvářeném bloku jako hash předchozího bloku [4].

2.1.2 Merkle tree

Jak bylo popsáno v subsekci 2.1.1, hlavička bloku v blockchainu obsahuje položku nazvanou *Merkle tree root*. Ta slouží jako určitá reprezentace nebo „shrnutí“ transakcí zahrnutých v daném bloku. Merkle tree je datová struktura, která je použita k získání této výsledné sumarizace transakcí a umožňuje ověřit, zda je určitá transakce součástí daného bloku.

Jedná se o postupný hash dat v transakcích tak, že nakonec jsou všechny transakce obsažené v bloku reprezentovány jedním výsledným hashem. Algoritmus výpočtu takového hashe postupně vytváří stromovou strukturu, ve které se používají mezivýsledky jako vstupy do dalšího hashování. Transakce tvoří listy tohoto binárního stromu a jejich hashe jsou vstupem do další úrovně stromu.

Postup vytvoření Merkle tree pro 4 transakce je ilustrován na obrázku 2.2, kde výraz `A+B` značí konkatenaci dat A a B, `hash(X)` značí hashovací funkci aplikovanou na data X a označení `TX_1` reprezentuje transakci číslo 1.



Obrázek 2.2: Tvorba Merkle tree pro 4 transakce

2.2 Adresy a peněženky

Pokud chce uživatel Bitcoinu mít možnost přijmout platbu, může si vytvořit tzv. *adresu*. Ta je vytvořena hashováním z veřejného klíče, který si uživatel vygeneruje. Veřejný klíč je svázán s privátním klíčem, který je rovněž vygenerován, jako v jiných případech užití asymetrické kryptografie.

Pomocí adresy lze v transakci určit, kam chce uživatel bitcoiny poslat. Záměrně zde není použita formulace „komu poslat“, jelikož adresy nejsou nijak spjaty s konkrétním uživatelem.

Protože si každý uživatel může vytvořit libovolné množství adres, je nutné mít systém uchovávání přehledu o těchto adresách. Zároveň je potřeba ke každé adrese mít uložen příslušný privátní klíč, který uživateli umožní nakládat s prostředky spjatými s danou adresou.

Proto existují *peněženky*, což jsou aplikace umožňující ukládat privátní a veřejné klíče, vést záznamy o adresách patřících danému uživateli a aktualizovat zůstatky na těchto adresách. Zároveň je běžně možné pomocí těchto aplikací i provádět platby, tedy publikovat nové transakce do blockchain sítě.

Nejpodstatnější z hlediska analýz prováděných v této práci je u peněženek fakt, že sdružují několik adres patřících jednomu uživateli. Z toho důvodu je užitečné na ně pohlížet jako na určité *entity*, u kterých je běžné, že používají prostředky z více adres, jak je popsáno v sekci 5.1.1. Uživatel si rovněž může vytvořit i více peněženek, takže při analýzách nelze předpokládat, že tyto *entity* jsou již konečným obrazem jednotlivých uživatelů Bitcoinu.

2.3 Transakce

Pohyb bitcoinů je implementován pomocí transakcí, což jsou datové struktury obsahující informace o přenosech prostředků mezi uživateli v síti. Tyto přenosy pracují se základními elementy, které jsou nazývány *unspent transaction output* (zkráceně *UTXO*, dále v textu též *output*).

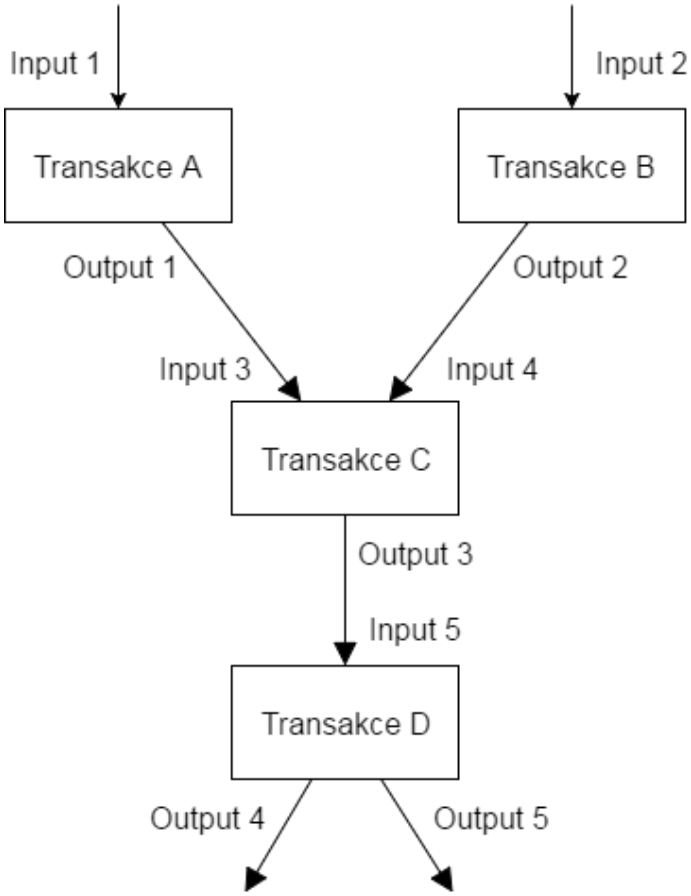
Jeden *UTXO* reprezentuje určitý objem bitcoinové měny, který je přiřazen konkrétnímu uživateli, resp. adrese, která s ním může dále nakládat. Tyto outputy „putují“ od transakce k transakci, nejsou nikde centrálně shromažďovány a seskupovány u konkrétních adres[2]. Z toho vyplývá, že získat z blockchainu například informaci o tom, kolik má daná adresa k dispozici BTC, není zcela triviální [2].

Jakýkoliv pohyb těchto outputů tedy znamená jejich použití v transakci. Běžná situace je taková, že uživatel, který má právo nakládat s daným *UTXO* (tedy je vlastníkem adresy, na kterou byl poslán), tento obnos získaný z předešlé transakce použije v nové transakci. Platí, že nelze použít pouze část

[2] Aby měl uživatel přehled o „zůstatcích“ na svých adresách, tuto funkcionalitu zpravidla implementují peněženky. Takže ačkoliv v Bitcoin síti není k adrese přiřazen žádný údaj jako „zůstatek“, uživatel ho má obvykle k dispozici díky své peněžence.

UTXO, tudíž pokud chce uživatel poslat jen část daného outputu, který má k dispozici, musí ho do transakce zapojit celý a jeho přebytečnou část odeslat na některou z jeho vlastních adres.

Jak již bylo zmíněno výše, *UTXO* vstupují do a vystupují z transakcí, přičemž transakce může mít jeden nebo více vstupů a jeden nebo více výstupů. Vstup do transakce odkazuje na výstup z předešlé transakce, který by měl být v této transakci použit. Schema s příkladem použití inputů a outputů v transakcích je na obrázku 2.3, kde je patrná provázanost inputů s outputy předešlých transakcí a také proměnlivé počty inputů a outputů v jednotlivých transakcích.



Obrázek 2.3: Průchod inputů a outputů transakcemi

2.3.1 Coinbase transakce

Speciálním případem transakcí jsou transakce typu *coinbase*, které vznikají při procesu těžby popsaném v následující sekci 2.4.

Tyto transakce nemají inputy, protože nepoužívají prostředky z předešlých transakcí, ale vytváří nové bitcoiny. Vznikají jako odměna „těžařům“, kteří poskytují svůj výpočetní výkon pro potvrzování bloků.

Do každého nově vytvářeného bloku je přidána tato speciální *coinbase* transakce, která vytváří output převádějící odměnu na adresu „těžaře“, kterému se podařilo blok potvrdit.

Z hlediska analýzy plateb v Bitcoin síti nejsou tyto transakce příliš zajímavé, avšak v datech jsou zpracovávány a jejich význam je zde proto krátce objasněn.

2.4 Těžba

Někteří uživatelé v Bitcoin síti („těžaři“) se mohou rozhodnout, že se budou podílet na potvrzování bloků, tedy *těžbě*. Tato činnost spočívá v hledání takové hodnoty *nonce* (položka hlavičky bloku – popsáno výše v sekci 2.1.1), aby hash hlavičky tohoto bloku splňoval stanovenou podmínku.

Tato podmínka stanovuje, že hodnota hashe hlavičky bloku musí být menší než stanovená mez[3]. Nalézt validní řešení tohoto problému je výpočetně velmi náročné, protože díky kryptografickým vlastnostem hashovací funkce lze postupovat pouze metodou hrubé síly, tj. zkoušením různých úprav hodnoty *nonce* a počítáním hashe hlavičky bloku.

Právě na náročnosti této úlohy je založena ona „neměnitelnost“ bloků, které jsou potvrzeny a zařazeny do blockchainu. Pokud by totiž některý uživatel změnil již potvrzený blok, musel by tuto úlohu pak znovu vyřešit pro všechny bloky, které následují po něm. Složitost takového „útoku“ tak stoupá s počtem bloků, které jsou potvrzeny po daném bloku, a již po zhruba 6 navazujících blocích je to považováno za prakticky téměř nemožné [3].

Uzel, kterému se podaří nalézt řešení, ho vyšle do sítě, ostatní uzly toto řešení ověří a poté přidají vytěžený blok do blockchainu, čímž je potvrzen. Zároveň s ostatními transakcemi je v bloku potvrzena i *coinbase* transakce přidaná „těžařem“, která mu zajišťuje získání odměny za vytěžení bloku. Tato odměna má dvě části:

- Pevně stanovená část – nové BTC vytvářené s každým blokem
- Variabilní část – suma poplatků z jednotlivých transakcí v bloku

[3]Tato mez je pravidelně každých 2016 bloků upravována, aby vytěžení jednoho bloku trvalo přibližně 10 minut.

Kapitola 3

Cílová databáze

Pro uložení a zpracování analyzovaných dat je třeba použít vhodnou databázi, která bude schopna podporovat jak potřebný formát dat, tak potřebné dotazy nad těmito daty.

3.1 Požadavky

Databáze bude využívána jednak pro import dat z blockchainu a jednak pro vykonávání analytických a reportovacích operací nad importovanými daty. Těmito operacemi jsou jak předzpracování importovaných dat, tak i uživatelské dotazy.

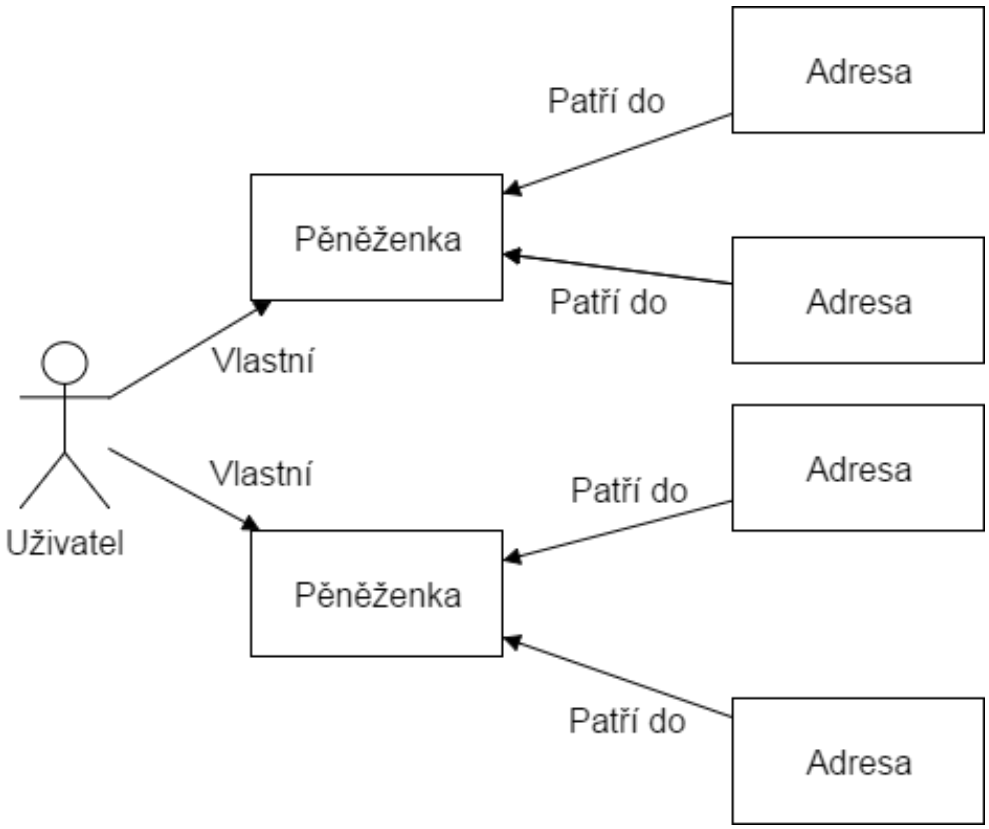
Základní množina nutných datových struktur, které by mělo být možné do databáze uložit, je následující:

- Blok
- Transakce
- Adresa
- Peněženka nebo shluk adres (v případě stavu po transformaci a analýze)
- Shluk peněženek nebo uživatelů (v případě stavu po transformaci a analýze)

Vzájemné vztahy mezi uživateli, peněženkami a adresami, které bude nutné v databázi modelovat, jsou znázorněny na obrázku 3.1.

Operace, které bude nutné nad datovou strukturou provádět, lze rozdělit následovně:

- Importovací
  - Vytvořit potřebné typy objektů s jejich atributy



Obrázek 3.1: Hierarchie vztahů k peněžence

  - Navázat transakci na existující adresy v databázi
  - Seskupit adresy do peněženky

- Předzpracovací
  - Tranzitivně přidat vazby mezi objekty
  - Uložit k objektům a vazbám vlastní atributy
  - Procházet sousledné objekty na základě odkazů mezi nimi

- Reportovací
  - Nalézt cestu mezi dvěma adresami nebo shluky
  - Nalézt adresy patřící do stejného shluku
  - Nalézt adresy nebo peněženky propojené transakcemi
  - Nalézt sousední shluky adres, peněženek nebo uživatelů

Dále musí být databáze schopna pojmout požadované množství dat, které lze očekávat v řádek stovek GB. Zároveň je pro použitelnost nutné, aby reportovací dotazy zvládala zodpovídat v reálném čase a uživatel s ní tak mohl „normálně“ pracovat. Lze očekávat, že předzpracovací operace budou trvat výrazně déle, avšak údaje jimi vytvářené je nutné získat a uložit pouze jednou, tento fakt proto nebrání v použití databáze.

3.2 Dostupné možnosti

Jako možné databáze pro reprezentaci dat z blockchainu a zbytek zde rozebíraného případu užití se nabízí několik možností, které jsou popsány níže.

MySQL nebo PostgreSQL a další relační databáze jsou pro zadaný problém nevhodné struktury, jelikož jakýkoliv pohyb přes jednu hranu (transakci) by zde znamenal buďto JOIN nebo další SELECT s vyhledáváním podle indexu. Toto uspořádání by mohlo být vhodné například pro jednorázové nalezení všech sousedů daného uzlu, avšak v případě hledání cesty delší než 1 skok by již bylo nutné „průchod grafem“ simulovat aplikačně a databáze by poskytovala pouze seznam sousedů. Navíc by zde byl poměrně zásadní problém v tom, že relační databáze mají pevně stanovené schema, takže například proměnlivý počet výstupních uzlů z transakce by se v nich velmi těžko modeloval.

Cassandra nebo Redis a jim podobné NoSQL databáze by sice byly vhodnými kandidáty z pohledu zvládání velkého objemu dat, avšak je u nich výrazně omezena možnost analytických dotazů potřebných pro implementaci cílové funkcionality.

Elasticsearch a další zástupci dokumentových NoSQL databází s většími analytickými možnostmi lze považovat za možnou volbu. Nicméně při různých pohledech na data, které se mohou při předzpracování a reportování objevit, jako například práce s daty z pohledu různých objektů (adresy „účastnící“ se transakce nebo naopak transakce, kterých se adresu „zúčastnila“), by hrozily problémy s datovým modelem a mohla by se objevit duplikace dat.

Neo4j jakožto grafová databáze by měla být rovněž vhodnou volbou, jelikož umožní de facto nativně modelovat vstupní data, která ve své podstatě tvoří graf. Stejně tak po analytických transformacích by mělo být vždy možné výsledek reprezentovat jako graf, jelikož oba hlavní případy užití pracují s cestami a sousedními uzly nebo shluky.

3.3 Grafová databáze

Formát vstupních dat i podoba požadovaných výsledků dotazů naznačují, že podstata problému je snadno formulovatelná v grafové podobě. Pro implementaci byla z tohoto důvodu zvolena jako nejvhodnější možnost databáze Neo4j, která je zdaleka nejpoužívanější grafovou databází [5]. Zároveň tato databáze dle dokumentace podporuje neomezený počet uzlů, takže by měla být schopna pojmout zvětšující se objem záznamů obsažených v blockchainu.

V sekci 5.1 je uvedeno, jakým způsobem jsou objekty z blockchainu namapovány do Neo4j. Zároveň tato databáze podporuje všechny potřebné operace rozebírané v sekci 3.1, takže je možné v ní provádět zmiňované importovací, předzpracovací i reportovací dotazy.

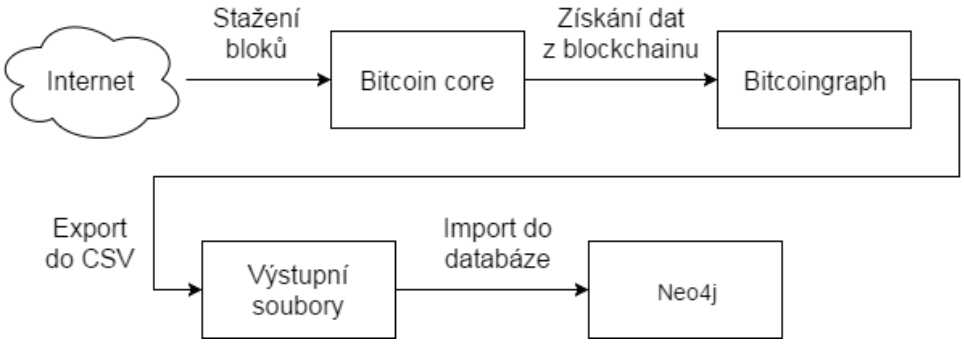
Kapitola 4

Získávání dat

V této kapitole je rozebrán postup, jakým lze z blockchainu získat data potřebná pro analýzu. Postup se skládá ze dvou částí:

- Stažení dat z blockchainu – tato část je nastíněna v sekci 4.1
- Naimportování stažených dat do grafové databáze – touto částí se zabývá sekce 4.2

Nad takto získanými daty jsou následně spouštěny analytické operace, které jsou popsány v následující kapitole 5. Celkový průběh postupu pro získání dat je ilustrován na obrázku 4.1.



Obrázek 4.1: Schema průběhu získávání dat z blockchainu a jejich importu do neo4j

4.1 Export blockchainu

V této sekci je popsán postup, jakým je možné získat data z blockchainu používaného k platbám bitcoiny.

Některé aspekty postupu jsou specifické pro různé cílové platformy a vzhledem k tomu, že implementace praktické části této práci probíhala na platformě Windows, jsou v postupech popsané i některé rozdíly oproti Linuxovým platformám, pro které jsou návody dostupné online primárně určeny. Spolu s těmito rozdíly jsou zdokumentovány i některé platformně závislé problémy, které bylo nutné během práce řešit.

4.1.1 Bitcoin core

Bitcoin core je program, který je volně dostupný ke stažení[4] z webu bitcoin.org pro všechny běžné počítačové platformy. Jeho použitím je zajištěno, že uživatel, resp. jeho počítač, přijímá pouze validní transakce ze správného blockchainu [6].

4.1.1.1 O programu

Tento program je schopen vykonávat například funkci peněženky, popsané v kapitole 2. V rámci použití za účelem exportu dat z blockchainu však stačí to, že je možné pomocí něj stahovat transakce z decentralizované sítě uživatelů bitcoinu.

Po stažení, instalaci a spuštění je k dispozici GUI, ve kterém může uživatel sledovat stav své peněženky a provádět platby (ilustrováno na obrázku 4.2). Pro účely exportu dat je důležité, že bitcoin core začne automaticky stahovat blockchain, takže jsou z internetu od ostatních uzlů v síti postupně získávány bloky s transakcemi.

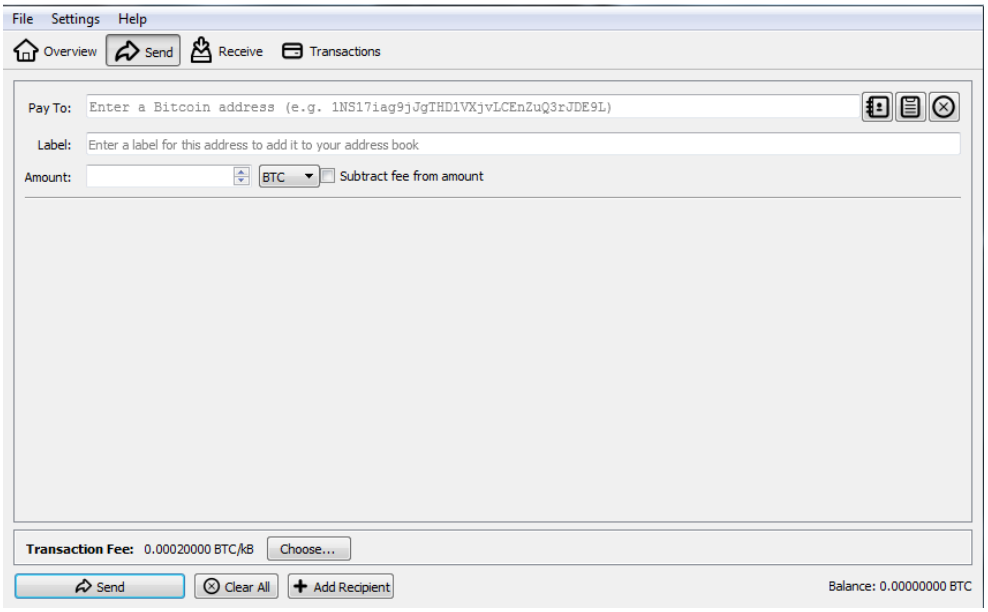
4.1.1.2 Nastavení parametrů

Dalším krokem je nastavení bitcoin core tak, aby v něm bylo povoleno RPC[5] a bylo možné používat RESTové rozhraní[6]. K docílení tohoto je nutné upravit konfigurační soubor bitcoin core dle popisu v dokumentaci nástroje bitcoin-graph [7].

Pro export jsou podstatné hodnoty těchto parametrů:

- `server` – nastavení na `1` povolí přijímání RPC
- `rpcuser` – uživatelské jméno
- `rpcpassword` – heslo pro ověření

[4] https://bitcoin.org/en/download
[5] RPC – Remote procedure call
[6] REST – Representational state transfer



Obrázek 4.2: Rozhraní bitcoin core, obrazovka pro odeslání platby

- `rpcclienttimeout` – maximální počet sekund, kolik může trvat požadavek (základní hodnota je `30`)[7]
- `rpcport` – port, na kterém bude bitcoin core poslouchat pro příchozí spojení (základní hodnota je `8332`)
- `txindex` – nastavení na `1` povolí index pro transakce mimo uživatelovu vlastní peněženku

Důležité je nezapomenout změnit v konfiguračním souboru hlavně nastavení `rpcuser` a `rpcpassword`, čímž je zajištěno, že rozhraní bitcoin core bude dostupné pouze po ověření správného uživatelského jména a hesla.

Nastavení, které povolí RESTové rozhraní, lze předat jak pomocí parametru `-rest` při spouštění z konzole, tak i přidáním patřičného parametru do konfiguračního souboru bitcoin core. V případě druhé metody je nutné přidat do konfiguračního souboru tento řádek:

```
rest=1
```

Finální podoba upravených hodnot v konfiguračním souboru může vypadat například jako v následující ukázce:

[7] V návodu nástroje bitcoingraph je uveden parametr `rpctimeout`, avšak v době vzniku práce je tento parametr v základní bitcoin core instalaci pro Windows nazván `rpcclienttimeout`.

```
server=1

rpcuser=uzivatelskeJmeno
rpcpassword=tajneHeslo

rpcclienttimeout=30
rpcport=8332

txindex=1
rest=1
```

Následně je možné nechat bitcoin core stahovat bloky z blockchainu a pak přistoupit k exportu dat do formátu, který potom bude použit pro import do grafové databáze.

4.1.2 Bitcoingraph

Poté, co bitcoin core stáhne potřebné množství bloků, lze přistoupit k druhé fázi exportu, kterou je použití nástroje *bitcoingraph*. Tento nástroj je volně ke stažení[8] a umožňuje transformovat data získaná z blockchainu do formátu, který je možné importovat do grafové databáze Neo4j.

4.1.2.1 Instalace

Tento nástroj vyžaduje ke své instalaci a spuštění Python verze 3.4, takže je nutné, aby byl k dispozici na stroji, kde bude export proveden. Samotná instalace probíhá tak, že je naklonován GIT repozitář a následně je pro instalaci použit Python.

Postup lze shrnout do následujících příkazů v konzoli:

```
$ git clone https://github.com/behas/bitcoingraph.git

$ cd bitcoingraph
$ pip install -r requirements.txt
$ py.test
$ python setup.py install
```

Po instalaci by měly být ve složce `scripts` dostupné tyto soubory:

- `bcgraph-compute-entities`
- `bcgraph-export`
- `bcgraph-synchronize`

[8] https://github.com/behas/bitcoingraph

4.1.2.2 Použití

Prvním nástrojem, který je z bitcoingraphu nutné použít, je `bcgraph-export`. Ten zajistí, že z blockchainu je získán stanovený rozsah bloků a lze je pak dále zpracovávat. Pro použití je třeba, aby byl spuštěn bitcoin core a byl správně nastaven. Kroky týkající se přípravy bitcoin core a jeho nastavní jsou popsány v sekci 4.1.1.

Nástroj lze spustit následujícím příkazem:

```
bcgraph-export 0 <max> -u <uzivatel> -p <heslo>
```

Části příkazu uzavřené mezi `<` a `>` jsou parametry závisející na nastavení bitcoin core, jejichž význam je popsán v seznamu níže.

- `max` – Maximální *výška* bloku, který bude exportován
- `uzivatel` – Uživatelské jméno (parametr `rpcuser` v konfiguračním souboru) nastavené v bitcoin core
- `heslo` – Heslo (parametr `rpcpassword` v konfiguračním souboru) nastavené v bitcoin core

Po dokončení exportu lze nalézt ve složce `blocks_0_<max>`, kde `<max>` je maximální ID bloku použité výše, výstupní soubory, které obsahují data exportovaná z blockchainu ve formátu CSV.

Ke každému souboru je navíc vygenerován ještě hlavičkový soubor (např. `addresses_header.csv` pro `addresses.csv`), ve kterém jsou uvedeny názvy jednotlivých sloupců a jejich datové typy.

Výstupní soubory a vysvětlení jejich obsahů jsou uvedeny v následujícím seznamu, kde je pro každý soubor vypsán seznam sloupců, které se v něm nachází.

- `addresses.csv` – seznam adres použitých v transakcích patřících do exportovaných bloků
  - `address`: identifikátor adresy
- `blocks.csv` – seznam bloků
  - `hash`: *hash* daného bloku
  - `height`: *výška* bloku, tj. jeho „pořadí“
  - `timestamp`: časová značka určující vznik bloku
- `transactions.csv` – seznam transakcí patřících do exportovaných bloků
  - `txid`: *hash* transakce

  - `coinbase`: booleovská hodnota, která značí, jestli je daná transakce *coinbase* nebo ne

- `outputs.csv` – seznam *outputů* z transakcí
  - `txid_n`: *hash* transakce spolu s identifikátorem *outputu*
  - `n`: identifikátor *outputu*
  - `value`: hodnota *outputu* (v BTC)
  - `type`: typ *outputu*
- `rel_block_tx.csv` – vazby mezi bloky a transakcemi
  - `hash`: *hash* bloku, do kterého *transakce* náleží
  - `txid`: *hash* dané transakce
- `rel_input.csv` – vazby mezi transakcemi a jejich vstupními *outputy*
  - `txid`: *hash* cílové transakce
  - `txid_n`: *hash* zdrojové transakce spolu s identifikátorem *outputu*
- `rel_output_address.csv` – vazby mezi *outputy* a adresami
  - `txid_n`: *hash* transakce spolu s identifikátorem *outputu*
  - `address`: identifikátor adresy, které byl přidělen daný *output*
- `rel_tx_output.csv` – vazby mezi transakcemi a výstupními *outputy*
  - `txid)`: *hash* transakce
  - `txid_n`: *hash* transakce spolu s identifikátorem *outputu*

Dalším krokem je po exportování dat z blockchainu identifikovat *entity*, kterým bude v databázi přiřazena jedna nebo více adres použitých v zadaném rozsahů bloků.

Identifikaci *entit* lze provést následujícím příkazem:

```
$ bcgraph-compute-entities -i blocks_0_<max>
```

Parametr `<max>` zde opět značí maximální zvolenou *výšku* bloku, která byla exportována. Jako argument volání skriptu `bcgraph-compute-entities` je tedy předána složka, ve které se nachází výstupní soubory z předešlého exportu. Výstupem tohoto skriptu jsou dva další soubory ve složce `<max>`, které obsahují záznamy opět v CSV formátu a jejich význam je rozepsán v seznamu níže. K těmto souborům již nejsou vygenerovány další hlavičkové soubory, ale hlavičky jsou vždy vloženy přímo na prvním řádku daného souboru.

- `entities.csv` – seznam identifikátorů entit

  - `id`: identifikátor dané entity

- `rel_address_entity.csv` – vazby mezi adresami a entitami
  - `address`: identifikátor adresy
  - `id`: identifikátor entity, které byla daná adresa přiřazena

Po skončení skriptu `bcgraph-compute-entities` jsou již připravena všechna data, která lze importovat do grafové databáze Neo4j. Importu těchto připravených dat se věnuje sekce 4.2.

4.1.2.3 Specifika pro Windows

Po rozebrání postupu exportu dat z blockchainu jsou v této sekci ještě popsány rozdíly v procesu exportu při použití platformy Windows, které byly zmíněny na začátku sekce 4.1.

Pro použití na platformě Windows je vhodné použít Cygwin, což je sada programů simulujících chování Linuxových systémů. Předpokladem následujících instrukcí je, že příkazy popsané v sekci 4.1.2.2 jsou spouštěny právě v doporučeném nástroji Cygwin.

Prvním problémem při provádění exportu na platformě Windows je volání konzolových programů z Python scriptu pomocí `subprocess.call`, které způsobuje, že nástroj bitcoingraph není s Windows kompatibilní. To lze napravit úpravou souboru `helper.py` ve složce `bitcoingraph`, ve kterém se toto volání nachází. Použito je zde na seřazení souboru (program `sort`) a funkce, která jej využívá, je tato (odřádkování je upraveno kvůli omezenému prostoru):

```
def sort(path, filename, args=''):
    if sys.platform == 'darwin':
        s = 'LC_ALL=C gsort -S 50% --parallel=4 {0} {1} -o {1}'
    else:
        s = 'LC_ALL=C sort -S 50% --parallel=4 {0} {1} -o {1}'
    status = subprocess.call(
        s.format(args, os.path.join(path, filename)), shell=True)
    if status != 0:
        raise Exception('unable to sort file: {}'.format(filename))
```

Vyřešit lze tento problém nahrazením volání `subprocess.call` tak, že používá Cygwin. Do řádku

```
s = 'LC_ALL=C sort -S 50% --parallel=4 {0} {1} -o {1}'
```

stačí přidat nejprve volání programu `bash`, ve kterém se následně spustí požadovaný `sort` a kód pak již pracuje správně. Řádek by tedy měl po úpravě vypadat například takto (odřádkování je opět upraveno kvůli omezenému prostoru):

```
s = 'C:\\cygwin64\\bin\\bash.exe -c
    \'LC_ALL=C sort -S 50\% --parallel=4 {0} {1} -o {1}\''
```

Následně je nutné ve složce, kam byl nainstalován bitcoingraph, spuštěním příkazu

```
$ python setup.py install
```

zajistit, aby se změna projevila v nainstalovaných skriptech.

Druhým problémem jsou rozdíly mezi Windows a Linux v sekvencích znaků pro ukončení řádku, které způsobují, že skript `bcgraph-compute-entities` chybně vyhodnocuje počty řádků v souborech vygenerovaných pomocí skriptu `bcgraph-export`. To lze opravit použitím programu `dos2unix`, který je dostupný v prostředí Cygwin. Ten zajistí, že po spuštění příkazu

```
$ dos2unix *
```

ve složce `blocks_0_<max>`, která obsahuje výstupní soubory skriptu `bcgraph-export`, jsou konce řádků správné a skript `bcgraph-compute-entities` zpracuje soubory správně.

4.2 Import do Neo4j

Po exportu dat z blockchainu a jejich přípravě do formátu CSV, což jsou úkony, jimiž se zabývala sekce 4.1, lze přejít k importu těchto dat do grafové databáze Neo4j.

Předpokladem je pochopitelně mít na daném počítači tuto databázi nainstalovanou, ale prozatím musí být vypnutá. Dále je pak nutné mít připraven adresář, do kterého se budou data importovat. Pro příklad bude použit ukázkový název adresáře `data.db`, v praxi je možné si zvolit jakýkoliv, avšak musí být prázdný.

Z adresáře, kde jsou dostupné výstupní soubory exportovacích skriptů, je pak spuštěn tento příkaz:

```
$ neo4j-import --into data.db \
--nodes:Block blocks_header.csv,blocks.csv \
--nodes:Transaction transactions_header.csv,transactions.csv \
--nodes:Output outputs_header.csv,outputs.csv \
```

```
--nodes:Address addresses_header.csv,addresses.csv \
--nodes:Entity entities.csv \
--relationships:CONTAINS rel_block_tx_header.csv,rel_block_tx.csv \
--relationships:OUTPUT rel_tx_output_header.csv,rel_tx_output.csv \
--relationships:INPUT rel_input_header.csv,rel_input.csv \
--relationships:USES rel_output_address_header.csv,rel_output_address.csv \
--relationships:BELONGS_TO rel_address_entity.csv
```

Po doběhnutí příkazu jsou data importována do složky `data.db` a z ní je může číst Neo4j. Ověřit to lze spuštěním Neo4j a kontrolou, že ve webovém rozhraní, které je po spuštění dostupné (v základním nastavení na adrese `localhost:7474`), jsou data dostupná.

Pro potvrzení, že import proběhl v pořádku, je možné použít například dotaz

```
MATCH(b:Block) RETURN count(b)
```

který by měl vrátit číslo o jedna vyšší, než kolik byla maximální *výška* bloku stanovená parametrem `<max>` v sekci 4.1.2.2. Označení *výšky* bloku začíná na hodnotě `0`, exportovaných bloků je proto o jeden více, než jaká je maximální exportovaná *výška*.

Kapitola 5

Zpracování dat

V této kapitole jsou popsány operace, které vedou k přípravě dat pro analytické dotazy uživatelů. Tyto operace jsou spouštěny pomocí Java aplikace, jejíž implementace je popsána v kapitole 6.

Nejprve je zde rozebrán datový formát, který je v Neo4j dostupný po dokončení získávání dat popsaném v kapitole 4. Výchozímu formátu dat se věnuje sekce 5.1, následným operacím nad daty pak sekce 5.2 a 5.3.

5.1 Formát importovaných dat

Výstupem importu jsou uzly a hrany v grafové databázi Neo4j, se kterými lze pracovat pomocí běžných dotazů pro tuto databázi. V následujících částech jsou zdokumentovány vlastnosti těchto objektů a jejich význam z hlediska blockchainu. Pro snadnější orientaci jsou v části 5.1.3 uzly a hrany popsány na příkladu.

5.1.1 Uzly

V následujícím seznamu jsou uzly vytvořené v databázi po importu, spolu s definicemi toho, co jednotlivé uzly reprezentují:

- `Address` – adresa použitá v blockchainu
- `Block` – jeden blok z blockchainu
- `Entity` – entita vlastnící jednu nebo více adres
- `Output` – výstup transakce
- `Transaction` – jedna transakce z určitého bloku

Význam uzlů `Address`, `Block`, `Output` a `Transaction` je zřejmý, nejzajímavějším objektem je zde proto uzel `Entity`. Ten označuje určitou *entitu*, která

v některé z exportovaných částí blockchainu použila více různých adres jako zdroj pro *outputy* vstupující do jedné transakce a exportovací nástroj dle toho usoudil, že dané adresy patří do jedné peněženky, resp. náleží nějaké *entitě*, která je ovládá [8].

5.1.2 Hrany

Mezi uzly importovanými do Neo4j jsou rovněž vytvořené hrany, které reprezentují určité vazby mezi danými objekty. Přehled typů hran a příslušných reprezentací je v seznamu níže. Rovněž je zde vždy v závorce uvedeno, mezi kterými uzly se tato hrana v grafu vyskytuje.

- `BELONGS_TO` (`Address`→`Entity`) – adresa patří k entitě
- `CONTAINS` (`Block`→`Transaction`) – blok obsahuje transakci
- `INPUT` (`Output`→`Transaction`) – output je vstupem do transakce
- `OUTPUT` (`Output`→`Transaction`) – output je výstupem z transakce
- `USES` (`Output`→`Address`) – output používá adresu

Podrobněji zde budou rozebrány dvě vazby, `BELONGS_TO` a `USES`, význam ostatních je zřejmý.

Vazba `BELONGS_TO` reprezentuje fakt, že daná adresa náleží dané entitě ve smyslu, který byl popsán spolu s uzlem `Entity` v sekci 5.1.1.

Vazba `USES` značí, že daný output byl „přiřazen“ dané adrese, tj. v transakci je uveden příslušný *hash* adresy. To v praxi znamená, že adresa nejprve obdržela platbu v transakci, která má v grafu na daný output vazbu `OUTPUT`. Následně pak tento output mohl být použit v další transakci. V tomto případě by měl vazbu `INPUT` na transakci, do které vstoupil.

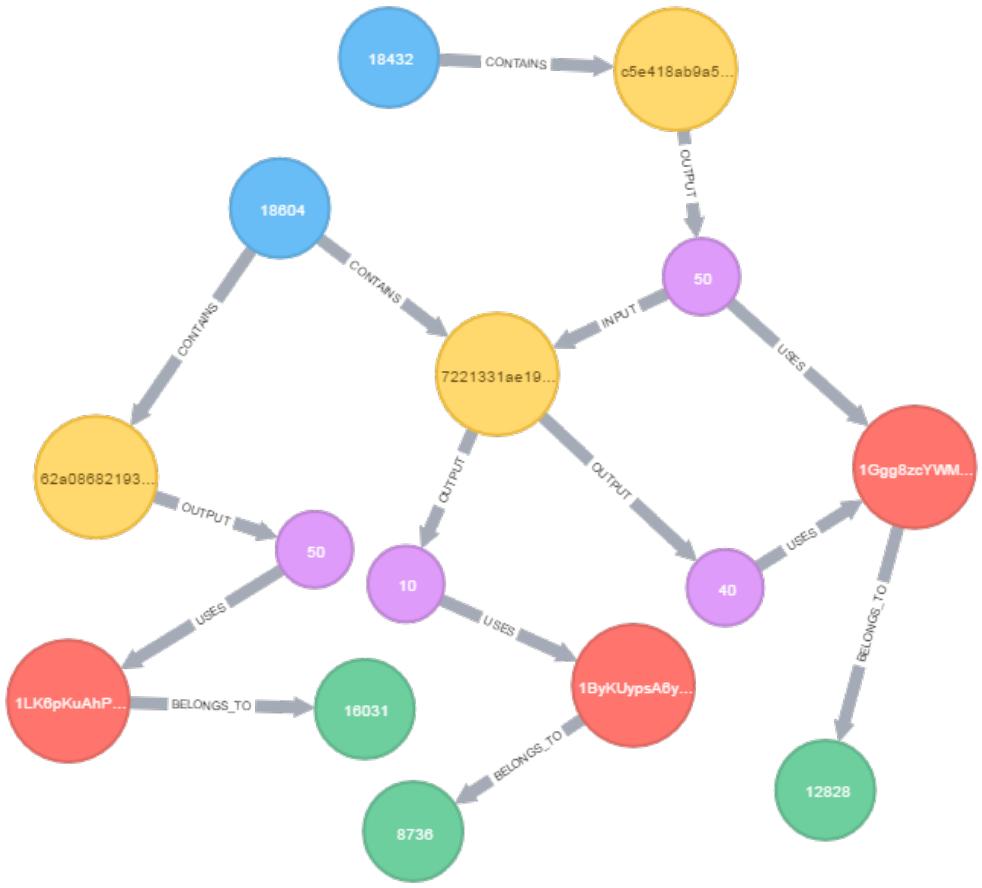
5.1.3 Ukázková data

Na obrázcích 5.1 a 5.2 níže jsou ilustrovány ukázky dat importovaných do Neo4j.

5.1.3.1 Základní struktura

V ukázce jsou zobrazeny dva bloky (modře, v uzlu *výška* bloku), tři transakce (žlutě, v uzlu *hash* transakce), tři adresy (červeně, v uzlu *identifikátor* adresy), tři entity (zeleně, v uzlu *identifikátor* entity) a čtyři outputy (fialově, v uzlu *value* outputu). Některé popisky uzlů jsou zkráceny z důvodu jejich přílišné délky, avšak všechny uzly jsou na základě nich vzájemně rozlišitelné.

Nejprve bude rozebrána transakce, která je na obrázku 5.1 označena jako `62a08682193`. Jelikož do transakce nevstupuje žádná hrana typu `INPUT`, jedná



Obrázek 5.1: Ukázka formátu dat importovaných do Neo4j

se o *coinbase* transakci. Tato transakce produkuje 50 BTC pro adresu `1LK6pKuAhP`, která náleží entitě `16031`.

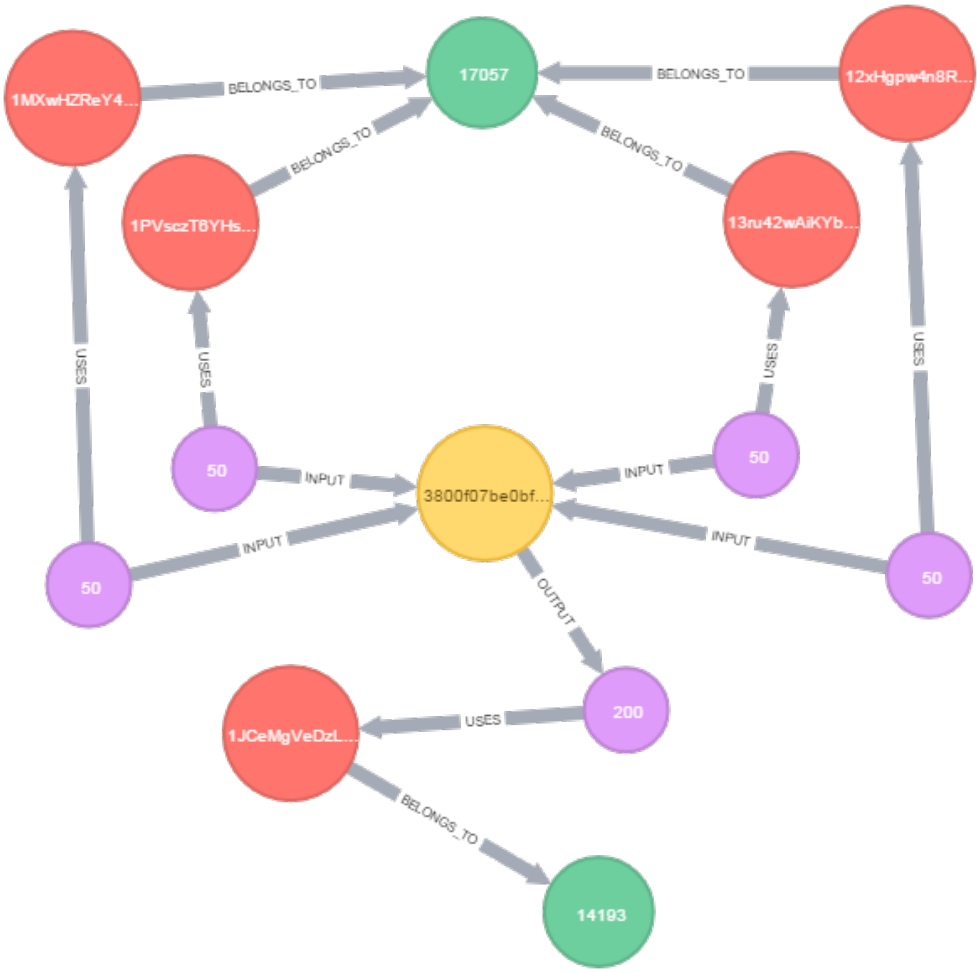
Další transakce pro rozbor je na obrázku 5.1 označena jako `7221331ae19`. Tato transakce používá jako `INPUT` output z transakce `c5e418ab9a5`, který má hodnotu 50 BTC a je přiřazen adrese `1Ggg8zcYWM` náležící entitě `12828`. Výstupy z této transakce jsou dva, jeden zpět na adresu `1Ggg8zcYWM` (s hodnotou 40 BTC) a druhý na adresu `1ByKUypsA6y` (s hodnotou 10 BTC). V praxi tedy nastalo následující:

- Na adresu `1Ggg8zcYWM` byla přijata platba 50 BTC,
- z této adresy bylo odesláno 10 BTC na adresu `1ByKUypsA6y`
- a zbylých 40 BTC bylo odesláno zpět na adresu `1Ggg8zcYWM`

5.1.3.2 Entita s více adresami

Na obrázku 5.2 je zobrazena entita s identifikátorem `17057`, ke které náleží více než jedna adresa. Tento případ v první ukázce, na obrázku 5.1, nenastal. Všechny adresy na obrázku (červeně) jsou přiřazeny této entitě, jelikož outputy (všechny s hodnotou 50 BTC) z nich vstupují zároveň do jedné transakce (žlutě) a jsou tak dle popisu entit v sekci 5.1.1 považovány za centrálně používané.

Zdrojové transakce, ze kterých outputy pochází, jsou v obrázku pro přehlednost vynechány. Výstupem této transakce je 200 BTC pro adresu `1JCeMgVeDzL`, která již patří jiné entitě (`14193`).



Obrázek 5.2: Ukázka entity s více adresami vytvořené v Neo4j

5.2 Transformace grafových objektů

Transformační operace prováděné nad importovaným grafem jsou popsány v této sekci. Lze je rozdělit na několik procedur, které na sebe navazují a jsou zdokumentovány v jednotlivých podsekcích níže.

Základem těchto operací je doplnění užitečných atributů k uzlům a přidání hran tak, aby bylo možné se v informacích v grafu snadněji orientovat a jednotlivé objekty lépe reprezentovaly informace, na které se mohou uživatelé ohledně grafu ptát. Jinými slovy, jde částečně o jakýsi „překlad“ z velmi technicky zformované podoby dat získané z blockchainu do intuitivnější podoby, která je dodává do grafu větší nadhled a umožňuje tak jednodušeji vyčíst znalosti v něm skryté.

5.2.1 Vytvoření indexů

Pro výraznou optimalizaci rychlosti v dalších operacích je nutné nejprve vytvořit v databázi indexy pro některé atributy uzlů, podle kterých je následně v průběhu zpracování vyhledáváno. Jde o atribut `txid` pro uzly typu `Transaction` a atribut `id` pro uzly typu `Cluster`. První index umožňuje rychle vyhledávat transakce podle jejich identifikačního *hashe*, druhý pak shluky podle jejich *id*, čehož bude využito při následném shlukování popsaném v sekci 5.3.

Na rozdíl od volání databáze Neo4j přímo z Javy, které je využito v ostatních případech a jehož použití je zdokumentováno v kapitole 6, je zde použit dotaz v jazyce Cypher, což je dotazovací jazyk pro Neo4j.

Použití je snadné, zavoláním příkazu

```
CREATE INDEX ON :Transaction(txid)
```

je vytvořen index pro transakce a zavoláním

```
CREATE INDEX ON :Cluster(id)
```

pak také pro shluky.

5.2.2 Doplnění časových značek do transakcí

Transakce nemají ve svých atributech časové značky pro indikaci, kdy proběhly. Tato informace je však obsažena v blocích a při dalších operacích je užitečné ji mít obsaženou i v atributech transakcí.

Tato procedura proto prochází všechny bloky, přičemž pro každý z nich doplní ke všem transakcím v něm obsaženým stejnou časovou značku, jako má daný blok.

5.2.3 Označení coinbase transakcí

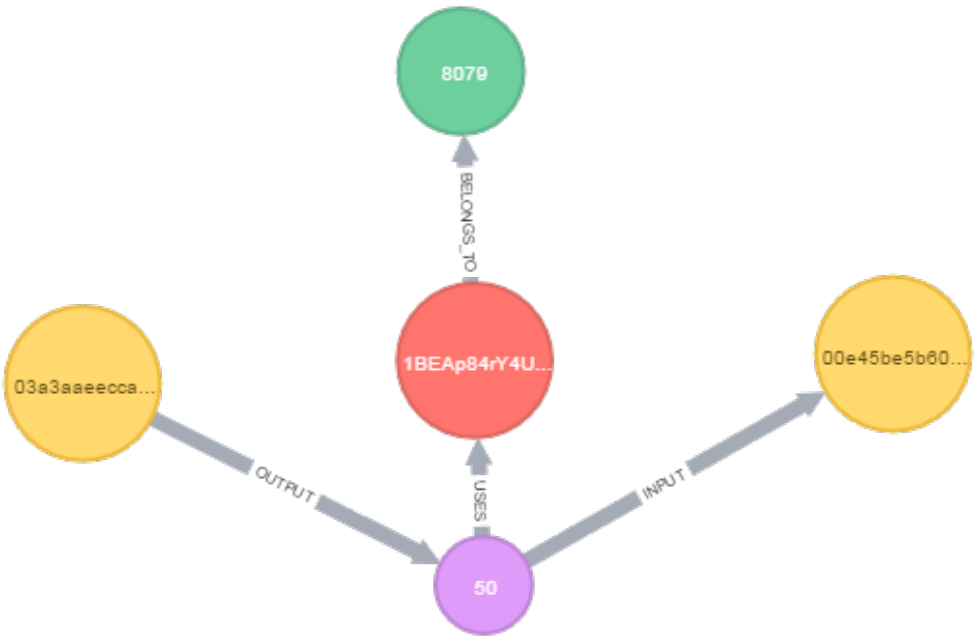
Pro uživatele, který se bude snažit blockchain analyzovat, pravděpodobně nebudou příliš zajímavé transakce, které vznikly během těžení. Z tohoto důvodu by měl mít možnost je z analýzy vynechat a pracovat jen s těmi, jež používají již dříve vytěžené bitcoiny.

Tato procedura zajistí, že všechny *coinbase* transakce jsou označeny pomocí booleovského atributu `coinbase` v uzlu `Transaction`. Coinbase transakce lze identifikovat tak, že do nich nevstupuje žádná hrana `INPUT` z uzlu `Output`, tj. nemají na vstupu žádné *outputy*. Všem takovým transakcím je pak nastaven atribut `coinbase` na `true`, zatímco u ostatních je v tomto atributu uložena hodnota `false`.

5.2.4 Propojení transakcí s adresami a entitami

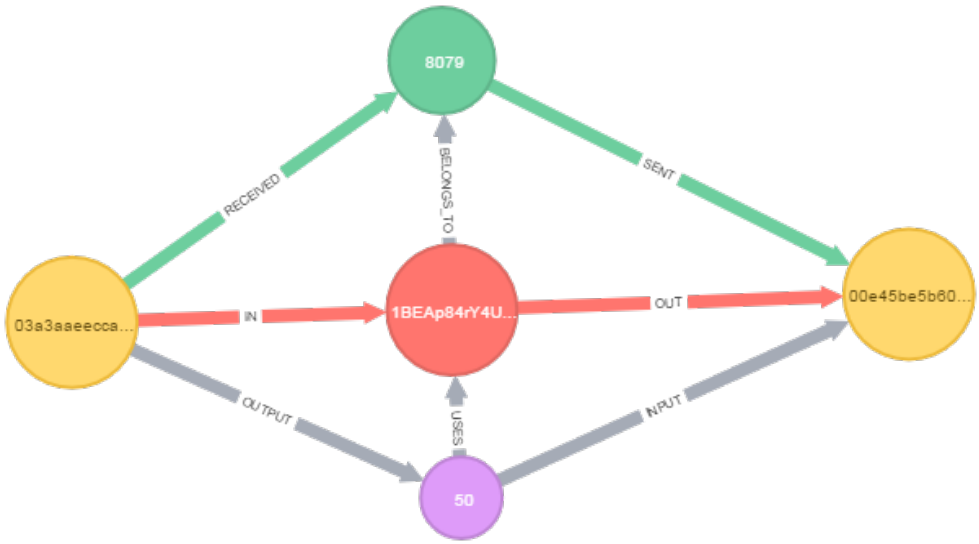
Dalším krokem je tranzitivní přenesení informace z vazeb mezi transakcí, outputem, adresou a entitou, na přímé vazby spojující transakci s adresou a s její entitou. To umožní získat lepší a intuitivnější přehled o transakcích a pohybech bitcoinů mezi adresami, resp. entitami. Tyto nově vytvořené hrany pak využívají další procedury, které pomocí nich doplňují další informace.

Změny jsou patrné z rozdílů mezi obrázky 5.3 a 5.4, které znázorňují objekty v databázi před a po provedení této procedury.



Obrázek 5.3: Ukázka stavu databáze před procedurou propojující transakce s adresami a entitami

Z obrázku 5.3 je zřejmé, že pro získání informace, která adresa použila výstup z transakce `03a3aaeecca` (vlevo) v transakci `00e45be5b60` (vpravo), je nutné „projít“ přes daný uzel `Output`. Stejně tak je tomu v případě dotazu na entitu, která tuto akci provedla pomocí jedné ze svých adres.



Obrázek 5.4: Ukázka stavu databáze po proceduře propojující transakce s adresami a entitami

Oproti tomu ve stavu po dokončení procedury, který je ilustrován na obrázku 5.4, jsou již oba objekty, adresa (červeně) i entita (zeleně), spojeny vazbou s příslušnými transakcemi. Lze tedy z grafu přímo a intuitivně zjistit, které adresy a entity se zapojily do daných transakcí. Tato úprava sama o sobě umožňuje uživateli jednak získat lepší přehled o transakcích kvůli zjednodušení orientace, ale také poskytuje hrany užitečné pro následující procedury.

Nově doplněné hrany jsou označeny `IN` pro příchozí transakci, resp. `OUT` pro odchozí, v případě adresy a `RECEIVED`, resp. `SENT` v případě entity. Navíc je do atributu `value` těchto hran přidána hodnota příslušného outputu a do atributu `timestamp` časová značka, která je v příslušné transakci, aby bylo možné tyto hrany snadno filtrovat podle času.

5.2.5 Výpočet zůstatků pro adresy a entity

Po doplnění hran `IN` a `OUT`, které bylo zajištěno předchozí procedurou, lze spočítat, kolik ještě „volných“ BTC je k dispozici pro každou adresu, resp. entitu.

Pro adresy lze spočítat tyto zůstatky, označené jako atribut `balance`, dle pseudokódu 1, kde *addresses* značí všechny adresy v grafu, $addr.rels(X)$

funkci vracející hrany typu X spojené s adresou, *input.value*, resp. *output.value* hodnotu daného *outputu* a *addr.balance* atribut `balance` pro danou adresu.

Pseudokód 1 Výpočet zůstatku pro adresy

```
for all addr in addresses do
  sum = 0;
  for all input in addr.rels(IN) do
    sum = sum + input.value;
  end for
  for all output in addr.rels(OUT) do
    sum = sum − output.value;
  end for
  addr.balance = sum;
end for
```

Následně lze získat zůstatky pro entity sečtením zůstatků všech adres, které do dané entity patří, jak je znázorněno v pseudokódu 2, kde *entities* reprezentuje všechny entity v grafu, $entity.rels(X)$ opět funkci vracející hrany typu X spojené s entitou a *rel.neighbor* sousední uzel, který je s entitou spojen hranou *rel*.

Pseudokód 2 Výpočet zůstatku pro entity

```
for all entity in entities do
  sum = 0;
  for all rel in entity.rels(BELONGS_TO) do
    addr = rel.neighbor;
    sum = sum + addr.balance;
  end for
  entity.balance = sum;
end for
```

5.2.6 Zjednodušení transakcí pro entity

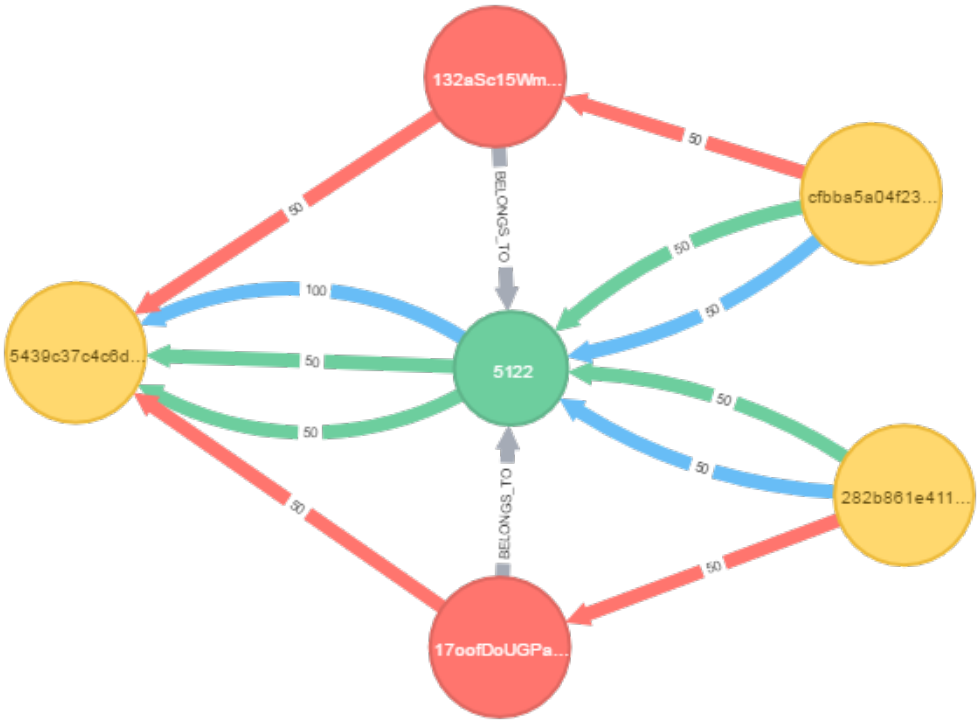
Předchozí procedury zajistily, že jsou pro každou entitu v grafu k dispozici hrany `RECEIVED` a `SENT`, které každá vyjadřují účast entity v transakci.

Mohou však nastat situace, kdy je entita navázaná na více těchto hran pro stejnou transakci, jako je tomu v případě ilustrovaném na obrázku 5.5. V tomto případě má entita `5122` dvě hrany `RECEIVED`, což je v pořádku, protože každá vychází z jiné transakce. Ale jsou zde i dvě hrany `SENT`, které jsou navázány obě na stejnou transakci, což již není intuitivně dobře pochopitelný stav. Tato situace vychází z toho, že daná entita použila v jedné transakci dvě své adresy (na obrázku červeně), takže pro každou z nich jí byla přiřazena hrana `SENT` do příslušné transakce.

Obrázek 5.5: Ukázka stavu databáze před sečtením vstupů a výstupů transakcí pro entity

Situaci řeší tato procedura tak, že pro entity spočítá vždy součet atributů `value` hran `SENT`, resp. `RECEIVED`, pro jednotlivé transakce, kterých se daná entita účastnila. Tyto součty jsou uloženy do pomocných hran `TOTAL_SENT`, resp. `TOTAL_RECEIVED`, které vyjadřují celkový součet BTC odeslaných z nebo přijatých do entity v rámci transakce, na kterou jsou navázány.

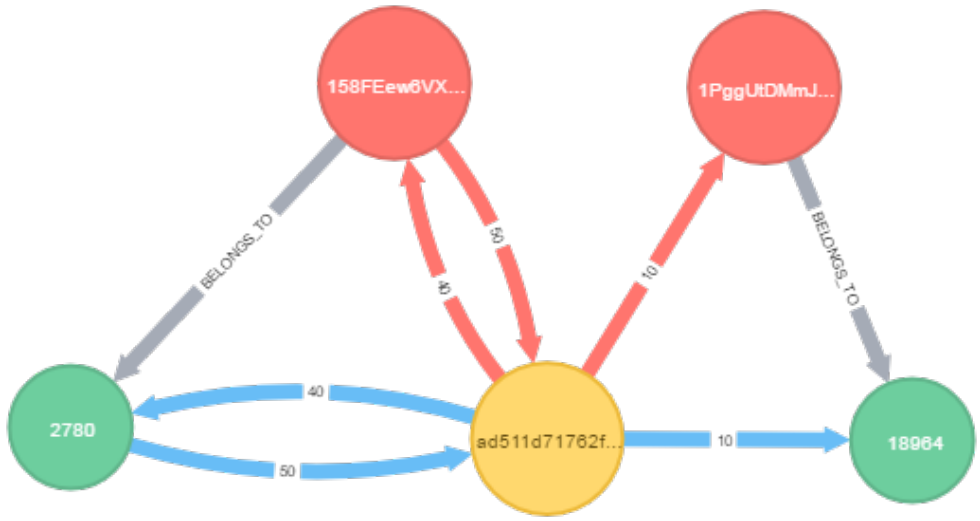
Příklad výstupu procedury je na obrázku 5.6, kde jsou u hran `IN`/`OUT` (červeně), `RECEIVED`/`SENT` (zeleně) a `TOTAL_RECEIVED`/`TOTAL_SENT` (modře) vyměněny v popiscích typy hran za hodnoty jejich atributů `value`, aby bylo lépe patrné sčítání mezi nimi. Ke dvěma hranám `SENT` s hodnotou 50 BTC byla v tomto případě přidána jedna hrana `TOTAL_SENT` s hodnotou 100 BTC tak, aby hrany „odesílající“ z entity bitcoiny do transakce `5439c37c4c6d` vlevo mohly být v budoucích dotazech a procedurách touto hranou nahrazeny.



Obrázek 5.6: Ukázka stavu databáze po sečtení vstupů a výstupů transakcí pro entity

Posledním krokem pro zpřehlednění přesunů BTC mezi entitami a transakcemi je výpočet výsledného pohybu napříč zúčastněnými adresami dané entity. Mohou totiž nastávat případy, jako např. na obrázku 5.7, kdy jedna entita do jedné transakce bitcoiny posílá a zároveň je z ní přijímá. Tato situace nastane tak, že některý output vstupuje do transakce z jedné adresy patřící entitě a část jeho hodnoty z této transakce vystupuje v podobě nového outputu do adresy patřící stejné entitě. Pak dochází k tomu, že tato entita je s danou transakcí spojena hranou typu `TOTAL_SENT` i hranou typu `TOTAL_RECEIVED` a reálný obnos ani směr jeho pohybu, který byl přesunut, tak není na první pohled zřejmý a bylo by nutné ho dopočítávat.

Tato procedura proto doplňuje kromě součtů také výsledné rozdíly, které jsou označeny hranami `BALANCE_IN` nebo `BALANCE_OUT` (podle směru pohybu obnosu) a reprezentují celkovou bilanci dané entity v rámci jedné transakce.

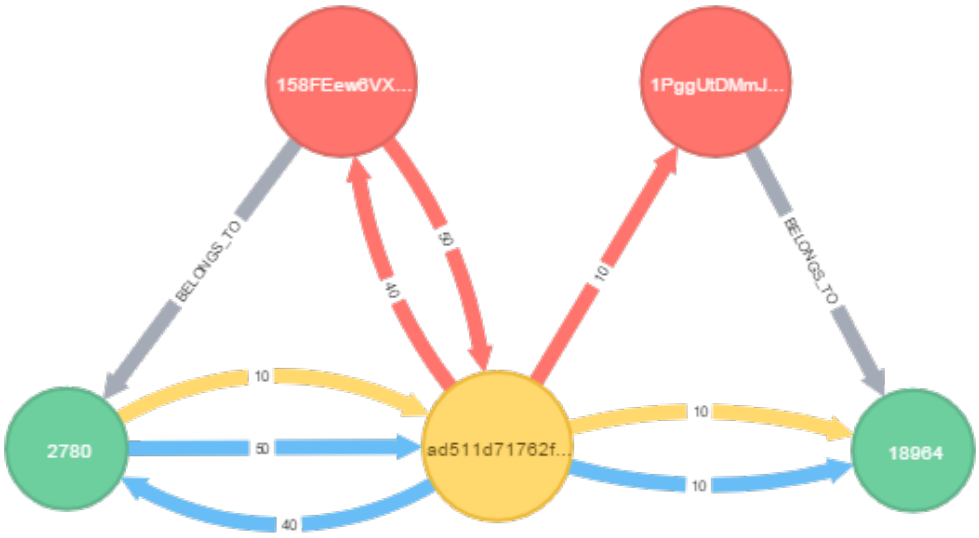


Obrázek 5.7: Ukázka stavu databáze před výpočtem bilancí transakcí pro entity (některé hrany jsou pro přehlednost vynechány)

Na obrázku 5.8 je ukázka toho, jak byly hrany `TOTAL_SENT` s hodnotou 50 (modře, směrem od entity) a `TOTAL_RECEIVED` s hodnotou 40 (modře, směrem k entitě) vlevo doplněny o hranu `BALANCE_OUT` s hodnotou 10 (žlutě, směrem od entity `2780`), která je rozdílem jejich hodnot.

Rovněž pro hranu `TOTAL_RECEIVED` vpravo s hodnotou 10 (modře, směrem k entitě) byla přidána hrana `BALANCE_IN` s hodnotou 10 (žlutě, směrem k entitě `18964`), která je zde doplněna z toho důvodu, aby bylo možné se následně dotazovat na tyto pohyby zůstatku mezi entitami a transakcemi konzistentním způsobem, tj. výhradně použitím hran `BALANCE_IN` a `BALANCE_OUT`.

Všechny hrany vytvořené během této procedury mají opět ve svém atributu `timestamp` časovou značku shodnou s časovou značkou transakce, ke které se vztahují, čímž je zajištěna možnost omezovat jakékoliv dotazy podle času.



Obrázek 5.8: Ukázka stavu databáze po výpočtu bilancí transakcí pro entity (některé hrany jsou pro přehlednost vynechány)

5.2.7 Označení plateb mezi entitami

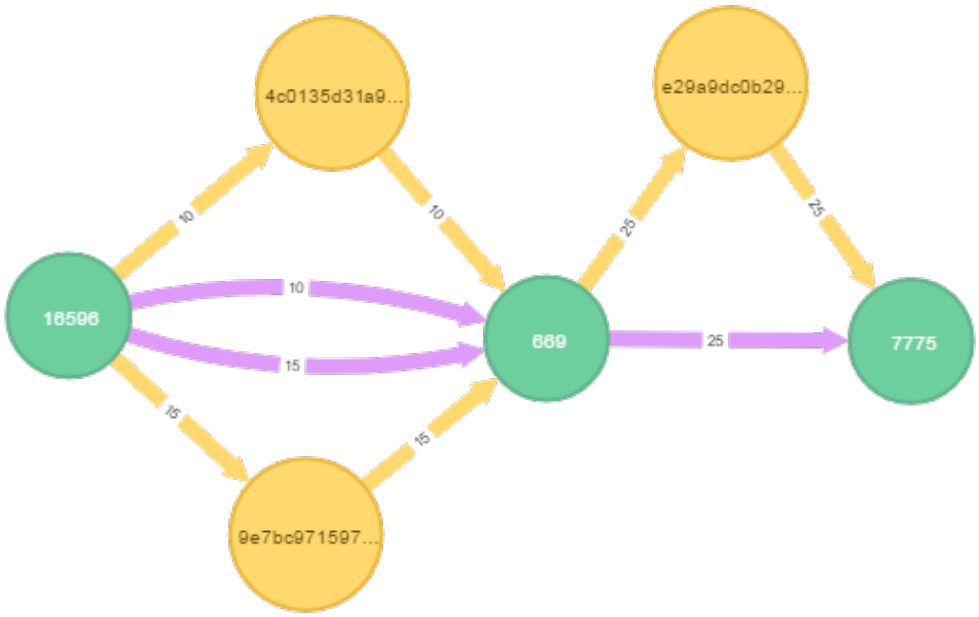
Poté, co jsou díky předchozím procedurám k dispozici hrany `BALANCE_IN` a `BALANCE_OUT`, které reprezentují pohyb zůstatku mezi entitami a transakcemi, zbývá již poslední krok k tomu, aby bylo možné se přirozeně dotazovat na „platby“ v pravém slova smyslu. Tímto krokem je transformace dvojic hran vedoucích „skrz“ transakci na jedinou hranu, která bude vyjadřovat pohyb zůstatku přímo mezi entitami.

Hrana s tímto významem je přidávána právě v této proceduře a je označena typem `PAYMENT`. Ilustrace je znázorněna na obrázcích níže, kde jsou patrné tři entity (zeleně) a tři transakce (žlutě). Entita `16596` vlevo odesílá postupně přes dvě transakce dva obnosy entitě `669` uprostřed, která pak oba tyto obnosy přeposílá dohromady entitě `7775` vpravo. Na obrázku 5.9 je ukázka stavu databáze před touto procedurou, přičemž některé hrany a především uzly `Address` pro přehlednost nejsou zobrazeny.

Obrázek 5.9: Ukázka stavu databáze před označením plateb mezi entitami (některé hrany jsou pro přehlednost vynechány)

Fialové hrany na obrázku 5.10 jsou hrany typu `PAYMENT`, které byly doplněny touto procedurou. Jak je patrné z obrázku, pomocí těchto hran je možné se dotazovat přímo na vazby mezi entitami, aniž by bylo nutné používat konkrétní transakce.

Stejně jako v předchozích případech, obsahují i tyto hrany ve svých atributech položku `value`, která značí hodnotu obnosu reprezentovaného danou hranou. Kromě toho je však do hran ještě doplněn atribut `txid`, který označuje identifikátor transakce, ve které proběhl přesun daného obnosu mezi entitami, aby bylo možné zpětně dohledat detaily o tom, jak k platbě došlo.



Obrázek 5.10: Ukázka stavu databáze po označení plateb mezi entitami (některé hrany jsou pro přehlednost vynechány)

5.3 Shlukování

Předchozí zpracování dat v grafové databázi, rozebírané v kapitole 5, bylo zaměřeno především na zpřehlednění vazeb mezi entitami pro uživatele. Výsledkem je graf ve formátu, kdy je možné klást uživatelské dotazy v intuitivní formě, aniž by bylo nutné je mapovat na nízkoúrovňové struktury dat.

Pokročilejší analýzou, která může přinést znalosti navíc, je shlukování, kterému se věnuje tato kapitola. Jedná se o metodu vytěžování dat, jejímž cílem je nalézt v datech skupiny objektů, které jsou si navzájem co nejvíce podobné a současně jsou co nejméně podobné objektům mimo svoji skupinu [9].

Shlukovou analýzu lze aplikovat také na grafové struktury, což je případ, který nastává v této práci. Obecně jsou grafové shlukovací algoritmy využitelné například pro řešení problémů z oblasti NLP[9], jako například rozpoznávání jazyků [10].

V kontextu dat zde zpracovávaných, se jedná především o analýzu vazeb mezi entitami, tj. uzly typu `Entity`. Rozhodující jsou v tomto případě hrany typu `PAYMENT` vytvořené procedurou popsanou v sekci 5.2.7, které značí platby, jež proběhly mezi danými entitami. Analýzou těchto vazeb lze odhalit skryté struktury a skupiny uzlů, mezi kterými probíhají časté platby nebo přesuny

[9]Natural Language Processing – Oblast počítačové vědy zabývající se zpracováváním přirozeného (lidského) jazyka počítačem a snahou, aby počítač jazyku porozuměl.

velkých obnosů. Ukázky výstupů shlukové analýzy dat v Neo4j jsou uvedeny v sekci 5.3.2.

Protože nejen v blockchainu objemy analyzovaných dat postupně rostou, je obecně častým problémem běžných shlukovacích algoritmů jejich časová složitost [11]. Pro shlukování dat ve formě grafu je v této práci použit algoritmus Chinese Whispers, který se na tento problém zaměřuje a využívá jednodušších a rychlejších postupů při shlukování. Nižší složitosti dosahuje díky tomu, že během shlukování využívá pouze lokální informace v okolí právě zpracovávaného uzlu. Tím je výrazně snížen čas potřebný k výpočtům, protože není třeba při každé iteraci procházet celý graf [10].

5.3.1 Chinese Whispers

Myšlenka tohoto algoritmu je inspirována dětskou hrou v češtině známé jako „Tichá pošta“, jejíž cílem je předávat si zprávu šeptem a postupně získat směšnou zkomoleninu původní zprávy. Chinese Whispers se oproti tomu snaží nalézt skupiny uzlů (hráčů), které vysílají podobnou zprávu svému okolí [12].

Chinese Whispers je speciální případ algoritmu MCL, který simuluje různé konečně dlouhé sledy v grafu a identifikuje shluky na základě toho, že je pravděpodobnější, že při náhodném sledu skončíme ve stejném shluku, kde jsme začali, než že překročíme hranici shluku [13].

Mějme prostý graf $G = \{U, H, \rho\}$ s množinou uzlů U a množinou ohodnocených hran H, který obsahuje n uzlů a m hran a ρ označuje incidenci. Na začátku algoritmu se každému uzlu přiřadí jeho vlastní třída, vznikne tedy n shluků, z nichž každý bude obsahovat právě 1 uzel. Poté je iterováno přes všechny uzly grafu G a každému z nich je přiřazena nová třída. Výběr nové třídy uzlu u je proveden tak, že mezi sousedy právě zpracovávaného uzlu je nalezen nejvíce zastoupený shluk. To znamená, že zpracovávaný uzel získá tu třídu, jejíž součet vah hran spojených s tímto uzlem je maximální. Pokud je tento součet po průchodu všech sousedů stejný pro více tříd, je z nich výsledná třída vybrána náhodně [10].

Algoritmus je popsán v pseudokódu 3, kde U označuje množinu uzlů grafu, u a v jednotlivé uzly a $u.class$ třídu uzlu u. Dále $rank[X]$ značí zastoupení třídy X, $rank.max$ třídu s maximální hodnotou v poli $rank$ (při shodě jednu náhodnou) a $w[u, v]$ váhu hrany mezi uzly u a v [12].

5.3.2 Aplikace na analyzovaná data

Po převedení do kontextu dat zpracovávaných v této práci tedy algoritmus pracuje tak, že iterativně shlukuje entity na základě toho, jaké množství prostředků si mezi sebou navzájem posílají.

Nejprve je každá entita ve svém vlastním, resp. žádném, shluku. Postupně se pak entity seskupují, až je výsledkem množina shluků, která je stálá (vyjádřeno podmínkou na proměnnou `changes` v pseudokódu 3).

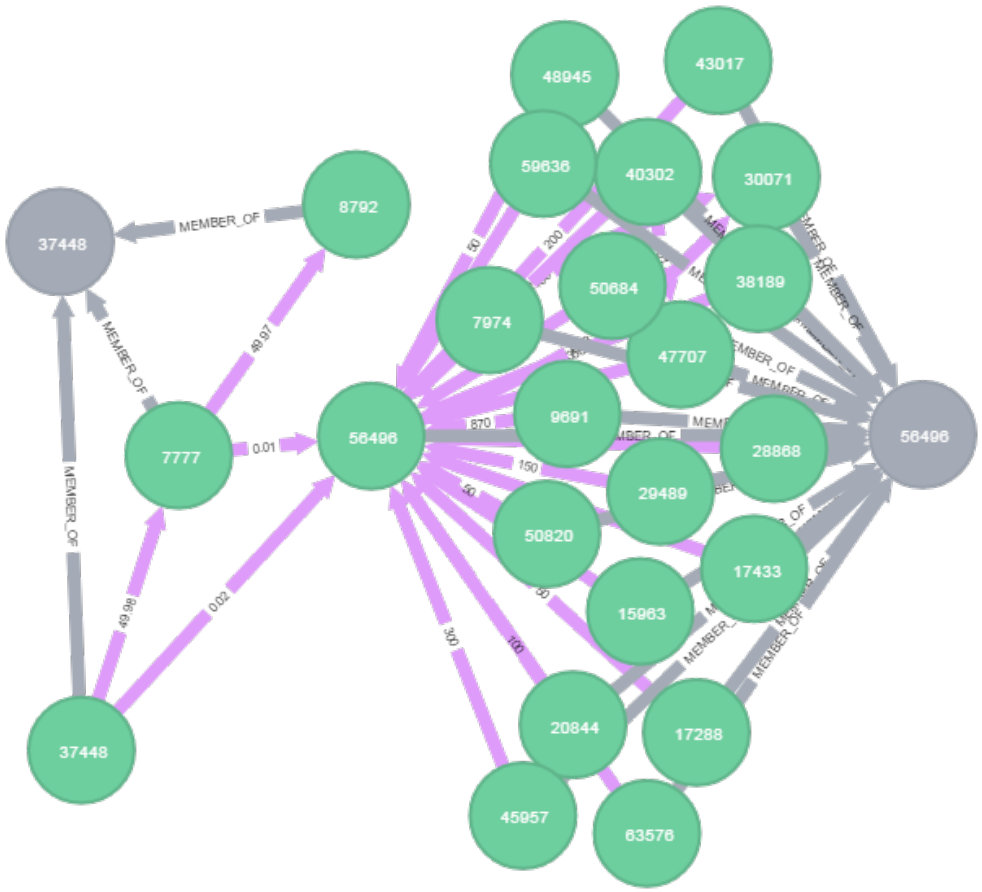
Pseudokód 3 Chinese Whispers

```
for all u in U do
  C[u] = u;
end for
while changes do
  for all u in U do
    for all v in neighbors(u) do
      rank[v.class] += w[u, v];
    end for
    u.class = rank.max;
  end for
end while
```

Po dokončení tohoto algoritmu má uživatel k dispozici snadnou cestu, jak na první pohled identifikovat entity, které jsou spolu navzájem spjaté. Protože jsou „váhy“ vazeb (tj. množství přesouvaných prostředků) mezi entitami sčítány, nezáleží na tom, jestli mezi nimi proběhlo více menších plateb nebo méně větších.

Příklad výstupu shlukování je ukázán na obrázku 5.11, kde jsou patrné doplněné hrany typu `MEMBER_OF` (šedivě) značící příslušnost entity (zeleně) do příslušného shluku (šedě), který je reprezentován uzlem typu `Cluster` (šedě).

Konkrétně zde lze pozorovat dva shluky, první (vpravo) obsahuje větší množství entit, z nichž jsou všechny spojeny s jednou „hlavní“ entitou uprostřed, která je rovněž členem tohoto shluku. Druhý oproti tomu obsahuje pouze tři entity vlevo a z jeho podoby je patrné, jak algoritmus reaguje na velikost obnosů přesouvaných mezi entitami. Ačkoliv jsou dvě entity z levého shluku svázány hranou `PAYMENT` také s „hlavní“ entitou pravého shluku, nebyly do tohoto shluku přidány, protože mezi nimi proběhly přesuny řádově vyšších obnosů (cca 50 BTC), než tomu bylo u onoho „hlavního“ uzlu (cca 0,01 BTC).



Obrázek 5.11: Ukázka stavu databáze po shlukové analýze (některé hrany a uzly jsou pro přehlednost vynechány)

Kapitola 6

Implementace

Tato kapitola se zabývá konkrétními detaily implementace postupů popsaných v kapitole 5, především je rozebrán objektový návrh a práce s grafovou databází Neo4j.

Základní myšlenkou je zpracování dat přímým přístupem do databáze přes dostupné Java API, které pracuje s embedded databází spouštěnou přímo v aplikaci. Jinými slovy, předzpracovací aplikace vytvořená v rámci této práce spouští přímo v rámci jedné JVM i instanci databáze Neo4j. Tento přístup má tři zásadní výhody:

- Aplikace je optimalizovaná pro vyšší výkon, jelikož u dotazů odpadá jednak nutnost komunikovat s databází přes externí rozhraní a jednak i nutnost stavět a parsovat dotazy v jazyce Cypher. Zároveň jsou jakékoliv úpravy objektů v databázi provedeny přímo, bez procházení indexů nebo jiných vyhledávacích struktur.

- S výhodou lze při implementaci algoritmů a „procházení“ grafu využívat přímé metody, které umožní přirozeně získat například hrany nebo sousedy uzlu. Není zde proto nutnost omezovat se na možnosti a rozhraní Traversal[10], ale je možné graf jakkoliv procházet a v průběhu rovnou upravovat.

- Jsou dostupné příkazy pro řízení transakcí a lze se dynamicky rozhodovat, jak by měly být dotazy zpracovávány. Tohoto faktu je využito například při omezení počtu dotazů v transakci, což je popsáno v sekci 6.1.1.1.

Zároveň je zde však zachována i možnost spouštět Cypher dotazy, které jsou použité například pro vytváření indexů.

[10]Traversal je framework v rámci Neo4j, který umožňuje předdefinovat pravidla pro průchod grafu.

6.1 Struktura aplikace

Předzpracovací aplikace implementovaná v rámci této práce je založena na nástroji Maven, přičemž mezi její závislosti patří artefakty[11] umožňující použití Neo4j.

Kromě běžných složek Maven projektu očekává aplikace ještě složku `neo4j`, v jejíž podsložce `databases` jsou uložena data pro zpracování. Základní konfigurace aplikace pracuje se souborem `data.db`, který je použit jako ukázka v sekci 4.2, avšak toto nastavení lze v aplikaci snadno změnit.

6.1.1 Neo4j embedded

Jak již bylo zmíněno v úvodu této kapitoly, k implementaci předzpracovacích operací je využita embedded databáze. Ta je používána pro jednoduché Cypher dotazy, jako například vytvoření indexů popsané v sekci 5.2.1 nebo dotaz

```
MERGE (c:Cluster {id:"XY"}) RETURN c
```

zajišťující nalezení nebo vytvoření uzlu typu `Cluster` s atributem `id` rovným řetězci `XY`. Tento dotaz je používán v implementaci shlukové analýzy rozebrané v sekci 5.3, kde zajišťuje, že uzel reprezentující daný shluk je použit, pokud již existuje, nebo vytvořen, pokud ještě neexistuje.

6.1.1.1 Řízení databáze

Základní třídou poskytovanou Neo4j, která je v předzpracovací aplikaci použita, je *GraphDatabaseService*[12]. Tato třída reprezentuje onu embedded databázi, která je uvnitř aplikace spouštěna. Z jejího rozhraní jsou využívány následující metody:

- `beginTx` – Spuštění databázové transakce, v rámci které je možné graf prohledávat a upravovat
- `execute` – Vykonání Cypher dotazu, který je předán v parametru
- `findNodes` – Nalezení uzlů[13], které splňují dané parametry, jako například daný *Label* (viz níže) nebo atribut

[11] Artefaktem je v tomto kontextu myšlen Maven artefakt, tedy projekt nebo knihovna potřebná pro spuštění aplikace, která na ní závisí.

[12] Kurzívou jsou značeny i další Java třídy, aby nedocházelo k záměně s označením pro `objekty` v databázi.

[13] Další používaná metoda `findNode` pracuje analogicky, pouze vrací jediný uzel místo množiny uzlů.

Databázová transakce je reprezentována třídou *Transaction*[14], z níž jsou využívány metody `success` a `close`, které, jak jejich názvy napovídají, slouží k potvrzení změn a následnému ukončení transakce. Tyto metody byly použity mimo jiné k tomu, aby byla omezena velikost prováděných transakcí. Při implementaci se totiž vyskytly potíže s nedostatkem paměti, jejichž příčina byla nalezena v přílišné velikosti databázových transakcí.

Pokud jsou ovšem prováděné operace rozděleny na více menších transakcí pomocí jejich ručního uzavírání a opětovného otevírání, nároky na paměť jsou tím sníženy. Výše zmíněné metody `success` a `close` jsou tedy periodicky volány ve spojení s metodou `beginTx` objektu *GraphDatabaseService*, což zapříčiňuje pravidelné obnovování transakcí. Intervaly, měřené počtem provedených iterací algoritmu, ve kterých se transakce obnovují, lze nastavit parametrem aplikace, jehož základní hodnota je 1000 iterací.

6.1.1.2 Uzly

Grafový uzel je v databázi reprezentován rozhraním *Node*, které poskytuje zejména metody pro práci s atributy a vazbami (hranami) příslušejícími danému uzlu.

Další důležitou součástí uzlu je jeho *Label*, což je vlastnost uzlu, jež je v textu této práce označovaná jako jeho „typ“. Jako příklad lze uvést uzel typu `Cluster` popisovaný v sekci 5.3.2, který má tedy reálně v databázi přiřazen *Label* `Cluster`.

Při předzpracování jsou používány tyto metody rozhraní *Node*:

- `setProperty` – Nastavení atributu uzlu na danou hodnotu
- `getProperty` – Získání hodnoty daného atributu uzlu
- `getRelationships` – Získání hran[15] konkrétního typu spojených s tímto uzlem (lze nepovinně zvolit i směr „z“ nebo „do“ uzlu)
- `hasRelationship` – Ověření, zda je k uzlu připojena hrana s daným typem a v daném směru
- `createRelationshipTo` – Vytvoření hrany daného typu z tohoto uzlu do jiného
- `delete` – Smazání uzlu (používáno pouze pro testovací běhy aplikace)

[14] Ačkoliv jde o stejný název třídy, nemá tato transakce nic společného s třídou *Transaction* popsanou níže, která reprezentuje transakci ve smyslu typu objektu uloženého v databázi.

[15] Další používaná metoda `getSingleRelationship` pracuje analogicky, pouze vrací jedinou hranu místo množiny hran.

6.1.1.3 Hrany

Pro reprezentaci hran v grafu je k dispozici rozhraní *Relationship*, jež obsahuje nejen přístup k typu a atributům hrany, ale také informace o tom, ke kterým uzlům je vázána. S výhodou tak lze využívat přímého přístupu k oběma koncovým uzlům, který je zajištěn metodami v tomto rozhraní.

Základními použitými metodami rozhraní *Relationship* jsou:

- `setProperty` – Nastavení atributu hrany na danou hodnotu
- `getProperty` – Získání hodnoty daného atributu hrany
- `getOtherNode` – Získání uzlu spojeného se zadaným uzlem touto hranou
- `delete` – Smazání hrany (používáno pouze pro testovací běhy aplikace)

6.1.2 Objektový návrh

Hlavní třídou pro předzpracovací operace je třída *Processor*, která obsahuje metody pro jednotlivé algoritmy používané k implementaci transformací popsaných v sekci 5.2. Dále jsou používány třídy pro modelování uzlů v databázi, třída pro

6.1.2.1 Implementace transformačních operací

Transformační operace zajišťující předzpracování dat přidáváním hran a uzlů, které je možné následně využít při analýze, jsou v aplikaci implementovány třídou *Processor*.

Metody této třídy jsou spouštěny postupně v pořadí dle seznamu níže a některé z nich využívají údajů, jež byly vypočítány v předcházejících metodách.

Funkcionalita zajišťovaná metodami třídy *Processor* odpovídá procedurám popsaným v sekci 5.2 (mimo vytvoření indexů v podsekci 5.2.1), přičemž v aplikaci jsou nazvány následovně:

- `fillTimestamps` – Doplnění časových značek do transakcí
- `markCoinbase` – Označení coinbase transakcí
- `connectTransactions` – Propojení transakcí s adresami a entitami
- `computeBalances` – Výpočet zůstatků pro adresy a entity
- `mergeTransactions` – Zjednodušení transakcí pro entity
- `findPayments` – Označení plateb mezi entitami

Interně používá třída pro práci s Neo4j objekt, který je instancí třídy *GraphDatabaseService*. Dostupné rozhraní této třídy je popsáno výše v podsekci 6.1.1.1.

Pro nastavení velikosti intervalu pravidelného obnovování databázové transakce, rovněž popsaného v podsekci 6.1.1.1, je použita proměnná `maxCommitSize` s datovým typem `int` uvnitř třídy. Hodnotu této proměnné lze nastavit pomocí volání metody `setMaxCommitSize`.

6.1.2.2 Implementace modelových tříd

Tyto třídy slouží k obalení objektů třídy *Node*, jež jsou dostupné v rámci Neo4j embedded, a přidání funkcionality, která ulehčuje práci s uzly v transformačních procedurách a shlukovacím algoritmu.

Všechny modelové třídy popisované v této sekci jsou podtřídami abstraktní třídy *NodeModel*, která slouží jako základ pro implementaci funkcionality vyžadované od všech modelových tříd. Některé podtřídy pak přidávají vlastní funkce, které jsou specifické pro jejich použití. Společné metody implementované v abstraktní třídě *NodeModel* pro usnadnění práce s vnitřními uzly grafu jsou následující:

- `getNode` – Získání vnitřního objektu typu *Node*, která je zabalován touto třídou
- `getNeighbor` – Získání souseda uzlu spojeného s tímto uzlem přes danou hranu. Tato metoda umožňuje přímočařejší zápis získávání sousedů tím, ze obaluje práci s hranou.
- `delete` – Smazání tohoto uzlu z grafu. Touto metodou je usnadněno mazání uzlů tak, že objekt třídy *NodeMode* sám nejdříve automaticky odpojí všechny hrany uzlu a následně smaže vnitřní uzel z grafu. Pokus o smazání uzlu, ke kterému jsou ještě připojeny hrany, by v Neo4j způsobilo chybu.

Dále jsou v této abstraktní třídě implementovány metody zkracující zápis v kódu běžně používaných volání metod, které všechny pracují analogicky a některé příklady jsou uvedeny v seznamu níže:

- `hasRelIn` – Ověření, zda je směrem do tohoto uzlu připojena hrana daného typu
- `getRelsOut` – Získání hran daného typu připojených směrem z tohoto uzlu
- `createRelTo` – Vytvoření hrany daného typu do uzlu předaného jako parametr

Namísto dlouhého zápisu využívajícího přímo metody objektů Neo4j embedded, jako například

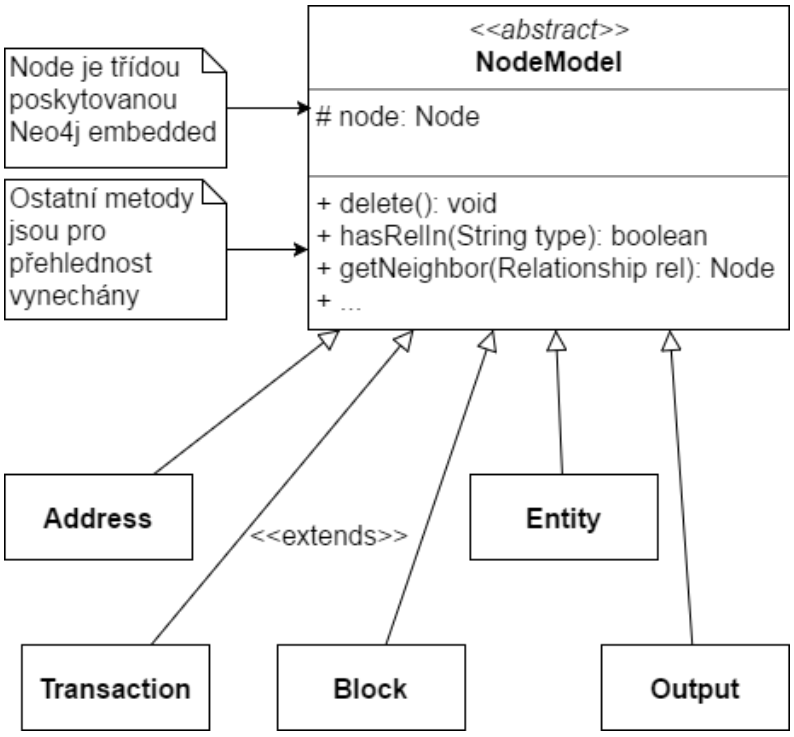
```
node.getRelationships(
    Direction.INCOMING,
    RelationshipType.withName("type")
);
```

lze pak psát kratší zápis

```
node.getRelsIn("type");
```

který při opakovaném použití tohoto typu metod výrazně zpřehledňuje kód a omezuje se na specifikaci pouze důležitých částí volání, kterými jsou směr a typ hrany, bez nutnosti doplňovat „technikálie“ okolo.

Vztahy mezi abstraktní třídou *NodeModel* a jejími podtřídami *Address*, *Block*, *Entity*, *Output* a *Transaction* ilustruje obrázek 6.1.



Obrázek 6.1: Vztahy mezi modelovými třídami

Některé z modelových tříd, z nichž každá reprezentuje uzel typu, který odpovídá jejímu názvu, navíc poskytují metody pro práci se specifickými vlastnostmi daného typu uzlu. Jako příklad lze uvést metodu `setBalance` třídy

Entity pro nastavení zůstatku, nebo metodu `getTargetTransaction` třídy *Output* pro získání jeho cílové transakce.

Dalším případem rozšířené funkcionality poskytované v modelové podtřídě jsou statické metody pro vyhledávání daných uzlů v grafu, jako například metoda `findById` třídy *Transaction*, která vrací uzel typu transakce s daným identifikátorem.

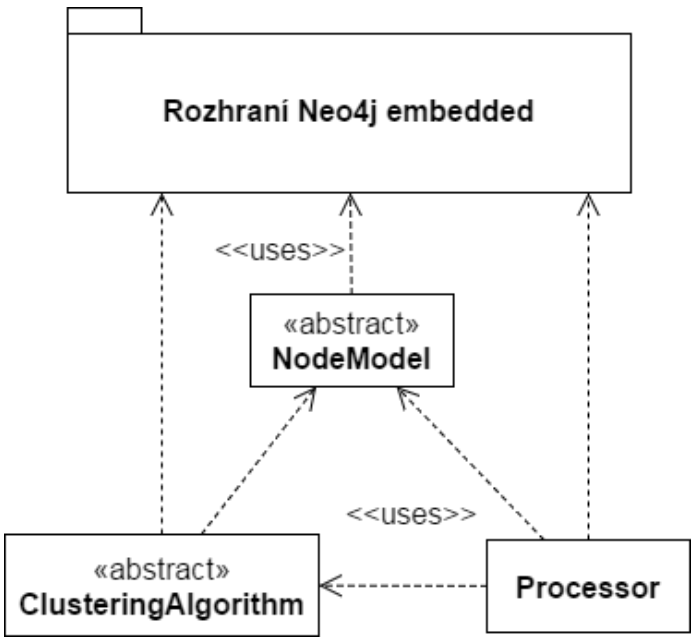
Zkracuje tak volání

```
// GraphDatabaseService db; String txid;
Transaction t = new Transaction(
    db.findNode(Label.label("Transaction"), "txid", txid)
);
```

na pouhé využití statické metody třídy *Transaction*:

```
// GraphDatabaseService db; String txid;
Transaction.findById(db, txid);
```

Zjednodušený model interakcí mezi třídami je znázorněn na obrázku 6.2, přičemž třída *ClusteringAlgorithm* je popsána v následující sekci 6.1.2.3.



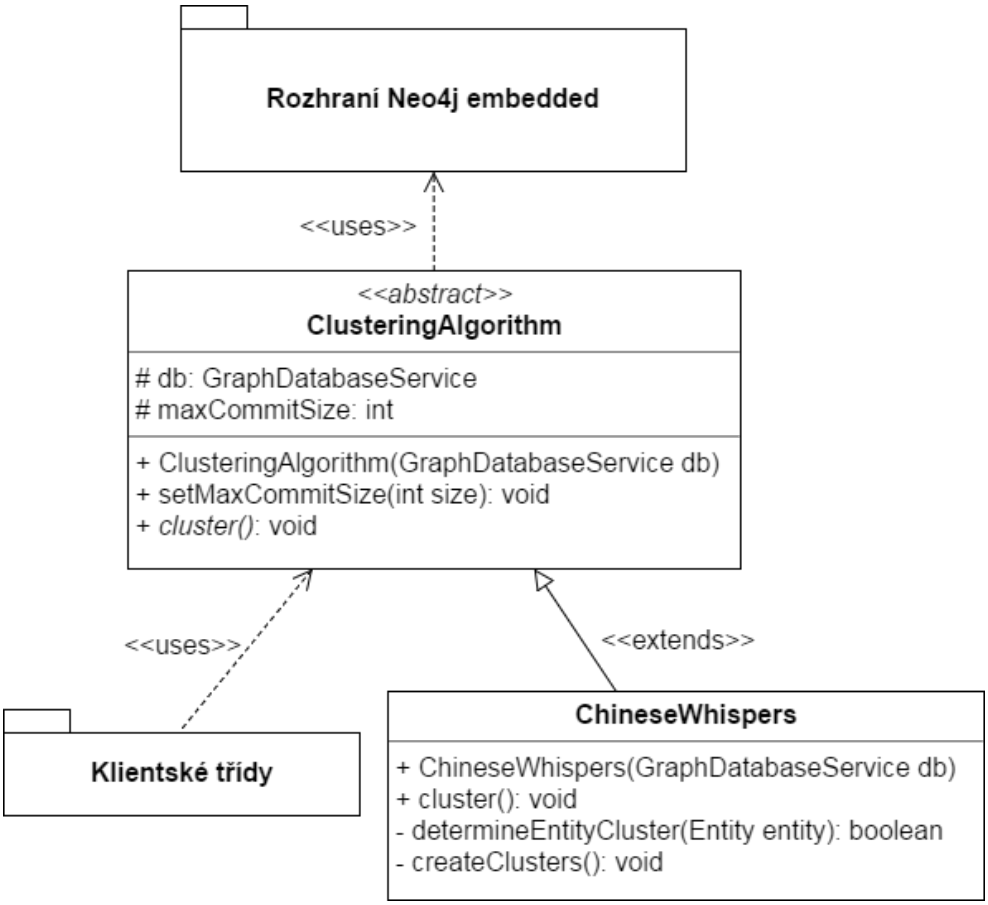
Obrázek 6.2: Interakce mezi hlavními třídami aplikace

6.1.2.3 Implementace shlukování

Třídou, která reprezentuje algoritmus implementující shlukovou analýzu, je *ClusteringAlgorithm*. Tato třída je abstraktní a sama o sobě neposkytuje možnost shlukování provést.

Jejím účelem je definovat rozhraní, které má být dodrženo různými algoritmy, aby je klienti mohli používat. Zároveň, stejně jako je tomu u třídy *Processor*, obsahuje objekty pro práci s grafem přes Neo4j rozhraní (instance třídy *GraphDatabaseService*) a pro nastavení velikosti intervalu omezujícího maximální počet provedených iterací bez obnovení databázové transakce (proměnná `maxCommitSize` typu `int`).

Diagram znázorňující vazby mezi *ClusteringAlgorithm* a ostatními třídami je na zobrazen na obrázku 6.3.



Obrázek 6.3: Třídy pro shlukovou analýzu a jejich vazby

Použitím této abstraktní třídy je zajištěno, že do aplikace bude možné snadněji přidat jiný algoritmus pro shlukovou analýzu. Klienti jakékoliv pod-

třídy mají zajištěno, že mohou použít rozhraní v ní definované. Oproti tomu instance této podtřídy budou mít přes vnitřní proměnné k dispozici rozhraní Neo4j a nastavení intervalů pro obnovování databázové transakce.

Ve třídě *ClusteringAlgorithm* je definovaný konstruktor přijímající v parametru instanci třídy *GraphDatabaseService*, která je následně uložena do vnitřní proměnné, aby ji bylo možné využívat při samotném shlukování.

Dále jsou zde definovány dvě metody:

- `setMaxCommitSize` – Nastavení velikosti intervalu pro maximální počet vykonaných iterací bez periodického obnovení databázové transakce
- `cluster` – Tato metoda je abstraktní a měla by tedy být implementována podtřídou tak, že provede shlukovou analýzu v grafové databázi, která je dostupná přes Neo4j rozhraní získané přes nadtřídu v konstruktoru

Konkrétní implementací shlukovací analýzy vytvořenou v rámci této práce je třída *ChineseWhispers* (na obrázku 6.3), která implementuje abstraktní metodu `cluster` třídy *ClusteringAlgorithm* a poskytuje algoritmus popsaný v podsekci 5.3.1 sekce 5.3 o shlukování. Kromě implementace abstraktní metody `cluster` přidává navíc dvě pomocné metody. První, `determineEntityCluster`, obaluje logiku algoritmu a přidává entitám pomocné atributy. Druhá metoda, `createClusters`, pak transformuje pomocné atributy algoritmu do výstupních objektů, které jsou uloženy do databáze jako výstup shlukové analýzy.

Kapitola 7

Testování

Experimenty s implementovanou aplikací probíhaly na stroji s následující konfigurací:

- Platforma – Windows 7 64-bit
- Procesor – Intel Core i5-2430M @ 2,40 GHz
- RAM – 4 GB (při testech bylo kvůli ostatním běžícím aplikacím dostupných přibližně 1,2 GB paměti)

Cílem bylo získat podklady pro odhad, jak závisí výpočetní čas na objemu zpracovávaných dat, což je podstatný údaj z hlediska výkonnosti implementované aplikace při použití na větších objemech dat ze zdrojové databáze.

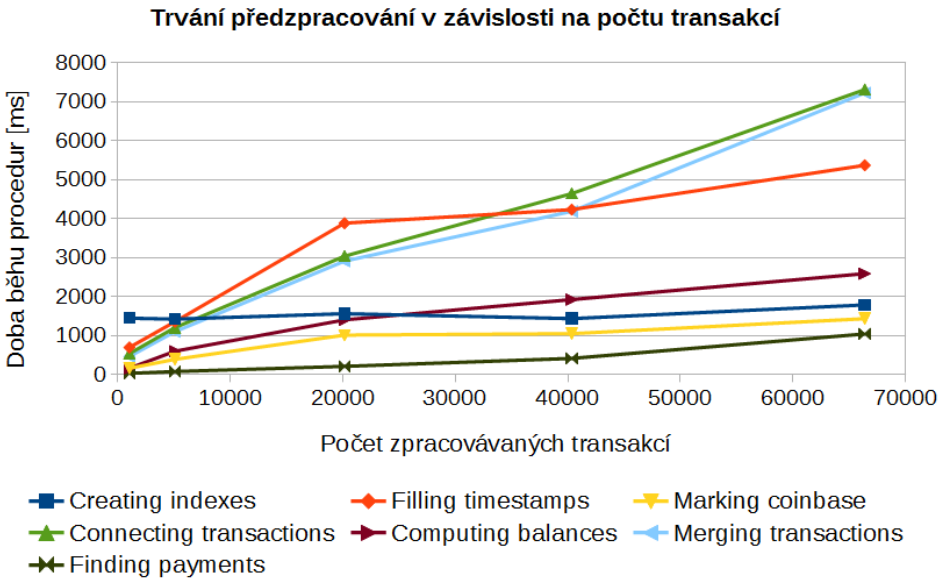
Měření bylo provedeno vždy třikrát pro každou instanci a výsledky byly zprůměrovány, aby byla snížena odchylka způsobená interferencí s ostatními aplikacemi běžícími na testovacím stroji.

7.1 Předzpracovací procedury

Výsledky běhů předzpracovacích procedur pro testovací instance, kterými bylo 1 000, 5 000, 20 000, 40 000 a 60 000 bloků z blockchainu, jsou znázorněny na obrázku 7.1. Jednotlivé procedury jsou zde označeny následovně:

- Creating indexes – Vytvoření indexů
- Filling timestamps – Doplnění časových značek do transakcí
- Marking coinbase – Označení coinbase transakcí
- Connecting transactions – Propojení transakcí s adresami a entitami
- Computing balances – Výpočet zůstatků pro adresy a entity

- Merging transactions – Zjednodušení transakcí pro entity
- Finding payments – Označení plateb mezi entitami



Obrázek 7.1: Měření doby běhu jednotlivých procedur pro různé objemy dat

Z grafu je patrné, že naměřené výsledky vykazují lineární závislost doby běhu jednotlivých procedur na počtu zpracovávaných transakcí.

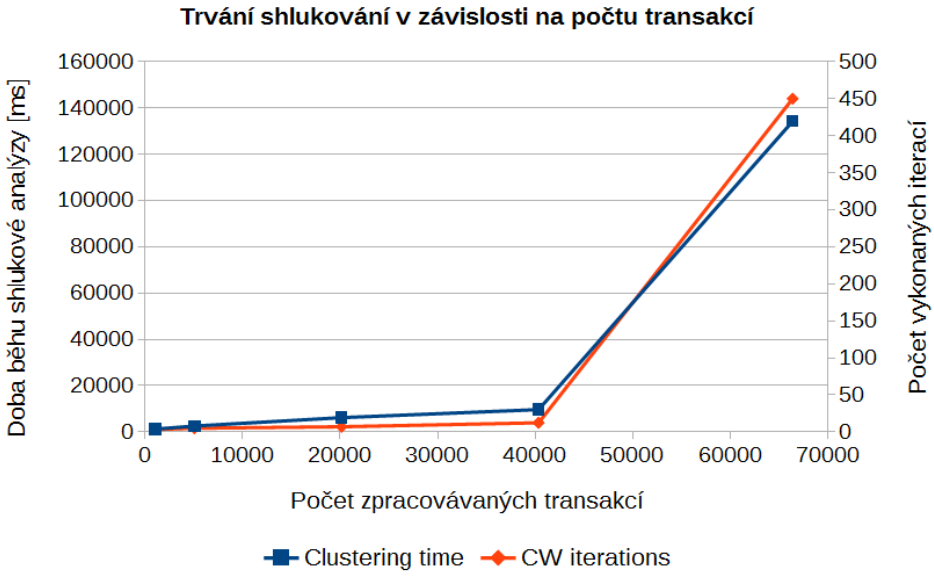
Lze proto očekávat, že časová náročnost neporoste se zvětšováním objemu dat rychleji než lineárně a tyto předzpracovací operace proto budou využitelné i pro rostoucí počet záznamů ve zdrojové databázi.

7.2 Shluková analýza

Měření výkonnosti shlukové analýzy v aplikaci komplikuje fakt, že implementovaný shlukovací algoritmus provádí proměnný počet iterací v závislosti na vstupních datech.

Zároveň zde hraje roli randomizace, jelikož, jak bylo popsáno v sekci 5.3.1, u algoritmu Chinese Whispers existují situace, kdy nastane „remíza“ a uzlu je přidělen shluk náhodně vybraný z dostupných kandidátů. Tato situace v grafech s ohodnocenými hranami sice nastává výrazně méně často, než je tomu v případě neohodnocených hran, avšak její výskyt může ovlivnit výsledek shlukování v různých bězích.

Na obrázku 7.2 je graf znázorňující závislost doby běhu shlukování (osa Y vlevo) a počtu iterací algoritmu Chinese Whispers (osa Y vpravo) na počtu zpracovávaných transakcí.



Obrázek 7.2: Měření doby běhu shlukové analýzy pro různé objemy dat

Z grafu vyplývá, že doba běhu shlukové analýzy roste pro počet transakcí nižší nebo roven přibližně 40 000 lineárně, stejně jako tomu bylo v případě předzpracovacích procedur popsaných v předešlé sekci 7.1.

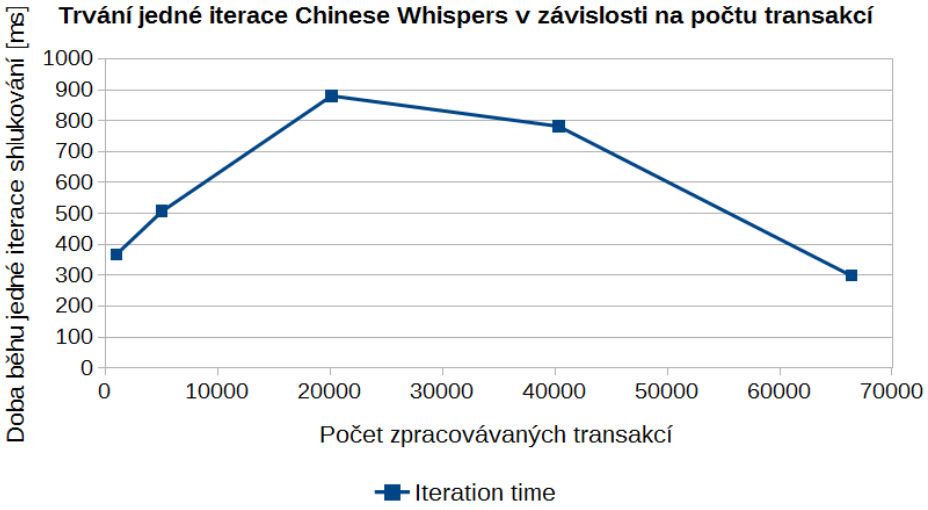
Zlom nastává v případě běhu zpracovávajícího 60 000, kdy dochází k prudkému nárůstu potřebného výpočetního času. Tento efekt však není způsoben „zpomalením“ algoritmu, nýbrž zvýšení počtu vykonávaných iterací, který je v grafu rovněž znázorněn.

Toto zvýšení počtu iterací algoritmu Chinese Whispers vyplývá převážně z vlastností, nikoliv objemu dat. Jak bylo vysvětleno v pseudokódu 3, iterace probíhají do té doby, než zůstane přiřazení uzlů do shluků neměnné. Mohou však nastat případy, kdy některé uzly „oscilují“ mezi dvěma nebo více shluky a v každé iteraci tak u nich dojde ke změně, která poté způsobí pokračování algoritmu. Tuto situaci lze vyřešit nastavením maximálního počtu iterací, který je možné provést před ukončením algoritmu.

Je-li tedy takto omezen maximální počet provedených iterací[16], pak je zaručeno, že doba běhu algoritmu závisí pouze na tomto zvoleném parametru a době běhu jedné iterace.

[16]Tento omezující parametr by měl být závislý na velikosti grafu a to tak, že neporoste rychleji než lineárně s počtem jeho uzlů.

Graf 7.3, který znázorňuje závislost průměrného času potřebného pro vykonání jedné iterace na počtu zpracovávaných transakcí, pracuje s takto zvoleným parametrem a časem běhu jedné iterace algoritmu Chinese Whispers.



Obrázek 7.3: Měření doby běhu jedné iterace shlukování pro různé objemy dat

Při pohledu na výše uvedený graf 7.3 lze vyčíst, že odchylky v časech běhů jednotlivých iterací se pohybují v rozmezí 600 milisekund a není patrné, že by tyto časy rostly v závislosti na počtu zpracovávaných transakcí.

Na základě tohoto pozorování lze soudit, že zásadní vliv na časovou náročnost algoritmu má nastavení parametru omezujícího počet iterací, kterým lze zabránit „oscilaci“ entit mezi shluky a provádění nadbytečných iterací.

Doby běhu jednotlivých iterací jsou z podstaty věci do určité míry ovlivňovány také počtem uzlů v grafu, avšak pro celkovou dobu běhu shlukování jsou dominujícími faktory počet iterací a „kvalitativní“ struktura hran mezi uzly, která může způsobit nárůst právě tohoto počtu provedených iterací.

Závěr

Cílem této práce bylo importovat data obsažená v distribuované databázi blockchain, která je využívána kryptoměnou Bitcoin, do databáze vhodné pro analýzu, a následně vytvořit aplikaci pro předzpracování těchto dat takovým způsobem, aby bylo možné z nich získat užitečné informace nebo znalosti.

Importu dat a samotné implementaci předcházela analýza zadané distribuované databáze a možností, jakými lze z dat v ní obsažených získávat ucelené informace. Následný import byl pak proveden pomocí zvoleného nástroje, jež umožňuje transformovat struktury ze zdrojové databáze do grafové struktury, která je použita v cílové databázi.

Jednotlivé kroky v implementaci pak byly provedeny formou na sebe navazujících procedur, které pracují s daty importovanými do grafové databáze Neo4j, která byla zvolena jako cílová databáze.

Výsledná data získaná za pomoci implementovaných procedur jsou v podobě, kdy se nad nimi lze dotazovat intuitivní formou, která již není svázána původním formátem struktur ve zdrojové databázi. K tomu byly do dat dále doplněny nové záznamy o shlucích některých objektů, které mohou poskytnout další podrobnější vhled do dostupných informací.

Měření ukazují, že doby běhů transformačních procedur rostou lineárně v závislosti na objemu zpracovávaných dat, což umožňuje jejich použití pro rostoucí množství záznamů ve zdrojové databázi. Rovněž algoritmus pro shlukovou analýzu implementovaný v rámci této práce vykazuje při zpracování dat dostatečnou výkonnost, pokud je vhodným omezením maximálního počtu provedených iterací zabráněno „oscilaci“ uzlů mezi shluky.

Budoucí práce

Jak bylo zmíněno v sekci 6.1.2.3, do aplikace je možné přidat kromě Chinese Whispers další implementace algoritmů pro shlukovou analýzu a následně srovnávat jejich úspěšnost v různých případech, aby si uživatel mohl vybrat,

který algoritmus bude pro shlukování použit, nebo měl k dispozici výstupy všech z nich.

Dalším možným rozvojem poskytované funkcionality jsou úpravy založené na uživatelských požadavcích, které se mohou v budoucnu objevit. Tyto úpravy by se mohly týkat například doplňování podrobnějších výpočtů k jednotlivým uzlům v grafu nebo přidávání nových informací s využitím znalostí třetí strany.

Literatura

[1] Steklík, Š.: *Vizualizace a vyhledávání v distribuované databázi.* Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

[2] Antonopoulos, A. M.: *Mastering Bitcoin: Unlocking Digital Crypto-Currencies.* O'Reilly Media, Inc., první vydání, 2014, ISBN 1449374042, 9781449374044.

[3] Bitcoin Developer Guide. [online], [cit. 2017-04-22]. Dostupné z: `https://bitcoin.org/en/developer-guide`

[4] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. Květen 2009. Dostupné z: `http://www.bitcoin.org/bitcoin.pdf`

[5] DB-Engines Ranking of Graph DBMS. [online], [cit. 2017-04-16]. Dostupné z: `https://db-engines.com/en/ranking/graph+dbms`

[6] Bitcoin Core. [online], [cit. 2017-04-22]. Dostupné z: `https://bitcoin.org/en/bitcoin-core/`

[7] Bitcoingraph documentation. [online], [cit. 2017-04-18]. Dostupné z: `https://github.com/behas/bitcoingraph/blob/master/README.md`

[8] Ron, D.; Shamir, A.: Quantitative Analysis of the Full Bitcoin Transaction Graph. *IACR Cryptology ePrint Archive*, ročník 2012, 2012: str. 584. Dostupné z: `http://eprint.iacr.org/2012/584`

[9] Jain, A. K.: Data Clustering : 50 Years Beyond K-Means. *Pattern Recognition Letters*, 2010. Dostupné z: `http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.18.2720&rep=rep1&type=pdf`

[10] Biemann, C.: Chinese Whispers - an Efficient Graph Clustering Algorithm and its Application to Natural Language Processing Problems. In *Proceedings of TextGraphs: the Second Workshop on Graph*

Based Methods for Natural Language Processing, New York City, USA, 2006, s. 73–80. Dostupné z: http://www.bibsonomy.org/bibtex/210699e6fa5efdf7b8b10d1b052ae54be/fluctuator

[11] Mishra, R.; Shukla, S.; Arora, D. D.; aj.: An Effective Comparison of Graph Clustering Algorithms via Random Graphs. *International Journal of Computer Applications*, ročník 22, č. 1, May 2011: s. 22–27.

[12] Křeček, M.: *Rozšíření platformy Clueminer o grafové algoritmy*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

[13] van Dongen, S.: *A Cluster algorithm for graphs*. Dizertační práce, Centrum voor Wiskunde en Informatica, 2000. Dostupné z: http://www.bibsonomy.org/bibtex/2d7d305114e2d45acdea509c28769881c/jullybobble

Příloha **A**

Seznam použitých termínů

API Application Programming Interface – Rozhraní pro programování aplikací

BTC Bitcoin – Označení jednotky bitcoinové měny

CSV Comma-separated values – Formát dat s hodnotami oddělenými čárkami

Cygwin Sada programů simulujících chování Linuxových systémů

Cypher Dotazovací jazyk pro grafovou databázi Neo4j

Embedded Vestavěný, zabudovaný systém, v tomto kontextu databáze běžící uvnitř jiné aplikace

GIT Systém pro správu verzí

GUI Graphical user interface – Grafické uživatelské rozhraní

JVM Java Virtual Machine – Virtuální stroj pro spouštění programů v jazyce Java

Maven Nástroj pro správu, řízení a automatizaci buildů aplikací

NLP Natural Language Processing – Oblast počítačové vědy zabývající se zpracováváním přirozeného (lidského) jazyka počítačem a snahou, aby počítač jazyku porozuměl

P2P Peer-to-peer – síť, ve které jsou jednotlivé uzly navzájem rovnocenné

RAM Random Access Memory – operační paměť

REST Representational state transfer – Architektura rozhraní pro HTTP komunikaci

RPC Remote procedure call – Vzdálené volání procedur

UTXO Unspent transaction output – Určitý objem BTC, který vzešel z transakce

Příloha B

Obsah přiloženého CD

- `dp-proc` ..... adresář s projektem aplikace
  - `neo4j` ..... adresář s testovacími daty
- `src` ..... adresář se zdrojovou formou práce ve formátu LaTeX
  - `img` ..... adresář s obrázky pro zdrojovou formu práce
- `text` ..... adresář s textem práce
  - `DP_Krecek_Martin_2017.pdf` ..... text práce ve formátu PDF