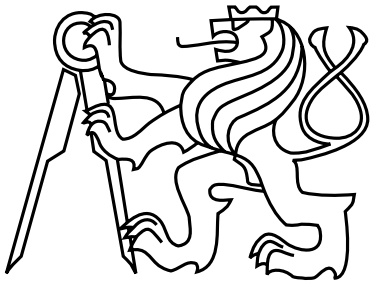
Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Analýza a návrh abstraktní vícevrstvé architektury pro práci s grafovou databází realizující metadatové úložiště pro data lineage

Bc. Jakub Moravec

Vedoucí práce: Ing. Michal Valenta, Ph.D.

Studijní program: Otevřená Informatika, Magisterský

Obor: Softwarové inženýrství

7. května 2018

Poděkování

Zde můžete napsat své poděkování, pokud chcete a máte komu děkovat.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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. V souladu s ust. § 2373 občanského zákoníku tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům), vč. možnosti Dílo upravit či měnit, spojit jej s jiným dílem a/nebo zařadit jej do díla souborného. Toto oprávnění je časově, teritoriálně i množstevně neomezené a uděluji jej bezúplatně.

V Praze dne 7. května 2018 .....

Abstract

Translation of Czech abstract into English.

Abstrakt

Abstrakt práce by měl velmi stručně vystihovat její obsah. Tedy čím se práce zabývá a co je jejím výsledkem/přínosem.
Očekávají se cca 1 – 2 odstavce, maximálně půl stránky.

Obsah

1 Úvod **1**
- 1.1 Data lineage . . . 1
- 1.2 Definice problému . . . 2
- 1.3 Struktura diplomové práce . . . 2

2 Grafové databáze **3**
- 2.1 NoSQL databáze . . . 3
- 2.2 Grafy . . . 5
- 2.3 Reprezentace grafu . . . 7
- 2.4 Typy dotazů . . . 9
- 2.5 Indexy . . . 11
- 2.6 Transakce . . . 13
- 2.7 Distribuovatelnost . . . 14
- 2.8 Databáze - příklady . . . 15
- 2.9 Dotazovací jazyky . . . 16

3 Softwarové abstrakce **19**
- 3.1 API . . . 20
- 3.2 Softwarové architektury . . . 21
  - 3.2.1 Vícevrstvá architektura . . . 23
  - 3.2.2 REST . . . 23
  - 3.2.3 Architektury cloudových aplikací . . . 25

4 Analýza **29**
- 4.1 Manta Flow . . . 29
- 4.2 Metadatové úložiště . . . 29
- 4.3 Popis komponent serverové části . . . 32
  - 4.3.1 Connector . . . 32
  - 4.3.2 Merger . . . 33
  - 4.3.3 Viewer . . . 34
  - 4.3.4 Public API . . . 34
  - 4.3.5 Exporter . . . 35
- 4.4 Popis ostatních komponent . . . 35
  - 4.4.1 Manta Flow Client . . . 35
  - 4.4.2 Configurator . . . 35

4.4.3 Updater . . . 35
4.5 Omezení stávající architektury aplikace . . . 35
4.5.1 Datová konzistence . . . 36
4.5.2 Výkon aplikace . . . 38
4.5.3 Škálovatelnost . . . 39
4.5.4 Viditelnost . . . 40
4.5.5 Modifikovatelnost . . . 40
4.5.6 Orchestrace aplikací . . . 41
4.5.7 API grafových databází . . . 42
4.6 Požadavky na návrh architektury aplikace . . . 42
4.7 Existující řešení pro abstrakci grafových databází . . . 43

5 Návrh architektury **45**
5.1 Úprava architektury komponenty Connector . . . 45
5.1.1 Transakční model a řízení konzisence dat . . . 46
5.1.2 Návrh komponent . . . 48
5.1.2.1 Doménový model . . . 49
5.1.2.2 Databázová vrstva . . . 50
5.1.2.3 Perzistentní vrstva . . . 50
5.1.2.4 Vrstva datového přístupu . . . 52
5.1.2.5 Algoritmy . . . 53
5.2 Orchestrace komponent Manta Flow . . . 53
5.2.1 Orchestrace na jednom zařízení . . . 53
5.2.2 Orchestrace po síti . . . 54
5.3 Možnosti horizontálního škálování aplikace . . . 55
5.3.1 Škálování grafové databáze . . . 56
5.3.2 Škálování Java komponent aplikace . . . 57

6 Implementace prototypu **59**
6.1 Doménový model . . . 59
6.2 Databázová vrstva . . . 60
6.3 Perzistentní vrstva . . . 60
6.3.1 Vrstva *mapperů* . . . 60
6.3.2 Dotazy do metadatového úložiště . . . 60
6.3.3 Rozhraní pro *query* metody . . . 61
6.4 Vrstva datového přístupu . . . 62
6.5 Validace a testování . . . 63

7 Závěr **65**

A Seznam zkratek **73**

B Diagramy **75**

C Obsah přiloženého CD **87**

Seznam obrázků

2.1 Ohodnocený graf 6
2.2 Atributový graf 6
2.3 Reprezentace (orientovaného) grafu 9
2.4 Příklad průchodu (*traverzování* grafem) 10
2.5 Znázornění dotazů na nadgraf a podgraf 12

4.1 Model grafové databáze 30
4.2 Způsob verzování modelu metadat 32

5.1 Upravená architektura modulu Connector 49
5.2 Struktura perzistenční vrstvy 53

B.1 Stávající architektura *Manta Flow* 76
B.2 Interakce mezi *klientskou* a *serverovou* částí *Manta Flow* 77
B.3 Aktuální orchestrace aplikací *Manta Flow Server, Client, Updater a Configurator* 78
B.4 První krok aktualizace všech komponent - aktualizace *Manta Flow Toolbox* . 79
B.5 Standardní aktualizace komponent na jednom zařízení 80
B.6 Aktualizace komponenty *Manta Flow Client* přes *HTTPS* 81
B.7 Upravená orchestrace aplikací *Manta Flow Server, Client a Toolbox* 82
B.8 Doménový model 83
B.9 Implementace kontroly oprávnění 84
B.10 Diagram komponent prototypové implementace 85

Seznam tabulek

4.1 Funkční požadavky na architekturu aplikace 42
4.2 Nefunkční požadavky na architekturu aplikace 43

6.1 Pokrytí prototypové impolementace návrhu jednotkovými testy 63

Seznam příkladů

2.1 Gremlin (Java) - průchod grafem 17
2.2 Ukázka Cypher dotazu 17
3.1 Hypermedia zpráva ve formátu JSON 25
6.1 Mapování entity *Flow* (implementace pomocí *Gremlin 2.x*) 60
6.2 Nalezení uzlu dle kvalifikovaného jména (implementace pomocí *Pipes*) 61
6.3 Fluent interface query metod 62

Kapitola 1

Úvod

1.1 Data lineage

Data a z nich získávané informace vždy byly v centru pozornosti informačních technologií a jejich význam každým rokem stoupá. V digitální podobě jsou dnes zakódovány takřka všechny informace včetně našich osobních údajů, bankonvních transakcí, či zdravotních informací. Zároveň stále roste roste množství těchto dat[1]. Je tedy kladena velká pozornost na procesy, kterými jsou data zpracovávána a pomocí kterých jsou z dat získávány informace. Existuje několik přístupů k tomuto problému, ať už se jedná o tradiční relační databázové systémy, datové sklady a *Online Analytical Processing (OLAP)* analytické nástroje, *data miningové* technologie, nebo novější obory, jako jsou *NoSQL* databáze a analytické nástroje. Ať už je zvolen kterýkoliv z těchto přístupů, procesy zpracovávající data bývají komplexní a často ne zcela intuitivní pro samotné vývojáři, natož potom pro analytiky či dokonce byznys uživatele.

Do popředí se tak dostává nová skupina nástrojů označovaných jako *data lineage*.[2] Jejich cílem je analyzovat end-to-end datové toky v systému - zdroje, transformace a cíle dat a pomocí této analýzy umožnit uživateli vhled do tohoto procesu. To může být velmi komplexní úkol, informační systém se typicky skládá z řady navzájem propojených technologií, a nástroj pro analýzu Data lineage si musí umět poradit nejen s každým z nich separátně, ale také s případnými transformacemi na hranicích těchto systémů.

Jedním z úspěšných nástrojů pro *data lineage* je *Manta Flow*[3]. Nástroj analyzuje zdrojové kódy vybraných *RDBMS* databází, Big Data nástrojů a *ETL* nástrojů. Zdrojové kódy analyzovaných systémů jsou pravidelně parsovány dle syntaktických a sémantických pravidel podporovaných nástrojů a následně jsou analyzovány přímé a nepřímé[4] datové toky a

[1] Podle statistiky IDC [36] se celkový objem dat virtuálního světa zdvojnásobuje každé dva roky.

[2] Stejně jako mnoho další termínů z oblasti informačních technologií se *data lineage* nepřekládá, nebudeme ho tedy překládat ani my. Pokud bychom termín však přeci jen chtěli popsat českými slovy, nejvhodnější překlad by byl zřejmě "řízení datových toků".

[3] <https://getmanta.com/>

[4] Představme si relační databázi s tabulkami *A*, *B* a *C*. Představme si, že data z tabulky *A* jsou *ETL transformací* přenesena do tabulku *B*. Tato *ETL transformace* filtruje data z tabulky *A* dle dat z tabulky *C*. Potom z tabulky *A* do tabulky *B* vede *přímý datový tok* a z tabulky *C* do tabulky *B* vede *nepřímý datový tok*.

transformace dat v informačním systému. Získané informace jsou ukládány do metadatového úložiště, jímž je v současné době grafová databáze *Titan*[5]. Webové rozhraní aplikace potom umožňuje uživateli vizualizovat datové toky dle zadaných parametrů (zdroj a cíl datového toku, úroveň abstrakce atd.). Dynamicky tak vznikají komplexní dotazy do metadatové databáze, pomocí kterých jsou procházeny grafy datových toků a vraceny výsledky.

1.2 Definice problému

Přestože má použití grafové databáze jako metadatového úložiště pro Data lineage nástroje silné opodstatnění[6], přináší s sebou krom nesporných výhod také řadu problémů. Jejich společným jmenovatelem je fakt, že v oblasti grafových databází, která je relativně nová a stále prochází dynamickým rozvojem, nejsou zatím jasně definovány obecně podporované standardy. Neexistuje například univerzální, stabilní a obecně podporovaný dotazovací jazyk pro grafové databáze (například v oblasti relačních databází tuto úlohu plní SQL). To vede mimo jiné k tomu, že nejsou v tuto chvíli definovány doporučené postupy softwarového inženýrství pro tvorbu abstraktních rozhraní pracujících s grafovými databázemi. Není tak překvapaním, že je při používání grafových databází v aplikacích často míchána perzistentní a byznys logika aplikace, což je typickou ukázkou špatného návrhu[69]. Pro produkt *Manta Flow* je tento problém velice aktuální - používaná databáze *Titan* již není dále vyvýjena, brzy skončí její podpora [7] a je pravděpodobné, že dojde k její výměně za jinou technologii. Cílem této práce je navrhnout abstraktní architekturu pro práci s grafovou databází, která bude vyhovovat potřebám nástroje *Manta Flow* a bude v co největší míře oddělovat perzistentní logiku od zbytku aplikace. Zavedení této vrstvy aplikace bude pravděpodobně znamenat zásah do celé architektury aplikace. Součástí práce tedy musí být nový návrh architektury aplikace reflektující změnu v přístupu ke grafové databází.

1.3 Struktura diplomové práce

Práce je rozdělena na sedm kapitol: *Úvod*, dvě kapitoly věnované rešerši (*Grafové databáze* a *Softwarové abstrakce*), tři kapitoly věnované řešení definovaného problému (*Analýza*, *Návrh architektury*, *Implementace prototypu*) a *Závěr*.

Cílem rešerše je popsat obecné principy grafových databází, jejich vnitřní organizaci a možnosti dotazování dat. Dále jsou popsány možnosti abstrakce ve světě softwarového inženýrství - ať už na úrovní procesů a programových rozhraní (*API*), nebo na úrovni softwarových architektur. Obě kapitoly tvořící rešerši představují nutný teoretický základ pro řešení definovaného problému.

Praktická část práce obsahuje hlubší analýzu potřeb projektu *Manta Flow* pro manipulaci s metadatovým úložištěm reprezentovaným grafovou databází. Na základě rešerše je navržena vícevrstvá architektura, vytvořena prototypová implementace řešení a ta zvalidována a otestována nad reálnými daty.

[5] <http://titan.thinkaurelius.com/>

[6] Grafová databáze umožňuje výrazně rychlejší hledání datových toků v informačních systémech, než by umožňovali jiné architektury. Způsob procházení grafů je popsán v kapitole 2.4.

Kapitola 2

Grafové databáze

Grafy jsou velice přirozeným způsobem reprezentace dat, zvláště v době, kdy většina dat je vytvářena uživateli a není strukturovaná. Grafy pomocí uzlů a hran přirozeně popisují objekty a vztahy mezi nimi, nevynucují náročné datové modelování skutečnosti do složitých datových struktur a usnadňují tak proces objevování informací v datech. Díky tomu, že grafy umožňují jednoduché prohledávání objektů, které mezi sebou mají vztahy (uzlů propojených hranami), jsou operace tohoto typu nad grafy relativně (například vůči relačním databázím) rychlé.

Grafové databáze umožňují data reprezentovaná grafem ukládat a procházet tak, aby byly tyto přirozené výhody grafů zachovány a v některých případech byly také zajištěny některé vlastnosti relačních databázových systémů (jako například atomicita transakcí). Problémů, pro které je vhodné využití grafových databází je mnoho, příkladem může být ochrana proti podvodům v bankovním sektoru. Některé vzorce v datech je složité odhalit pomocí modelů relačních databází. Grafové databáze naopak přináší nový pohled na data a implicitně ukazují vztahy mezi nimi. Umožňují tak odhalit v datech podezřelé vzorce, které často mohou znamenat právě bankovní podvody.[77] Dalším z mnoha příkladů jsou také řešení v oblasti *data lineage*, která si kladou za cíl sledovat vztahy mezi daty pocházejícími zněkolika datových zdrojů a mapovat proces jejich zpracování.

Tato kapitola má za cíl představit základní principy grafových databází. Popisuje reprezentaci grafů v databázi, možnosti analýzy dat a rozhraní, kterých je k tomu možné použít.

2.1 NoSQL databáze

NoSQL databázové sice nejsou předmětem této práce, grafové databáze nicméně bývají zařazovány jako podkategorie *NoSQL* a proto zde základní principy této skupiny technologií popíšeme. Samotná zkratka *NoSQL* je vykládaná různě, většinou jako *"Not only SQL"* [22] a vykládá se jako označení pro databáze nerelačního typu určené pro zpracování velkého objemu různorodých dat - *Big Data*[1]. Mezi hlavní uváděné výhody *NoSQL* databází patří:

[1] Big Data jsou soubory dat velkého objemu, velké rychlosti a/nebo velké různorodosti, která vyžadují nové formy zpracování pro umožnění lepšího rozhodování, většího porozumění domény a optimalizace procesů.[45]

- *Škálovatelnost:* Zatímco klasické databázové systémy využívají *vertikální škálovatelnost*, *NoSQL* databáze umožňují efektivní *horizntální škálování*. *Vertikální škálování* je prováděno pomocí navýšení zdrojů (ať už výpočetní kapacity, nebo paměti) jednoho zařízení, což má své technické a ekonomické limity. Na druhé straně u horizontálně škálovatelného systému, lze navýšit jeho výkon a/nebo kapacitu přidáním dalšího zařízení. Jedná se tedy o síť spolupracujících zařízení. Data v *NoSQL* databázích bývají typicky distribuovaná na několik uzlů[2] a jejich zpracování tak může probíhat paralelně.

- *Efektivní čtení: NoSQL* datatabáze jsou silně orientovány na rychlé čtení (koncept *write once, read many times*). Ve většině případů již nejsou data po zapsání do databáze nikdy modifikována, pouze nad nimi jsou vykonávány analýzy (dotazy). Za cenu potenciálně pomalejšího zápisu dat do databáze (který nám nevadí) získáme tedy výrazně vyšší rychlost čtení dat.

- *Flexibilní datový model: NoSQL* databáze buď nevyžadují žádné datové schéma, nebo je schéma volné (a lze ho tedy upravovat bez nutnosti úpravy stávajících dat). Zajištění dodržování konzistence dat je tedy na aplikaci, která *NoSQL* databázi používá.

- *Ekonomická stránka:* jak již bylo uvedeno, relační databáze je nutné typicky škálovat vertikálně a to je nákladné - jsou navyšovány prostředky jednoho stroje (serveru), což má technická omezení a čím více se těmto omezením blížíme, tím je škálování dražší. Vedlejším efektem této skutečnosti je často také *vendor-locking*, tedy situace, kdy jsme nuceni používat specifický hardware (často od jedné konkrétní společnosti), který je velmi nákladný. Na druhé horizontální škálování je relativně levné - škáluje se přidáním nového uzlu. Vzhledem k tomu, že *NoSQL* databáze nevyžadují specifický hardware, je možné používat takzvaný *komoditní hardware* (uzly nemusí být hardwarově homogenní), který je levný.

Aby těchto výhod *NoSQL* databáze dosáhly, využívají různé přístupy k reprezentaci dat a manipulaci s daty. Jejich hlavní rozdělení je následující:

- *Databáze typu klíč-hodnota:* Tyto databáze mají zcela volné datové schéma, jsou realizovány jako mapa klíčů a hodnot. Hodnota přitom typicků může nabývat několika datových typů (například číslo, text, binární řetězec, kolekce některého z předchozích). Operace nad tímto úložištěm jsou poměrně jednoduché a zpravidla neposkytují pokročilé nástroje pro analýzu dat na základě obsahu - pouze na základě klíče. Příklady tohoto typu databází jsou *Redis*[3], *Riak*[4], *Persistit*[5], nebo *Voldemort*[6].

- *Dokumentová databáze:* Dokumentové databáze ukládají a spravují zpravidla strukturované dokumenty. Nejčastější formáty dokumentů jsou *JavaScript Simple Object Notation (JSON)* a *Extensible Markup Language (XML)*. Narozdíl od databází typu

[2] Pojem cluster může mít několik významů, zde je požíván jako množina síťově propojených počítačů.
[3] <https://redis.io/>
[4] <http://basho.com/products/>
[5] <https://github.com/SonarSource/sonar-persistit>
[6] <http://www.project-voldemort.com/voldemort/>

klíč-hodnota umožňují dokumentové databáze přistupovat k dokumentům a analyzovat je dle jejich obsahu. Příkladem mohou být *MongoDB*[7] a *CouchDB*[8].

- *Sloupcové databáze:* Sloupcové databáze se skládají z tabulek, ve kterých může mít každý řádek libovolný počet sloupců (nezávislý na ostatních řádcích). Volné vkládání sloupců sloupců nijak nesnižuje výkon sloupcových databází, které bývají masivně distribuované. Příklady sloupcových databází jsou *HBase*[9], *BigTable*[10] a *Cassandra*[11].

- *Grafové databáze:* Konečně posledním a pro nás nejzajímavějším typem *NoSQL* databází jsou grafové databáze. Tyto databáze jsou určené pro data, která je vhodné modelovat a dotazovat jako grafy (viz. kapitola 2.2). Vnitřní reprezentace grafů může být různá podle toho, pro jaký typ grafových úloh je daná databáze primárně určena. Konkrétní databáze budou popsány v kapitole 2.8.

2.2 Grafy

Ještě před popisem vlastních grafových databází blíže popíšeme grafy jako takové a jejich typy. Nejdříve uvedeme základní matematický rámec teorie grafů, se kterým budeme dále pracovat.

Graf G je trojice $G = (V, E, \epsilon)$, kde V je množina vrcholů (uzlů) a E množina hran. ϵ je přiřazení, které každé hraně přiřazuje:

- množinu dvou vrcholů (koncové vrcholy) pro *neorientovaný graf*.

- uspořádanou dvojici vrzcholů (počáteční a koncový vrchol) pro *orientovaný graf*.

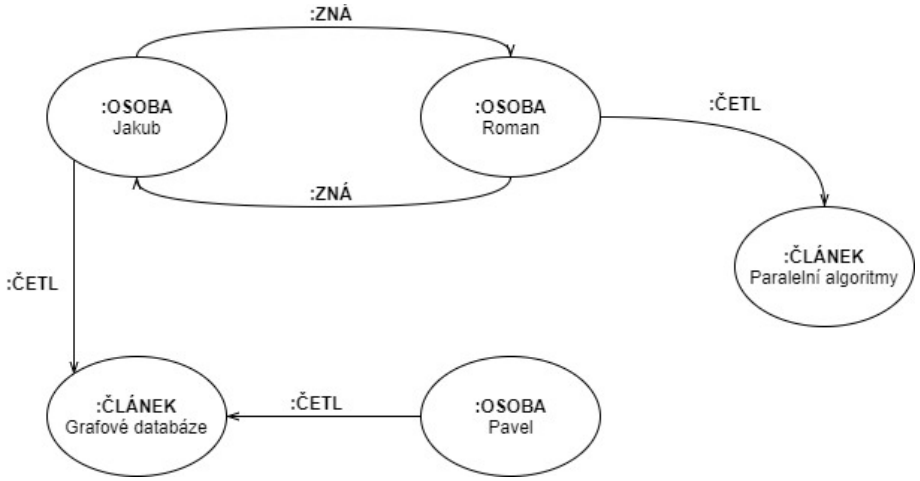
Pokud graf neobsahuje *paralelní hrany (multihrany)*, tedy hrany, které mají stejné počáteční a koncové vrcholy (orientovaný graf), respektive stejné koncové vrcholy (neorientovaný graf), je označován jako *prostý graf*. Naopak pokud graf paralelní hrany obsahuje, je označován jako *multigraf*.[19]

Grafy z pohledu informačních technologií matematickou definici grafu dále rozšiřují. Umožňují definovat typy hran a uzlů, čímž vzniká *ohodnocený graf*. Například ohodnocený graf na obrázku 2.1 obsahuje dva typy uzlů (:OSOBA a :ČLÁNEK) a dva typy hran (:ZNÁ a :ČETL).

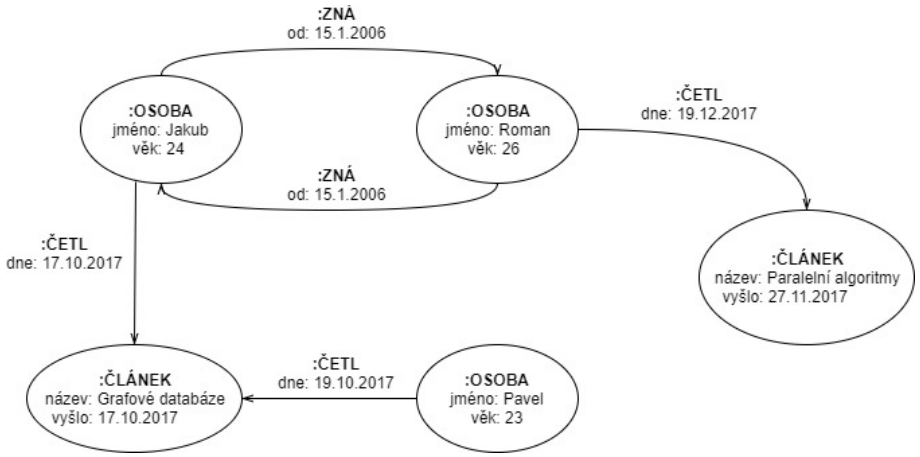
Ohodnocené grafy sice poskytují možnost rozlišit typy hran a uzlů, v mnoha situacích je ale žádoucí zachytit do grafu více informací. *Atributové grafy* umožňují přiřadit hranám a uzlům libovolný počet atributů (dvojic klíč-hodnota), které obsahují informace o daném uzlu, respektive hraně. Příkladem může být věk osob, či datum vydání článku (viz obrázek 2.2). Většina grafových databází pracuje právě s atributovými grafy.[44]

Generalizací grafu je tzv. *hypergraf*. Ten se proti grafu liší tím, že zatímco hrana grafu vede mezi právě dvěma vrcholy, hyperhrana (tedy hrana v hypergrafu) je mnoužinou jednoho

[7] <https://www.mongodb.com/>
[8] <http://couchdb.apache.org/>
[9] <https://hbase.apache.org/>
[10] <https://cloud.google.com/bigtable/>
[11] <http://cassandra.apache.org/>



Obrázek 2.1: Ohodnocený graf



Obrázek 2.2: Atributový graf

a více vrcholů.[20] Hypergrafy jsou používány k reprezentaci dat v mnoha oborech a některé grafové databáze je přímo podporují (například HypergraphDB[12]).

2.3 Reprezentace grafu

Aby bylo možné v dalších kapitolách popsat pokročilé aspekty grafových databází (jako jsou dotazování, indexování, či transakce v grafových databázích), je nejdříve nutné popsat jejich základní principy. Nutnost ukládat data ve formě grafů je starší než grafové databáze samotné, existuje tak mnoho způsobů uložení grafů z nichž ty nejběžnější zde popíšeme. Ještě předtím je nutno poznamenat, k jakým typům úloh jsou grafové databáze zejména využívány. Základními grafovými úlohami jsou *průchod grafu do hloubky (BSF)* a *průchod grafu do šířky (DFS)*. Oba tyto algoritmy musí projít v nejhorším případě[13] všechny hrany a uzly grafu, je tedy vidět, že pro efektivitu grafových databází je klíčová rychlost přechodu z jednodoho uzlu do (všech) jeho sousedních uzlů pomocí existujících hran.

Než se pustíme do popisu nativních grafových úložišť, krátce popíšeme možnost ukládání grafů pomocí relačních databází. Jedním z možných řešení může být dvojice tabulek *UZLY* a *HRANY*. Tabulka uzlů by obsahovala identifikátor uzlu a případné další parametry uzlu, tabulka hrany by obsahovala referenci na počáteční a koncový uzel hrany a případné další parametry hrany[14]. Všechny hrany daného uzlu je tedy možné získat pomocí spojení těchto tabulek pomocí cízího klíče, složitější průchody potom mohou být docíleny pomocí relace tabulky hran na sebe sama. Problém tohoto přístupu je nutnost rekurzivních operací *join* a jejich vysokých nákladů[15]. Tyto problémy mohou být částečně eliminovány použitím sloupcových či dokumentových *NoSQL* databází. Přestože v takovém případě nejsou používány rekurzivní operace spojování tabulek (ve světě *NoSQL* tato operace neexistuje), je k datům přistupováno pomocí prohledávání indexů a operace je tak stále výrazně dražší, než při použití nativního grafového úložiště.[44]

Nejzákladnější nativní reprezentací grafu je tzv. **matice sousednosti**. Ta využívá možnosti reprezentování hran grafu jako dvojice uzlů a reprezentuje graf jako matici M o rozměrech $|V|x|V|$, kde $|V|$ je počet uzlů grafu. Pokud existuje mezi uzly i a j hrana, pak bude $M_{i,j} \neq 0$. Pokud nás zajímá pouze existence hrany, nabývá matice pouze hodnot $\{0, 1\}$. Pokud jsou hrany ohodnocené, pak nenulová hodnota buňky na dané pozici ukazuje nejen existenci hrany mezi danými uzly, ale také její ohodnocení. Pro neorientovaný graf je matice symetrická. Pokud je graf orientovaný, potom $M_{i,j}$ určuje existenci hrany z uzlu i do uzlu j a $M_{j,i}$ naopak existenci hrany z uzlu j do uzlu i. Tato reprezentace grafu zajišťuje vysokou efektivitu přidávání, odstraňování a kontroly existence hran - tyto operace jsou okamžité. Na druhou stranu přidání nového uzlu do grafu je nákladná operace, matice musí být přealokována a překopírována. Vzhledem k tomu, že paměťová náročnost matice sousednosti je kvadratická

[12] <http://hypergraphdb.org/>

[13] Pro konkrétní problémy jsou často BFS a DFS algoritmy upravovány tak, aby bylo možné některé větve průchodu tzv. ořezávat - tedy procházet pouze jejich část, nebo je neprocházet vůbec.

[14] V tomto případě předpokládáme orientovaný prostý atributový graf, pro jiné grafy by bylo nutné tuto strukturu upravit (což ale není těžké). Například v případě ohodnoceného grafu je možné uvažovat separátní tabulku pro každý typ uzlů.

[15] Spojování tabulek pomocí cizích klíčů je jedna z nejdražších operací ve světě relačních databází.

vzhledem k počtu uzlů a nezávisle na počtu hran (není tedy vhodná pro řídké matice[16]), je drahé i její překopírování. Pro nalezení všech sousedních uzlů je nutné všechny uzly projít a zkontrolovat, zda hrana existuje, či nikoliv. Hledání sousedních uzlů je nejčastější operací při průchodu grafu, tato reprezentace tedy není vhodná ani na složitější průchody.

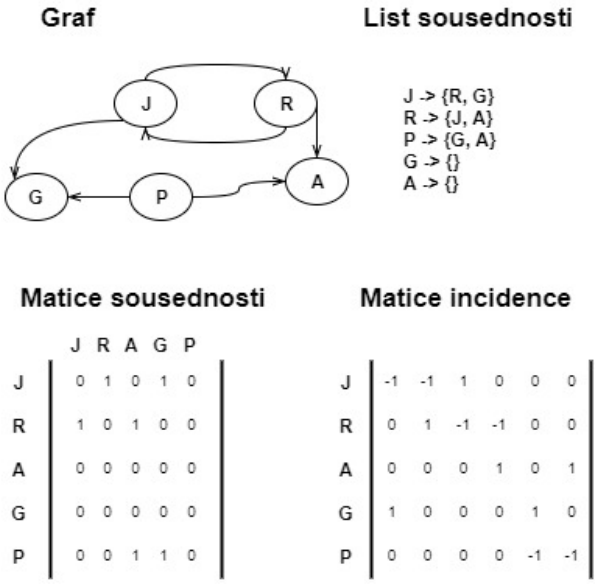
Speciální variantou matice sousednosti je **Laplaceovská matice**. Ta má stejně jako matice sousednosti rozměry */V/x/V/*. Diagonála Laplaceovské matice ukazuje stupeň vrcholu a pokud existuje mezi dvěmi vrcholy hrana, je hodnota matice na dané pozici *-1*. Pokud mezi vrcholy hrana není, hodnota je *0*. Hlavní výhodnou reprezentace grafu Laplaceovskou maticí je umožnění spektrální analýzy grafu[9], kdy jsou pomocí vlastních čísel matice analyzovány vlastnosti grafu.

Další možnou maticovou reprezentací grafu je **matice incidence**. Jedná se o dvoudimenzionální matici o rozměrech */V/x/E/*, kde každý sloupec matice odpovídá jedné hraně a řádek jednomu uzlu. Pokud je uzel *v* účastníkem hrany *e*, potom $M_{v,e}$ *!= 0*. Pokud se jedná o neorientované grafy, nabývá matice pouze hodnot *{0, 1}*. Pokud je graf orientovaný, potom je odchozí hrana kódována jako *-1* a příchozí jako *1*. V případě ohodnoceného grafu mohou být hodnoty obecně všechna celá čísla (podobně jako u matice sousednosti), přičemž opět platí, že záporné číslo značí odchozí hranu a kládné číslo příchozí. Vzhledem k tomu, že pro většinu grafů platí, že počet hran */E/* je vyšší než počet uzlů */V/* (často, například u sociálních sítí, je výrazně vyšší), je tato reprezentace pro svou vysokou paměťovou náročnost ve většině případů nevhodná. Narozdíl od ostatních reprezentací ale umožňuje ukládání hypergrafů (hrany hypergrafu mohou zahrnovat libovolný počet vrcholů). Zatímco u běžného grafu by každý sloupec matice incidence obsahoval právě dva nenulové prvky, u hypergrafu jich může být libovolný počet.

Některé problémy matice sousednosti řeší **list sousednosti**. Ten je implementován jako množina listů, kde každému uzlu připadá jeden list sousedních uzlů. Paměťová náročnost této reprezentace je tedy */V/+/E/*, kde */V/* je počet uzlů a tedy počet listů sousednosti a */E/* je počet hran a tedy celkový počet prvků obsažených v listech sousednosti. List sousednosti je výrazně úspornější reprezentací grafu z paměťového hlediska než je matice sousednosti a je tedy vhodný i pro řídké grafy. Zároveň umožňuje rychlé nalezení všech sousedních uzlů pro daný uzel, v listu jsou vypsány pouze ty uzly, se kterými je uzel propojen a procházení grafu je tak rychlé. Přidání hrany do grafu je přidáním jednoho prvku do listu (respektive dvou listů v případě neorientovaného grafu) a přidání uzlu je také relativně efektivní - znamená vytvoření nového listu a přidání ho do množiny ostatních listů. Drahé jsou naopak operace odebrání hrany (je nutné projít všechny sousedy obou koncových uzlů hrany) a odebrání uzlu (je nutné projít všechny hrany). Také ověření existence konkrétní hrany je relativně drahé (je potřeba opět projít všechny sousedy daného uzlu). Toto může být řešeno alternativní implementací, napříkald mohou být listy sousedů řazeny. Tím se sice sníží složitost ověření existence hrany, ale zvýší se složitost přidání nové hrany. Při reprezentaci velkých grafů pomocí listů sousednosti se používají kompresní techniky, díky kterým je možné dále snižovat paměťovou náročnost.[12] Jednotlivé reprezentace jsou ukázány na obrázku 2.3.

Důležitým kritériem pro výběr reprezentace grafu je *lokalita dat*. Cílem je uložit graf tak, aby jejich fyzické umístění odpovídalo jejich logické vzdálenosti a při průchodu grafem tak

[16] Poznamenejme, že matice reprezentující graf jsou často řídké, jedná se o grafy, kde je na velký počet uzlů poměrně málo hran, příkladem mohou být sociální sítě.



Obrázek 2.3: Reprezentace (orientovaného) grafu

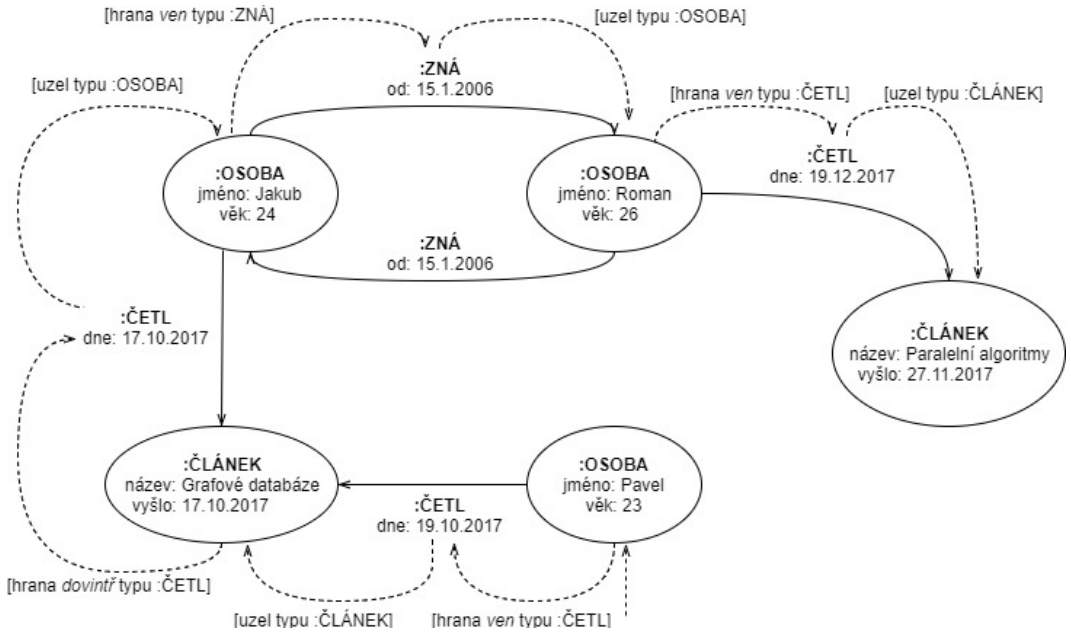
byla data pro další krok načtena co nejrychleji. K dosažení optimality lokality dat reprezentovaných grafem existuje mnoho technik, vzhledem k tomu, že se ale jedná o *NP-těžkou* úlohu, jedná se často o heuristiky a aproximační algoritmy. Některé z používaných metod jsou například metoda minimalizace šířky matice autorů *Cuthill a McKee* [18], *rekurzivní spektrální bisekce* [9], nebo metoda založená na průchodedech grafu do šířky (*Breadth First Search Layout*) [2].

2.4 Typy dotazů

Na rozdíl od indexově náročných množinových operací relačních databází využívají grafové databáze bezindexové místní prohledávání (traverzování) [4]. Relační databáze obsahují data typicky v normalizované formě v několika separátních tabulkách tak, aby bylo zamezeno duplikaci dat. Řádky těchto tabulek mohou být chápány jako objekty popsané svými vlastnostmi. Vztahy mezi objekty jsou definovány primárními a cizími klíči a musí být při dotazování explicitně vytvořeny spojením tabulek pomocí *join* operace. Jak vyhledávání záznamů v tabulkách, tak spojování tabulek je (v ideálním případě) realizováno pomocí vyhledávacích indexů, které časovou složitost těchto operací značně snižují, obecně ale ne dostatečně. Databázové vyhledávací indexy jsou typicky realizovány pomocí vyhledávacích stromů (konkrétně *B-stromů* [43]), vyhledání zánamu má tedy logaritmickou časovou složitost vzhledem k počtu uzlů[17]. Grafové databáze na druhé straně obsahují typicky pouze jednu strukturu - graf[18].

[17] Konkrétně operaci vyhledání uzlu má složitost $\Theta(b \log_b(n))$ a operace vyhledání rozsahu uzlů $\Theta(b \log_b(n) + k/b)$, kde n je počet uzlů, b je úroveň stromu a k je velikost rozsahu.[17]

[18] Grafové databáže mohou obsahovat jeden velký graf, nesouvislý graf, nebo množinu grafů. Z hlediska této práce je toto ale technický detail a pokud nebude explicitně řečeno jinak, budeme předpokládat, že grafová



Obrázek 2.4: Příklad průchodu (*traverzování* grafem)

Ten ve své fyzické reprezentaci obsahuje všechny své hrany, vztahy mezi objekty (vrcholy) v grafu jsou tedy implicitní, není potřeba je při každém dotazu vytvářet. Tato skutečnost s sebou nese výhody i nevýhody. Jednou z podstatných nevýhod je složitost rozdělení grafu na podgrafy a tedy obtížná distribuce grafu (popsáno v kapitole 2.7). Naopak výhodou je instantní čas, ve kterém je možné se v grafu posouvat mezi sousedními uzly. To je natolik výraznou vlastnostní grafových databází, že výrazně ovlivňuje způsob, jakým je přistupováno k datům v grafových databázích. Tím nejčastějším je právě průchod grafem (traverzování, nebo také *Traversal pattern*).

Na začátku průchodu grafu jsou pomocí indexu vyhledány výchozí uzly, ze kterých bude graf procházen. Z nich se dotaz pohybuje po grafu pomocí sousedních hran a uzlů dle zadaných kritérií. Těmi jsou typicky typy hran a uzlů, podmínky na jejich vlastnosti (uvažujeme atributový graf, viz kapitola 2.2) a hloubka prohledávání. Součástí definice dotazu je také zvolení typu průchodu - průchod do hloubky (*DFS*, nebo průchod do šířky (*BFS*. Když jsou projity všechny cesty odpovídající zadaným kritériím, jsou vráceny uzly, ve kterých průchod skončil. Typickými úlohami řešenými pomocí průchodů grafem je například zjištění existence cesty mezi dvěma uzly, nalezení nejktraší cesty, nalezení všech cest a uzlů odpovídajících zadaným kritériím apod. Na obrázku 2.4 je ukázán průchod, který by odpovídal na otázku "Co čtou přátelé lidí, kteří čtou stejný článek jako Pavel".

Dalšími typy dotazou kromě obecného průchodu gravu jsou *dotazy na podgrafy*, *dotazy na nadgrafy* a *dotazy na podobné grafy*. Zadáním těchto dotazů je také graf, označme G_{dotaz}. Tento dotazový graf může obsahovat uzly a hrany, u kterých nejsou specifikovány bližší vlastnosti, v grafu může být určena množina přípustných typů uzlů a hran, nebo například

databáze obsahuje jeden velký graf.

maska jejich typů. Dotazy na podgrafy hledají specifický vzor grafů v grafové databázi - v databázi jsou vyhledávány všechny grafy, který G_{dotaz} obsahují. V případě dotazu na nadgraf naopak hledáme v databázi všechny datové grafy, které jsou v G_{dotaz} obsaženy. U dotazů na podobné grafy je výsledek dotazu vyhodnocován pomocí metriky podobnosti.[40] Tyto typy dotazů jsou vyhodnocovány ve třech krocích:

1. Extrakce: Z grafu G_{dotaz} jsou vyextrahovány indexované charekteristiky (stejným způsobem jako z datových grafů).

2. Filtrace: Porovnáním vyextrahovaných charakteristik jsou z datových grafů vybrány ty, které odpovídají charakteristikám G_{dotaz}. Tím vznikne množina kandidátů na výsledek, přičemž cílem indexační techniky je, aby tato množina obsahovala co nejmenší počet falešně pozitivních kandidátů. Pokud by množina neobsahovala žádné falešně pozitivní kandidáty (indexační technika by měla dostatečnou filtrační schopnost), byla by tato množina rovna množině výsledných grafů (a nebylo by nutno provádět třetí fázi).

3. Verifikace: Grafy z množiny kandidátních řešení jsou podrobně porovnány s G_{dotaz}, čímž jsou vyloučeni případní zbylí falšeně pozivní kandidáti a dostáváme množinu skutečných výsledků.

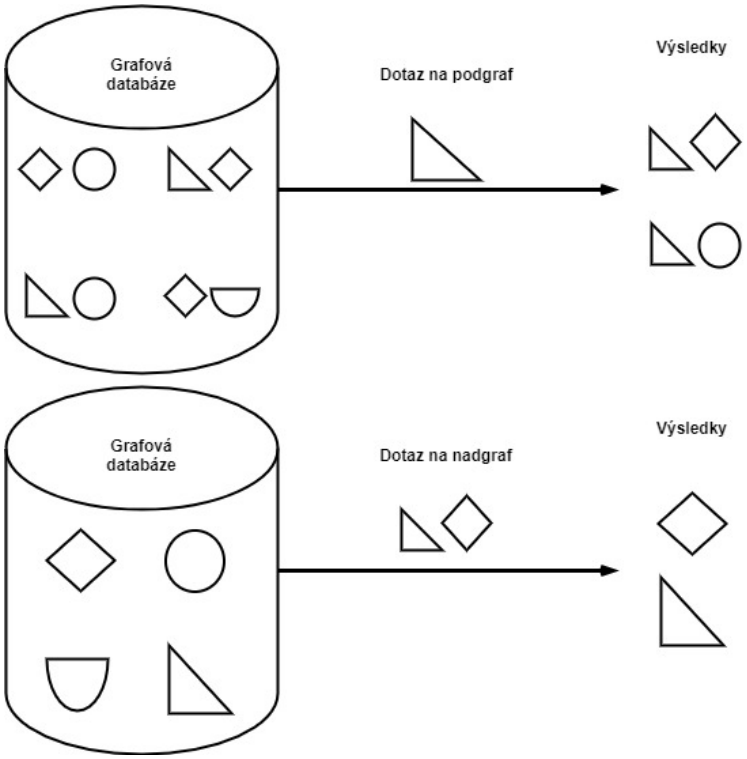
Je zjevné, že tyto typy dotazů jsou velmi závislé na konkrétních indexačních technikách, ty jsou popsány v kapitole 2.5. Grafické znázornění dotazů na nadgraf a podgraf je na obrázku 2.5.

2.5 Indexy

Stejně jako je tomu u relačních databází, i ve světě grafových databází se hojně používají indexy (a také už byly v předchozím textu několikrát zmíněny) a je jich hned několik typů. Průchody (*traverzování*) grafem používají pouze základní indexy a to ve chvíly, kdy jsou hledány výchozí uzly. Konkrétní databáze můžou k indexům přistupovat specifickým způsobem, je tedy vhodné si u každé databáze ověřit, co je automaticky indexováno. Zpravidla je žádoucí indexovat typy (*labely*) a vybrané atributy hran a uzlů. Například pokud by existoval index na typ uzlů a chtěli bychom v grafu o milionu uzlů vyhledat všechny uzly typu *:OSOBA* (a z nich graf procházet), databáze by neprocházela všechny uzly, ale podle indexu by vyhledala přesně uzly tohoto typu. Tyto základní indexy si můžeme představit jako indexy které známe z relačních databází, tedy jako *B-stromy* a jim podobné struktury[19].

U strukturálních dotazů je situace komplikovanější. V případě *dotazů na podgrafy* se indexační metody dělí na tzv. *mining based* (indexy založené na dolování dat (*data mining*)) a *non-mining based*. *Mining based* indexační metody jsou založené na pozorování, že pokud množina vlastností grafu G_{dotaz} není obsažena v některém z datových grafů, nebude takový datový graf obsahovat ani celý dotazový graf. Stačí tedy vybrat vhodné podstruktury grafu G_{dotaz} a vyhodnocovat dotazy pouze na nich (typicky se jedná o časté podstromy či

[19] Implementace indexu v Neo4J (*GB+Tree*): <https://github.com/neo4j/neo4j/blob/3.4/community/index/src/main/java/org/neo4j/index/internal/gbptree/GBPTree.java>



Obrázek 2.5: Znázornění dotazů na nadgraf a podgraf

podgrafy). Pro tyto podmnožiny vlastností je vytvořen *invertovaný index*[20] a při vyhodnocování grafu G_{dotaz} jsou nejdříve identifikovány a vyhodnoceny tyto struktury, čímž je získána množina kandidátů. Nevýhodou tohoto přístupu je závislost *invertovaného indexu* na jak datových, tak dotazových datech. Pokud se výrazně změní typ dotazových grafů, nebo se změní datové grafy tak, že vybrané podstruktury již nemají potřebnou filtrační schopnost, je nutné přeindexovat. Mezi *mining based* indexační metody patří například *GIndex* [82] a *TreePI* [85]. *Non-mining based* indexační metody naopak indexují grafy jako celek, neanalyzují jejich strukturu nabo časté vzory. Nejsou tak závislé na dotazových ani datových grafech, nedosahují ovšem takové filtrační schopnosti jako *mining based* metody a tím pádem vyžadují delší fázi verifikace. Patří mezi ně *GraphGrep* [33], *GDIndex* [79], *GString* [39] a *GraphREL* [61]. Narozdíl od *dotazů na podgrafu* neexistuje pro *dotazy na podgrafy* zatím tolik indexačních technik (a to i přesto, že se jedná o důležitý problém), mezi existující patří *cIndex* [15], *GPTree* [86] a *iGQ* [76]. U *dotazů na podobné grafy* je cílem nalézt grafy, které jsou dotazovému grafu G_{dotaz} podobné, je tedy nutné určit metriku, podle které bude podobnost měřena. Indexačními metodami pro tuto oblast jsou *Grafil* [83], *Closure Tree* [35] a *SAGA* [70].

2.6 Transakce

Pojďme nejprve připomenout standardní transakční model světa relačních databází - **ACID**. Tento model říká, že transakce v databázovém systému musí být atomické (***A**tomocity*, tedy nedělitelné. Pokud tedy selže některá z operací transakce, selže celá transakce a je proved její *rollback*[21]. Nemůže se tedy stát, že by byla provedena jen část operací transakce a databáze se ocitla v nekonzistentním stavu. Jinými slovy, po provedení jakékoliv transakce (úspěšném i neúspěšném) bude databáze vždy konzistentní (***C**onsistency*). Všechny transakce jsou také izolované (***I**solated*), tedy transakce jsou na sobě navzájem nezávislé a neovlivňují si navzájem data. V případě, že dojde ke konfliktu několika transakcí, databázový systém konflikt vyřeší tak, aby byla tato vlastnost zachována[22]. Právě na úrovni izolace konfliktujících transakcí záleží, jakým způsobem bude konflikt vyřešen. Poslední vlastností transakcí v transakčním modelu *ACID* je trvanlivost (***D**urability*). Tato vlastnost zajistí, že změny provedené transakcí jsou trvalé.

Vlastnosti *ACID* transakčního modelu jsou klíčové pro relační databázové systémy, které mají širokou škálu využití. Pro svět *NoSQL* databází (do kterého bývají grafové databáze často řazeny) tento model ale není vhodný, a to především kvůli silnému důrazu na konzistenci dat a tedy jejich obtížné distribuovatelnosti[23]. Zaven je tedy alternativní transakční

[20] Invertovaný index ukazuje pro všechny hodnoty seznam celků, v kterých je hodnota obsažena, v tomto případě tedy pro všechny podstruktury ukazuje v kterých datových grafech jsou obsaženy.

[21] Rollback transakce je událost, která nastává, pokud transakce selže, nebo pokud je uživatelem explicitně vyvolána. Všechny kroky, které byly v rámci transakce provedeny jsou vráceny, po rollbacku transakce je tedy databáze ve stejném stavu, jako před spuštěním transakce. Opakem rollbacku je operace *commit*, která potvrzuje všechny změny provedené v transakci.

[22] Existuje několik stupňů izolace transakcí , výčtem *read uncommitted*, *read committed*, *repeatable read* a *serializable*.

[23] Distribuovanost dat je klíčová pro *NoSQL* databáze obecně, pro grafové databáze to ale ne vždy platí. Toto bude více rozebráno v kapitole 2.7.

model - **BASE**. Ten říká, že systém numusí být nutně dostupný vždy, stačí základní dostupnost (***B**asic **A**vailability*) - tedy dostupnost po většinu času. Mohou nastastat částečné výpadky, ale nikdy nedojde k výpadku celého systému. Tato vlastnost se opět týká především těch *NoSQL* databází, které předpokládají vysokou distribuovanost dat a skládají se z mnoha uzlů. Od grafových databází, které distribuované nejsou očekáváme tedy dostupnost úplnou. Systém je v takzvaném volném stavu (***S**oft-state*), je dynamický, neustále dochází ke změnám. V případě distribuování a replikaci dat je důsledkem této vlastnosti mimo jiné to, že ne všechny repliky musí být nutně vzájemně konzistentní. A konečně, máme jistotu, že systém bude vždy "nakonec"uveden do konzistentního stavu (***E**ventual consistency*), ale nemusí být konzistentní v každém okamžiku. Například k vynucení konzistence může dojít až při čtení dat, při jejich zápisu nutná není. Tento model tedy také umožňuje zachování jistoty konzistence dat, není ale automatická, jako tomu je u předchozího modelu. [60]

Jak už bylo zmíněno, grafové databáze jsou typem *NoSQL* databází. Přesto většina grafových databází není *NoSQL* databází v pravém smyslu slova. Výsledkem tak je, že některé grafové databáze plně podporují *ACID* transakční model, jiné naopak *BASE*. Toto je blíže diskutováno v popisu konkrétních grafových databází (kapitola 2.8).

2.7 Distribuovatelnost

V kapitole 2.1 jsme zmínili, že distribuovanost dat je klíčovou vlastností *NoSQL* databází[24]. Pro distribuci dat se využívá kombinování dvou technik - *rozdělení (sharding)* a *replikace* dat.

Rozdělení dat je základním předpokladem pro jejich distribuci - musí dojít k rozdělení dat na vhodné množiny a ty uložit na různé uzly v clusteru. Při manipulaci s daty je tak přistopováno na jeden nebo více uzlů podle toho, jaká data jsou zpracovávána. Při rozdělování dat je bráno v potaz jejich rovnoměrné rozdělení, minimalizace počtu uzlů využívaných k provádění operací nad daty a optimalizace jejich fyzického rozmístění.

Aby byla při případném výpadku části uzlů stále přístapná všechna data (všechny *shardy*) jsou po částech mezi uzly několikrát replikována. Tím je zajištěno, že při selhání (ať už dočasném, nebo trvalém) jednoho uzlu nedojde ke ztrátě dat samotných, ani ke ztrátě přístupu k nim (pokud neselžou všechny repliky obsahující danou část dat). Počet, kolikrát jsou data v databázi replikována (replikační faktor) určuje mimo jiné algoritmus, pomocí kterého jsou data na repliky ukládána a rychlost jejich čtení.

U grafových databází je ale distribuce dat (a tedy horizontální škálování) výrazně komplikovanější, než u ostatních *NoSQL* databází. Data v nich jsou uložena do několika několika struktur (tabulek, dokumentů, dvojic klíč-hodnota), zatímco obsahem grafové databáze je typicky pouze jediná struktura - graf. Rozdělení grafu na podgrafy je obtížný problém (zvláště pokud se jedná například o úplný graf), proto obecně ne mnoho grafových databází tuto možnost vůbec umožňuje. Přesto existují metody, které se distribucí grafu zabývají. Výsledná distribuce je potom optimální pouze pro podmnožinu grafových úloh (například průchod do šířky, hledání nejmenší kostry atd.). Distribucí grafu na několik uzlů se mění čas, v jakém je možné získat seznam sousedních vrcholů grafu v případech, kdy hrana mezi

[24] Pokud je to možné, je vždy efektivnější když data distribuovaná nejsou. K jejich distribuování přikračujeme až ve chvíli, kdy nás k tomu nutí okolnosti - například velikost dat.

vrcholem a jeho sousedem spojuje dva podgrafy na dvou různých uzlech, je tedy obtížné graf distribuovat při snaze zachovat co nejvyšší rychlost průchodu grafem. Příkladem algoritmů pro rozdělení grafu může být *1D dělení* a *2D dělení* matice sousednosti [84], nebo mnoho technik aproximujících *optimální k-rozdělení grafu*[25].[32] [23] [10]

2.8 Databáze - příklady

V této sekci popíšeme vlastnosti několika nejpoužívanějších grafových databází (relevantních pro projekt *Manta Tools* a tedy i tuto diplomovou práci). Dlouhodobě nejoblíbenější grafovou databázi na trhu [66] je **Neo4j**[26] Databáze je implementovaná v programovacím jazyku *Java*, má *nativní grafové úložiště*, nabízí *horizontální škálovatelnost* a také plnohodnotný *ACID transakční model*. Dotazování dat můžebýt prováděno pomocí objektového *Java API*, nebo jazyků *Cypher* a *Gremlin* (viz 2.9). Databáze *Neo4j* je licencována pod *GPLv3*[27] licencí.

Další oblíbená databáze, **OrientDB**[28] podporuje hned několik způsobů práce s daty - může být používána jako databáze grafová, dokumentová, objektová, nebo databáze typu klič-hodnota (označuje se jako *multi-modelová*) [55]. Uzly grafů jsou reprezentovány jako (*JSON*) dokumenty, hrany mohou být buďto také dokumenty(*regular*), nebo mohou být pouze parametrem uzlů (*lightweight*). Nejedná se tedy o *nativní reprezentaci grafu*, jak je popsána v kapitole 2.3. Stejně jako *Neo4j* je i *OrientDB* implementovaná v programovacím jazyce *Java* a umožňuje *horizontální škálování*. Výchozím transakčním modelem je *ACID*, od verze *2.1.7* je ale možná konfigurace transakčního modelu *BASE* ve dvou režimech řešení kolizí [54], což má pozitivní dopady na rychlost databáze. S databází je možné komunikovat pomocí *HTTP REST/JSON API*, *Java API* a jazyka *Gremlin*. Krom těchto možností nabízí *OrientDB* ještě vlastní dialekt jazyk *SQL*. Licenční politika je více přívětivá komerčnímu použití než je tomu u *Neo4j*, *OrientDB* je licencováno pod *Apache 2.0*[29] licencí. Databází s velice podobonými charakteristikami jako má *OrientDB* je **ArangoDB**[30], hlavním rozdílem je, že jádro *ArangoDB* je implementováno v *C++* a dle vlastních testů [6] je (zvláště pro některé typy úloh) výrazně efektivnější.

Titan se oproti předchozím databázím výrazně liší, nejedná se plnohodnotnou databázi s vlastním úložištěm, ale "pouze"a softwarovou mezivrstvu, která poskytuje funkcionalitu grafových databází jako nadstavbu nad jinými *NoSQL* databázemi. Těmi jsou *Apache Cassandra*[31], *Apache HBase*[32] a *Oracle BerkeleyDB*[33]. *Titan* je zaměřen (jak už napovídají nabízené *back-end* databáze) na masivní škálovatelnost grafových databází, zároveň ale nabízí oba transakční modely - *ACID* i *BASE*. Dotazování dat v Titanu je možné pomocí *Java*

[25] Optimální bisekce grafu je NP-obtížný problém [29], tedy i optimální k-rozdělení grafu je nejméně NP-obtížný problém. Proto jsou používány aproximační algoritmy.

[26] <https://neo4j.com/>

[27] <https://www.gnu.org/licenses/gpl-3.0.en.html>

[28] <http://orientdb.com/>

[29] <https://www.apache.org/licenses/LICENSE-2.0>

[30] <https://www.arangodb.com/>

[31] <http://cassandra.apache.org/>

[32] <http://hbase.apache.org/>

[33] <http://www.oracle.com/technetwork/database/database-technologies/berkeleydb>

API, nebo jazyka *Gremlin*. V roce 2015 proběhla akvizice *Titanu* (respektive společnosti *Aurelius*, která ho vyvíjela) společností *DataStax*, čímž prakticky skončil vývoj této databáze. *Open-source forkem* a pokračovatelem[81] **Titanu** je databáze *JanusGraph*[34]. Obě databáze jsou implementovány v programovacím jazyce *Java* a licencovány pod *Apache 2.0* licencí.

2.9 Dotazovací jazyky

Do roku 2018 nebyly zatím definovány standardy pro dotazování dat v grafových databázích. Existuje mnoho způsobů a nástrojů, které dotazování dat umožňují, většina z nich je ale spjata s konkrétní databází. Některé dotazovací jazyky jsou sice již podporovány více databázemi, v podstatě žádný ale zatím nelze prohlásit za obecně používaný standard. Základním způsobem dotazování dat v kontextu grafových databází jsou *programová rozhraní*, nebo případně *REST API* nezávislá na konkrétním programovacím jazyku. Jednoznačnou výhodou tohoto přístupu je, že tato *API*, která má každá grafová databáze své, podporují všechny funkce dané databáze. To je zároveň i hlávní nevýhodou - grafové databáze fungují na různých, často protichůdných principech (např. tranasakční a netransakční databáze, distribuované a nedistribuované atd.), a jejich *API* se tedy liší. Není tedy možné předpokládat, že průchody grafem napsané pro jednu databázi budou fungovat na druhé[35].

Tyto překážky v rozdílnosti grafových databází se snaží vyřešit hned několik dotazovacích jazyků, ty postupně vznikaly spolu s tím, jak se grafové databáze vyvíjely. Tyto jazyky lze dělit mnoha způsoby, jedním z nejdůležitějších je dělení na *deklarativní* a *imperativní*. Zatímco *imperativní* jazyky formulují řešení problému pomocí pousloupnosti příkazů a určují tak přesný algoritmus řešení problémů, *deklarativní* jazyky naopak způsob řešení problému nepopisují vůbec, ale soustředí se na popis požadovaného výsledku - deklarují, jaký má být výsledek (použitý algoritmus tedy určuje interpret daného jazyka).[14]

Jedním z nejpoužívanějších grafových dotazovacích jazyků je **Gremlin**. První verze *Gremlina* vyšla 25. 12. 2009 [72] jako součást *open-source* projektu *TinkerPop*. *Gremlin* byl uveden jako *turingovsky kompletní* programovací jazyk pro práci s atributovými grafovými databázemi. Umožňoval *Create Read Update Delete (CRUD)* operace, různé způsoby průchodů grafem a operace s vyhledávacími indexy. V této verzi podporoval ze zavedených databází pouze databázi *Neo4j*. V roce 2010 bylo uvedeno rozhraní *Blueprints* [71], jehož implementací grafová databáze získává podporu jazyka *Gremlin*. Ve verzi *Gremlina 2.6.0* [73] už jsou *Blueprints* poměrně široce podporovány[36]. To je ale také poslední verze vydaná pod samostatným projektem *TinkerPop*, poté se celý projekt stává součástí *Apache software foundation*[37]. To znamená několik změn v projektu, ve verzích *3.x* [5] je pozměněna systaxe jazyka a také se mění *API* pro podporu *Gremlina* na *Gremlin Structure API*. Od této verze podporuje *Gremlin* také *deklarativní* dotazování dat (do té doby byly všechny dotazy pouze *imperativní*). Mezi databáze podporující Gremlin *3.x* patří *Neo4j, DataStax, InfiniteGraph,*

[34] <http://janusgraph.org/>

[35] Toto zcela neplatí ani u *SQL*, které je jednoznačným standardem ve světě relačních databází. Rozdíl je ale v tom, že zatímco různé dialekty *SQL* se liší (pokud se liší) zpravidla pouze málo a pouze v syntaxi jazyka, u *API* různých grafových databází mohou být rozdíly mnohem zásadnější.

[36] Ve verzi *2.6.0 Gremlin* podporují databáze *Neo4j, OrientDB, Sparksee, Titan, Faunus, InfiniteGraph, Rexster*, a další.

[37] <http://www.apache.org/>

JanusGraph, Cosmos DB a další. V příkladu 2.1 je ukázán průchod grafem v *Java* notaci jazyka *Gremlin*.

```
/*
 * Gremlin (Java) traversal
 */
g.V().match(
  as("a").has("name","gremlin"),
  as("a").out("created").as("b"),
  as("b").in("created").as("c"),
  as("c").in("manages").as("d"),
    where("a",neq("c"))).
  select("d").
  groupCount().by("name")
```

Příklad 2.1: Gremlin (Java) - průchod grafem

Dalším velice oblíbeným dotazovacím jazykem je **Cypher** [50], a to především díky jeho podobnosti s *SQL*. *Cypher* je *deklarativní DSL jazyk*, který vznikl jako součást Neo4j. Vzhledem k tomu, že se jedná o *deklarativní* jazyk, snaží se co nejčitelněji popsat[38], co má být výsledkem dotazu a naopak se zaměřuje na to, jakým způsobem bude dotaz proveden. Přesto je možné vykonávání dotazů ovlivnit pomocí tzv. *hintů* (například lze ovlivnit použití indexů). *Cypher* si získal mezi uživateli značnou oblibu, vznikla proto iniciativa rozšířit ho i mezi další grafové databáze a pokusit se ustanovit *Cypher* jako standardní dotazovací jazyk pro grafové databáze. Výsledkem této iniciativy byl vznik *OpenCypheru* [51] v roce 2015. Podpora tohoto jazyku ale zatím není příliš široká.[39] V příkladu 2.2 je ukázka *Cypher* dotazu.

```
-- Cypher dotaz

MATCH (you {name:"You"})
MATCH (expert)-[:WORKED_WITH]->(db:Database {name:"Neo4j"})
MATCH path = shortestPath( (you)-[:FRIEND*..5]-(expert) )
RETURN db, expert, path
```

Příklad 2.2: Ukázka Cypher dotazu

Existují i další dotazovací jazyky pro grafové databáze než výše zmíněné, většina z nich je ale navržena pro specifické účely a pro obecné použití se nehodí, a nebo nejsou příliš rozšířené.

[38] Syntaxe *Cypheru* je inspirována syntaxí jazyka *SQL* a také takzvaným *ASCII artem* - <https://cs.wikipedia.org/wiki/ASCII_art>

[39] *OpenCypher* v tuto chvíli podporují spíše méně známé grafové databáze: <http://www.opencypher.org/projects>.

Kapitola 3

Softwarové abstrakce

Vývoj softwaru je dnes jistě v mnoha ohledech jednodušší, než v padesátých letech dvacátého století, kdy programátor musel znát strojový kód, počty a velikosti registrů a v nejhorších případech musel fyzicky propojit vodiče mezi výpočetními jednotkami. Postupem času s příchodem nových programovacích jazyků byl programátor stále více abstrahován od hardwaru a i některých softwarových vrstev nad ním. Vývojáři webových stránek stačí vhodný operační systém, aplikační a databázový server k tomu, aby mohl rychle vytvořit webové stránky, nepotřebuje k tomu přitom podrobnou znalost všech používaných technologií. Stejně tak Java vývojář dnes již nebude psát celou aplikaci s pomocí Javy samotné, ale využije mnoho existujících open-sourcových knihoven, které mu práci výrazně usnadní. Není tedy nutné, aby uměl vývojář vyřešit každý problém, pro řešení některých stačí přepoužití vhodných nástrojů. To má za následek nejen usnadnění práce při implementaci, ale také celkově čistší a přehlednější kód aplikace - kód obsahuje pouze aplikační logiku a nezbytné množství obslužných rutin. Přehledný kód potom vede k jednoduššímu vývoji, testování a údržbě aplikace. Lze tedy říci, že vhodně zvolené abstrakce na různých úrovních - ať už se jedná o architekturu celé aplikace, či návrh několika spolupracujících tříd, jsou pro softwarové systémy klíčové.

"Software je postaven na abstrakcích. Když vyberete ty správné, programování bude přirozeně vyplývat z návrhu, moduly budou mít minimalistická a jednoduchá rozhraní a nová funkcionalita bude dobře zapadat bez zásadních změn. Pokud vyberete špatné, programování bude řada ošklivých překvapení, rozhraní budou nucena vyhovět neočekávaným požadavkům, stanou se nemotorná a i nejjednodušší změna bude obtížně dosažitelná. Systém postavený na chybných konceptech nemůže zachránit žádné množství úprav, pouze jeho nové vytvoření od začátku." [38]

Základním principem zajišťující abstrakci na úrovni zdrojového kódu aplikací je *zapouzdření* - tedy princip, který říká, že stav objektů by neměl být přímo přístupný, ale mělo by k němu být přistupováno pomocí operací k tomu určených. Tím je zaručeno, že stav objektu bude upravovány pouze v rámci daných omezení (například viditelnost polí, číselný rozsah atd). Stejně tak je žádoucí, aby nebyla veřejně vystavována vnitřní implementace softwarových celků - tříd, knihoven, modulů, atd. Snažíme se tedy abstrahovat uživatele od toho, jak je takový celek vnitřně implementován a nabízíme mu pouze rozhraní pro práci s ním - *Appplication Programming Interface (API)*.

Cílem této kapitoly je popsat pro práci relevantní principy sloužící k zavedení vhodných úrovní softwarové abstrakce. Konkrétně popisuje *API* jako obecný nástroj k oddělení různých částí aplikace (kapitola 3.1); vícevrstvou architekturu (kapitola 3.2.1), jejíž návrh pro definovaný problém je jedním z cílů této diplomocé práce; a architektonický styl *REST* (kapitola 3.2.2), který je dnes jedním ze standardů pro architektury webových aplikací. Na závěr jsou popsány principy architektur určených pro *cloudové* systémy (kapitola 3.2.3). Ty jsou již několik let sílícím trendem a bylo by krátkozraké je nevzít v potaz.

3.1 API

API je jasně definovaným rozhraním, které poskytuje uživateli přístup k funkcím softwarové jednotky, kterou obaluje. Vzhledem k tomu, že *API* je obecný pojem a má v softwarovém inženýrství široké uplatnění (které je v této kapitole alespoň částečně popsáno), neexistuje obecně uznávaná definice, která by vymezovala, z čeho se *API* skládá. Obecně je *API* souborem specifikací programových rutin a datových struktur.

Důležitou charakteristikou *API* je, zda se jedná o privátní, nebo veřejné *API*. *Privátní API* je pouze technickým nástrojem pro modularizaci aplikace a přistupují k němu pouze jiné části aplikace. Takové *API* může být postupným vývojem aplikace upravováno bez dopadů na další (okolní) systémy. *Veřejné API* naopak může být používáno mnoha různými spotřebiteli (v okamžik uveřejnění *API* poskytovatel často ztrácí kontrolu nad tím, kdo *API* používá), což má pro jeho charakteristiku několik dopadů. Tím nejdůležitějším je, že je velmi obtížné měnit jednotlivé části veřejného *API* - vždy existuje někdo, kdo je používá. Musí být tedy kladen ještě větší důraz na návrh a testování takového *API*. Speciálním případem veřejných *API* je *Service Provider Interface (SPI)*, což je *API* poskytované bez implementace, ta je zajištěna třetími stranami.

API jsou používána na různých úrovních softwarové abstrakce. Nejblíže hardwaru jsou **lokální API** sloužící pro komunikaci mezi aplikacemi a různými vrstvami operačních systémů. Příkladem takového *API* je například *POSIX* [37] - rodina standardů, jejímž cílem je zajištění kompatibility mezi operačními systémy.

Na úrovni konkrétních programovacích jazyků jsou používána *API* sloužící jako rozhraní poskytující funkcionalitu softwarové knihovny (například *Log4j*[1]), frameworku (například *Java Collections*[2]), konrétní vrstvy či modulu aplikace. Zde neexistuje souhrné poujmenování, respektive, jako kategorie zde slouží programovací jazyky, pro které je *API* určeno - **Java API**, **Scala API**, atd. Tato *API* bývají často používána pro svou přímost, omezují ale své použití na prostředí homogenní z pohledu programovacího jazyka.

Spolu s rozvojem počítačových sítí a později také internetu vznikla poptávka po možnosti vzdáleného volání aplikací (nebo částí aplikací). Jedním z průkopníků v této obasti je *Remote Procedure Call (RPC)*, technologie umožňující vykonání kódu, který je fyzicky na jiném zařízení, než volající program. Teoretické návrhy *RPC* pochází ze sedmdesátých a první implementace z osmdesátých let dvacátého století. V roce 1998 se objevil soubor pravidel, který popisuje jak využívat technilogie *RPC* a *XML* k vzdálenému volání procedur přes internet - *XML-RPC* [80]. *XML* poskytuje slovník pro popis *RPC* volání přenášená mezi

[1] <https://logging.apache.org/log4j/2.x/manual/api.html>
[2] <https://docs.oracle.com/javase/tutorial/collections/>

počítači, technologie tak prakticky umožňuje vytváření API pro komunikaci dvou a více nehomogeních prostředí [64]. Nástupcem *XML-RPC* se stal *SOAP*, který je postavený na podobných principech jako *XML-RPC*. Cílem *SOAP*u je zajistit rozšiřitelnost, neutralitu a nezávislost komunikace. Specifikace *SOAP*u byla zveřejněna v roce 1999, ale až verze 1.2 se v roce 2007 stala doporučením *W3C*[3] [75]. Souběžně byl definován architektonický styl *REST* (kapitola 3.2.2), který se *SOAP*em několik let soutěžil o převahu a v současné době je *de-facto* standardem pro architektury webových aplikace a tedy i **webových API**.

Bez ohledu na účel, kterému *API* slouží, jeho návrh by se měl řídit několika obecnými pravidly [11]. Dobré *API* by mělo:

- *být snadno použitelné*: *API* by mělo být intuitivní a funkcionalita poskytovaná *API* by měla být přístupná jednoduše. *API* by také mělo být minimální, tj. mít co nejméně veřejných prvnků a mělo by mít jasnou a svému účelu adekvátní jmennou konvenci. Takové *API* je jednoduché na pochopení, zapamatování a ladění.
- *být kompletní*: *API* by mělo pokrývat veškerou předpokládanou funkcionalitu tak, aby nemuselo být pro nekomplenost nijak obcházeno. Toto je v konfliktu s principem minimality, obě vlastnosti musí být dobře vyváženy.
- *být rozšířitelné*: *API* by mělo být možné v budoucnu rozšiřovat bez změn na stávající části *API* (veřejná, ale i privátní, *API* je velice komplikované zpětně upravit).
- *vést k čitelnému kódu*: *API* by mělo umožnit psaní přehledného, čitelného a efektivního kódu. Naopak by jeho použití nemělo vést k nutnosti přidávání zbytečného kódu (*boilerplate code*).

3.2 Softwarové architektury

Softwarová architektura je úroveň návrhu softwaru, která sahá za hranice konkrétních algoritmů a datových struktur výpočtu. Navrhuje a specifikuje obecnou strukturu systému včetně komunikace, synchronizace a přístupu k datům. Architektura přiřazuje funkcionalitu jednotlivým návrhovým elementům, předurčuje jejich fyzickou distribuci, kompozici, škálovatelnost, výkon a také jejich možné alternativy [31]. Jádrem softwarové architektury je princip abstrakce - skrývání některých detailů pomocí zapouzdření za účelem lepší identifikace vlastností návrhových elementů [62]. Komplexní systém může mít mnoho úrovní abstrakce a mnoho operačních fází, přičemž každá z nich může mít vlastní architekturu [42].

Formálně jsou softwarové architektury popisovány pomocí konfigurací tzv. *architektonických elementů* (*komponent*, *konektorů* a *dat*) a omezování jejich vzájemných vztahů za účelem dosažení požadovaných vlastností softwarové architektury. *Komponenty* jsou abstraktní softwarové jednotky skládající se z instrukcí a vnitřního stavu, které poskytují transformace dat pomocí svých rozhraní. *Konektroy* jsou abstraktní mechanismy, které zajišťují komunikaci, koordinaci a spolupráci mezi komponentami a data jsou jednotky informace posílané mezi komponentami. [63]

Vzhledem k tomu, že softwarové architektury ztělesňují funkční i nefunkční požadavky, může být složité porovnat přímo architektury určené pro různé typy systémů. Proto jsou

[3] https://www.w3.org/

definovány *architektonické styly*, podle kterých je možné konkrétní softwarové architektury kategorizovat [52]. Architektonický styl tedy určuje, jaké komponenty a konektory mohou být použity architekturami, které jsou instancemi daného stylu, a omezují způsoby jejich kompozice [30].

Architektura softwaru (resp. architektonický styl) implikuje řadu klíčových vlastností softwaru [13], mezi nejdůležitější patří:

- *Výkon*: Výkon je jedním ze zásadních důvodů, proč je třeba se soustředit na architekturu softwaru. Použití vhodné architektury často znamená dopad na výkon softwaru. Na výsledný výkon má vliv mnoho faktorů, mezi nimiž je například *rychlost a efektivita komunikace* (často, ale ne vždy, se jedná o síťovou komunikaci), vlastní algoritmická složitost problému a další. Z uživatelského pohledu lze výkon měřit veličinami jakou jsou *latence* (čas mezi vyvoláním akce a prvním signálem značícím, že akce je vykánána, nebo se vykonává) a *completion time* (čas potřebný na vykonání akce).

- *Škálovatelnost*: Škálovatelnost je vlastnost, která úzce souvisesí s výkonem a je silně ovlivněná architekturou systému. Je-li systém škálovatelný, musí být schopen zvládnout velké množství komponent a interakcí mezi nimi bez zásadního dopadu na výkon. Toho může být dosaženo decentralizací interakcí, distribucí služeb a omezením závislostí mezi komponentami. V současné době je na škálovatelnost kladen velký důraz v souvislosti s *Cloud Computingem* [48] a ta se tak často stává zásadním faktorem ovlivňujícím výběr architektury.

- *Jednoduchost*: Primárním způsobem zajištění jednoduchosti softwaru je princip oddělení zájmů komponent. Pokud může být funkcionalita přidělena komponentám takovým způsobem, že jsou jednotlivě podstatně méně složité než jejich kompozice, budou komponenty jednoduše pochopitelné a implementovatelné.

- *Modifikovatelnost*: Modifikovatelnost říká, jak jednoduché, či složité je stávající architekturu upravit. Modifikovatelnost je dána několika faktory: *vyvíjitelnost* určuje, zda je možné vyvíjet jednotlivé komponenty nezávisle na ostatních; *rozšiřitelnost* určuje, zda je možné do systému přidávat novou funkcionalitu bez dopadu na zbytek systému; *přizpůsobitelnost* určuje, zda může být chování komponenty dočasně upraveno, nebo změněno pro jednoho klienta bez dopadu na ostatní klienty; *konfigurovatelnost a přepoužitelnost* určuje, zda může být komponenta upravena po její implementaci (*konfigurovatelnost*) a v různých konfiguracích přepoužívána v různých aplikacích bez vzájemného ovlivnění aplikací (*přepoužitelnost*).

- *Spolehlivost*: Z pohledu architektury softwaru spolehlivost vyjadřuje schopnost systému zvládnout chyby jeho dílčích částí - komponent, konektrorů, nebo dat.

- *Viditelnost*: Viditelnost se vztahuje k schopnosti komponenty monitorovat či usměrňovat interakce mezi dvěmi různými jinými komponentami systému a umožnit tak zlepšení výkonu pomocí zavedení *cache*, škálování pomocí přidání víceúrovňových služeb, nebo zvýšení zabezpečení systému například přidáním *firewallu*.

- *Přenositelnost*: Pokud je systém přenositelnýný, může být spuštěný na různých prostředích.

3.2.1 Vícevrstvá architektura

Vícevrstvá architektura (alias *layered architecture*) je jedním z nejpoužívanějších architektonickýc stylů z toho důvodu, že v jejím centru čato stojí databáze, což je přirozenou vlastností mnoha aplikací [58]. Aplikace je rozdělena do několika vrstev, každá vrstva má vlastní specifický účel, který neplní žádná z ostatních vrstev a naopak, tato vrstva neplní žádné jiné úkoly. Každá vrstva poskytuje služby vrstvě nad sebou a využívá služeb vrstvy pod sebou [31]. Jednotlivé vrstvy jsou izolovány a komunikují mezi sebou pomocí dobře definovaných *API*. Jejich implementace je skryta a při změně některé z vrstev tak nejsou ostatní vrstvy ovlivněny vůbec, nebo nejhůře v případě změny *API* vrstvy je ovlivněna pouze nejbližší vyšší vrstva. To výrazně snižuje *coupling* a zjednodušuje vývoj, údržbu a testovaní aplikace. Příkladem mohou být komunikační modely *ISO/OSI* a *TCP/IP* [87].

V kontextu webových aplikací je tento architektonický styl spojován se stylem *klient-server* a vzniká *vícevrstvý klient-server* architektura. Architektury postavené na tomto stylu jsou nazývány jako *n-tiered* architektury (*3-tiered, 4-tiered*, atd.) [74]. Architektonický styl obecně nespecifikuje, kolik a jaké vrstvy má obsahovat, často se ale jedná o následující:

- *Prezentační vrstva (presentation layer)*: vrstva aplikace sloužící k interakci uživatele s aplikací - uživatelské rozhraní aplikace
- *Vrstva byznys logiky (bussiness layer)*: část aplikace obsahující vlastní logiku aplikace (podmínky, za kterých mohou být data vytvářena, ukládána atd.)
- *Perzistentní vrstva (persistence layer)*: poskytuje přístup k datům uloženým v perzistentním úložišti (databázi) a umožňuje manipulaci s nimi.
- *Databázová vrstva (database layer)*: server realizující úložiště dat (relační databáze, grafová databáze, atd.)

Vrstvy jsou typicky *zamčené*, je tedy nutné, aby vrstva při zpracování požadavku pracovala pouze se sousedními vrstvami, není možné žádné vrstvy přeskakovat. Jsou ale případy, kdy je vhodné nechat některé vrstvy *odemčené* a mít tedy možnost je přeskakovat. Například pokud chceme umožnit aplikacím třetích stran používat aplikaci jako *webovou službu*, necháme *prezentační vrstvu* odemčenou a umožníme tak přístup na *servisní vrstvu (service layer)*, která potom komunikuje s *vrstvou byznys logiky* (pokud tyto dvě vrstvy nejsou spojeny v jednu). Dalším uplatněním odemčených vrstev může být zvýšení efektivity, režie při zpracování dat způsobená jejich předáváním přes několik vrstev je primární nevýhodou vícevrstvé architektury [16].

3.2.2 REST

Representational State Transfer (REST) je architektonický styl, který ve své dizertační práci v roce 2000 popsal Thomas Fielding [24]. Jeho cílem je definovat standard pro škálovatelné webové aplikace. Styl je definovaný několika architektonickými omezeními:

- *Architektura klient-server: REST* je určen pro webové aplikace a základní architektonická omezení tak přebírá od architektonického stylu *klient-server*, jehož hlavním

principem je rozdělení zodpovědností na část uživatelského rozhraní (*klient*) a část aplikační logiky a persistence dat (*server*). Systém je tak lépe škálovatelný díky zjednodušení separátně modifikovatelných komponent [56] a uživatelské rozhraní jednoduše přenositelné napříč různými platformami.

- *Bezstavovost:* Komunikace mezi *klientem* a *serverem* je bezstavová, *server* tedy nemůže mezi dvěmi různými požadavky přechovávat žádný kontext *klienta*. Je tak zaručeno, že požadavky jakéhokoliv *klienta* budou vždy obsahovat všechny informace nutné k obsluze požadavku a kontext těchto požadavků bude udržovat pouze *klient*. Komunikaci je díky bezestavovosti čitelnější, splehlivější a škálovatelnější.

- *Schopnost používat cache:* Pro zlepšení efektivity síťě je přidán také požadavek na používání *cache*. Tento požadavek vyžaduje, aby všechna data odesíláná odpovědí na požadavky byla explicitně označena jako *cacheable* a *non-cacheable*. Pokud je odpověď požadavku *cacheable*, potom může být *klientem* uložena do *cache* a přepoužita v případě volání identického požadavku.

- *Vícevrstvá architektura:* Dalším architektonickým omezením je použití vícevrstvé architektury (popsána v kapile 3.2.1). Ta zaručuje omezení složitosti systému (na složitost jedné softwarové vrsty) a umožňuje zapouzdření *legacy* služeb.

- *Jednotné rozhraní:* Důležitým architektonickým omezením *REST*u je jednotné rozhraní mezi komponentami. To vyžaduje posílání dat v serializované formě, čímž je zaručeno zjednodušení architektury a snížení implementační závislosti. To je vyváženo snížením efektivity způsobeným režií serializace a deserializace dat.

- *Code on demand architektura:* Posledním skupina omezení je přebrána z architektonického stylu *Code on demand* [27], kde *klient* sice má přístup ke zdrojům, ale neví, jak mají být zpracovány. *REST* tak umožňuje aby funkcionalita *klienta* byla rozšířena stažením a spuštěním kódu ve formě *appletů* nebo *skriptů* ze *serveru*. Tento přístup ale snižuje čitelnost aplikace a je tak pouze volitelným omezením *RESTu*.

V seznamu architektonických omezení je důležitý bod a totiž, že vlastní komunikace mezi klientem a serverem je bezestavová. Nositeli stavu jsou tzv. *hypermedia* (která mohou být realizována hned několika formáty - stejně jako není *REST* spjatý s konkrétním komunikačním protokolem (i když jím je často *HTTP/HTTPS*)), není spjatý ani s konkrétním formátem zpráv (i když jím je často *JSON*). *Hypermedia* poskytují nejen vlastní odpověď na dotaz (data), umožňují také dynamickou navigaci systémem pomocí odkazů na zdroj vrácených entit. To je výrazný posun oproti architektonickému stylu *SOA*[4] , který má podobné cíle jako *REST*, ale rozhraní mezi klientem a serverem je pevně definované pomocí *WSDL*[5], které musí být předáno alternativním komunikačním kanálem. V příkladu 3.1 (převzato z [67]) je ukázka odpovědi serveru na požadavek na seznam produktů využívající *hypermedia* ve formátu *JSON*.

[4] *SOA (Service Oriented Architectures) je architektonický styl, jehož implementací je například SOAP.*

[5] *WSDL (Web Services Description Language) je jazykem popisujícím webové služby tak, aby mohly být využívána způsobem definovaným SOA. Zpravidla bývá používán spolu s protokolem SOAP.*

```
{
    "content": [ {
        "price": 499.00,
        "description": "Apple tablet device",
        "name": "iPad",
        "links": [ {
            "rel": "self",
            "href": "http://localhost:8080/product/1"
        } ],
        "attributes": {
            "connector": "socket"
        }
    }, {
        "price": 49.00,
        "description": "Dock for iPhone/iPad",
        "name": "Dock",
        "links": [ {
            "rel": "self",
            "href": "http://localhost:8080/product/3"
        } ],
        "attributes": {
            "connector": "plug"
        }
    } ],
    "links": [ {
        "rel": "product.search",
        "href": "http://localhost:8080/product/search"
    } ]
}
```

Příklad 3.1: Hypermedia zpráva ve formátu JSON

Podle [25] jsou *hypermedia* nejvyšší úrovní (level 3) využití *REST* architektury. Základem tohoto modelu je použití *HTTP* jako komunikačního protokolu (level 0), zavedení unkátně identifikovatelný (pomocí *URI*) zdrojů namísto jednoho přístupového bodu (level 1), a sémanticky správné použití *HTTP* požadavků - především *GET, HEAD, POST, PUT a DELETE* (leve 2). Každý odkaz obsažený v *hypermedia* odpovědi by měl implementovat *HTTP* požadavky (nebo jejich podmnožinu) dle jejich sémantiky a poskytnutí odkazu tak stačí k tomu, aby se mohl klient službou jendoduše navigovat. *API*, která poskytují služby splňující tyto podmínky jsou označována jako *RESTfull APIs*.

3.2.3 Architektury cloudových aplikací

Cloudové systémy jsou sílícím trendem posledních let, proto má také, stejně jako mnoho pojmů ze světa softwaru, mnoho neustálených definic. *NIST*[6] přiřazuje cloudovým systémům následující vlastnosti [48] :

- *Škálovatelnost na vyžádání*: Systém může jednostranně upravit výpočetní parametry (například serverový čas, velikost síťového úložiště, atd.) automaticky podle potřeby, bez nutnosti kontaktování poskytovatele/poskytovatelů služeb.

[6] National Institute of Standards and Technology - <https://www.nist.gov/>

- *Všeobecný síťový přístup*: Systém je přístupný po síti (často přes internet) pomocí standardních mechanismů podporujících heterogenní klientské platformy.
- *Pooling zdrojů*: Výpočetní zdroje (výpočetní výkon, paměť, síťová infrastruktura) poskytovatele jsou připraveny pro využití různými klienty s možností dynamického přiřazování různým klientům dle potřeby. Přestože přidělování zdrojů může být nezávislé na jejich geografické poloze, klient by měl být schopen jejich polohu zjistit alespoň na úrovni států.
- *Rapidní pružnost*: Zdroje musí být přídělovány pružně tak, aby bylo možné je (v některých případech automaticky) navyšovat či snižovat dle aktuálních požadavků.
- *Měřitelné služby*: Systém automaticky kontroluje a optimalizuje využívání zdrojů pomocí schopnosti měření metrik adekvátních pro daný typ služby (počet aktivních uživatelů, výpočetní výkon, úložiště). Měření využití zdrojů by mělo být transparentní pro poskytovatele služby i klienta.

Je definováno několik modelů cloudových služeb:

- *Infrastructure as a Service (IaaS)*: *IaaS* je nejnižší úroveň poskytování cloudových služeb, poskytovatel nabízí předkonfigurované hardwarové zdroje (výpočetní čas, úložiště, síťový přístup) pomocí virtuálního rozhraní. V modelu *IaaS* nejsou poskytovány žádné aplikace ani operační systém, poskytovatel pouze umožňuje přístup na hardwarové zdroje, kde (libovolný) software provozuje sám uživatel. Uživatel nemá kontrolu nad konfigurací hardwaru, má plná kontrolu nad konfigurací softwaru a v některých případech částečnou kontrolu nad konfigurací síťových zařízení (například *firewallů*). Příklady *IaaS* služeb jsou *Amazon EC2*[7], *IBM SoftLayer*[8] a *Google Compute Engine*[9].
- *Platform as a Service (PaaS)*: V modelu *PaaS* dostává uživatel softwarovou platformu, kterou používá pro spouštění svých aplikací vytvořených v programovacím jazyce s pomocí knihoven, služeb a nástrojů podporovaných poskytovatelem. Uživatel nemá přístup ke konfiguracím hardwaru, síťových prvků, ani k nižším softwarovým vrstvám, jako je např. operační systém, má ale přístup ke konfiguraci aplikace a k některém konfiguracím platformy (aplikačního prostředí). Příklady *PaaS* služeb jsou *Heroku*[10], *Amazon AWS Elastic Beanstalk*[11], *Google App Engine*[12] a *IBM BlueMix*[13].
- *Software as a Service (SaaS)*: V modelu *SaaS* uživatel využívá aplikaci uživatele běžící na cloudové infrastruktuře. Uživatel může k aplikaci přistupovat pomocí tenkého/tlustého klienta, nebo pomocí API a nemá přístup k žádným hardwarovým ani softwarovým konfiguracím.

[7] <aws.amazon.com/ec2>
[8] <www.ibm.com/Cloud/IaaS>
[9] <https://cloud.google.com/compute>
[10] <https://www.heroku.com/>
[11] <https://aws.amazon.com/elasticbeanstalk/>
[12] <https://cloud.google.com/appengine/>
[13] <https://console.bluemix.net/catalog/>

Protože tyto modely cloudových služeb přinášejí nové požadavky pro architektury aplikací, mění se i metodologie pro návrh a tvorbu aplikací. *The Twelf-Factor App* [78] popisuje dvanáct základních principů, které by měla dobře navržená cloudová aplikace dodržovat.

Vedle výše uvedených modelů cloudových služeb byl definován ještě model *Function as a Service* [59], který se v mnoha vlastnostech shoduje s modelem *PaaS*, přináší ale několik rozdílů ovlivňujících architekturu aplikací i samotné platformy. Zatímco model *PaaS* předpokládá rozdělení aplikací na poměrně velké celky (moduly, v extrémním případě může být aplikace monolitická), *FaaS* naopak předpokládá spouštění pouze malých kusů kódu - v extrémním případě jednotlivých funkcí. Ty jsou spouštěny v bezestavových kontejnerech, jejichž spouštění je vyvoláváno událostmi (životnost kontejneru je omezena na jeden požadavek). To mění požadavky na infrastrukturu poskytované platformy - zásadní je čas, za který je platforma schopná požadovanou *funkci* spustit. První dostupnou platformou pro tento model byla *Amazon AWS Lambda* [3]. Další platformy na trhu jsou například *Google Cloud Functions*[14], *Microsoft Azure Functions*[15], *IBM Cloud Functions*[16] a *Iron*[17].

Pro model *FaaS* vznikly nové softwarové architektury, které těží z jeho vlastností a nabízí vysoce dynamickou škálovatelnost aplikací - *Microservices* [1] a *Serverless architecture* [59]. Přestože mají tyto architektury mnoho zjevných výhod (popsaných v poskytnutých zdrojích), přináší také důležitý konceptuální problém, který byl tradičnějšími architekturami již překonán - *vendor locking*, tedy závislost na vybraném poskytovateli platformy. Tento problém se snaží řešit některé vznikající open-source frameworky jako jsou *Serverless*[18] a *Lambada Framework*[19], zatím je ale brzy na jejich hodnocení.

[14] <https://cloud.google.com/functions/>
[15] <https://azure.microsoft.com/en-us/services/functions/>
[16] <https://www.ibm.com/cloud/functions>
[17] <https://www.iron.io/>
[18] <https://serverless.com/>
[19] <https://github.com/lambadaframework/lambadaframework>

Kapitola 4

Analýza

4.1 Manta Flow

Manta Flow[1] je nástroj umožňující automatickou analýzu zdrojového kódu (*SQL*, Java) a následný popis transformační logiky v něm obsažené. Software je schopný rozpoznat i těžce čitelné konstruky zdrojového kódu. Díky tomu dokáže automaticky zanalyzovat rozsáhlé databáze a vytvořit z nich přehlednou mapu datových toků, neboli *Data Lineage*. To se v praxi využívá převážně k optimalizaci datových skladů, snižování nákladů na vývoj softwaru, provádění dopadových analýz a při dokumentování prostředí pro potřeby regulačních úřadů.

V souladu s architektonickým stylem *klient - server* má aplikace dvě hlavní komponenty (viz diagram B.1):

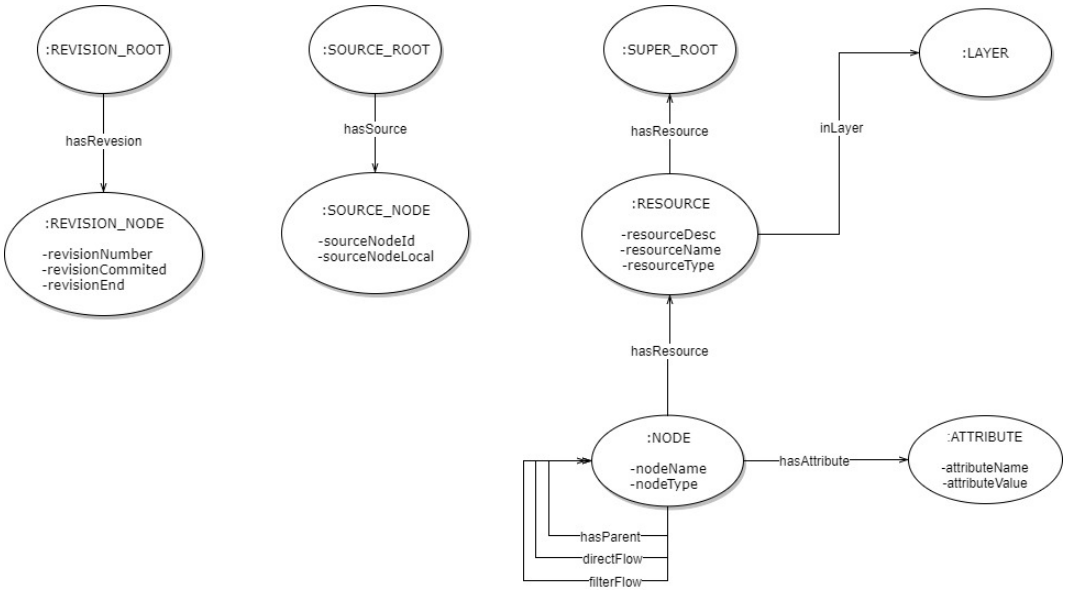
- *Manta Flow CLI*: je Java řádková aplikace provádějící extrakci skriptů ze zdrojových databází a úložišť a jejich analýzu. Analyzovaná data jsou následně poslána *Manta Flow Serveru*. *Klientská aplikace* také může nahrávat vygenerované exporty ze *serveru* do externí metadatové databáze.
- *Manta Flow Server*: je serverová Java aplikace, která ukládá získané informace do interního metadatového úložiště, transformuje je, umožňuje jejich visualizaci a přístup k nim pomocí veřejného *API*.

Interakce mezi *klientskou* a *serverovou* částí aplikace je popsána zjednodušeným sekvenčním diagramem B.2. Tento proces je označován jako *merge* a je blíže popsán v kapitole 4.3.2. Pro tuto práci je podstatná především *serverová* část aplikace (kapitola 4.3) a její interakce s metadatovým úložištěm (kapitola 4.2). *Klientská* část aplikace je proto popsána méně detailně (kapitola 4.4).

4.2 Metadatové úložiště

Jak již bylo zmíněno v úvodu práce (kapitola 1) metadatové úložiště produktu *Manta Flow* je aktuálně implementováno grafovou databází *Titan* (ve verzi 0.4) a je snaha o výměnu

[1] https://getmanta.com/



Obrázek 4.1: Model grafové databáze

této databáze[41]. Než přistoupíme k bližšímu popisu jednotlivých komponent aplikace a jejich interakcí s metadatovým úložištěm (kapitola 4.3, je třeba nejdříve popsat entity, které jsou součástí analýzy datových toků a datový model metadatového úložiště (zobrazený na obrázku 4.1[2]).

V procesu analýzy datových toků hraje roli mnoho entit z analyzovaných systémů. Ty se navíc mohou výrazně lišit systém od systému - *Manta Flow* může analyzovat širokou škálu spolu propojených databázových systémů a integračních služeb. Obecně lze říci, že každý systém obsahuje zdroje a cíle dat (tabulky, soubory, ...) a transformace dat (skripty, *Extract Load Transform (ETL)* workflow, procedury, makra a další).

Samotný datový model se skládá z devíti typů uzlů:

- *SUPER_ROOT*: Uzel (právě jeden v databázi), který slouží jako umělý kořen všech uzlů typu *RESOURCE*.
- *RESOURCE*: Uzly tohoto typu reprezentují zdrojové systémy - zdroje definic objektů, zdrojových kódů, *ETL* řešení a další.
- *NODE*: Uzly typu *NODE* představují reálné objekty zdrojového systému - databáze, tabulky, sloupce, procedury, skripty a další.
- *LAYER*: Uzly typu *LAYER* reprezentují vrstvy modelu metadat. Datové toky nalezené při analýze zdrojových kódů jsou vždy ukládány do *fyzické vrstvy*, ze které je potom možné generovat abstraktnější vrstvy modelu datových toků.

[2]Z modelu metadatového úložiště je zřejmé, že ne všechny podgrafy tvoří *strom*. Vlastnosti *stromu* nicméně porušují pouze hrany typu *directFlow* a *filterFlow*, které jsou výsledkem analýzy datových toků a jejichž odstraněním by strom vznikl. V textu je tak v některých případech používána teminologie vztahující se ke *stromům* (například *kořen*) - na celý graf je nahlíženo jako ne *les*.

- *ATTRIBUTE*: Uzly typu *ATTRIBUTE* reprezentují atributy uzlů typu *NODE* - parametry sloupců, popisy databázových objektů a další.
- *SOURCE_ROOT*: Uzel (právě jeden v databázi), který slouží jako umělý kořen všech uzlů typu *SOURCE_NODE*.
- *SOURCE_NODE*: Uzly typu *SOURCE_NODE* reprezentují soubory se zdrojovými kódy extrahovanými ze zdrojových systémů.
- *REVISION_ROOT*: Uzel (právě jeden v databázi), který slouží jako umělý kořen všech uzlů typu *REVISION_NODE*.
- *REVISION_NODE*: Uzly typu *ATTRIBUTE* reprezentují revize modelu metadat, definují tedy jeho verzování. Kromě dalších parametrů mají všechny hrany grafu parametry *tranEnd* a *tranStart* definující platnost hran (viz obrázek 4.2). Při každé analýze zdrojových systémů (která je prováděna dávkově klientskou částí aplikace) je vytvořena nová revize metadatového úložiště obsahující všechny objekty zdrojových systémů.[3]

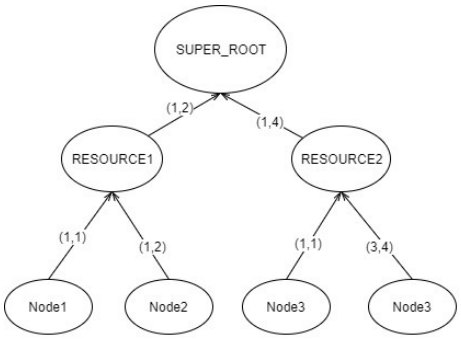
a osmi typů hran:

- *hasResource*: Hrana přiřazuje objekty (uzly typu *NODE*) ke svým zdrojovým systémům (uzlům typu *RESOURCE*). Hrana je také použite k propojení uzlů typu *RESOURCE* s uzlem *RESOURCE_ROOT*.
- *hasParent*: Hrana mezi dvěmi uzly typu *NODE* vytvářející klasickou hiearchickou strukturu mezi těmito uzly - strom závislostí objektů zdrojových systémů.
- *directFlow*: Hrana mezi dvěmi uzly typu *NODE* říkající, že mezi těmito uzly existuje přímý datový tok (ve směru hrany).
- *filterFlow*: Hrana mezi dvěmi uzly typu *NODE* říkající, že mezi těmito uzly existuje nepřímý datový tok (ve směru hrany).
- *hasAttribute*: Hrana přiřazující uzlům typu *NODE* jejich atributy (uzly typu *ATTRIBUTE*).
- *inLayer*: Hrana typu *inLayer* spojeju zdroje (uzly typu *RESOURCE*) a vrstvy a říká, že zdroj patří do dané vrstvy modelu metadat.
- *hasSource*: Hrana je použita k propojení uzlů reprezentujících zdrojové kódy (uzly typu *SOURCE_NODE*) s uzlem *SOURCE_ROOT*.
- *hasRevision*: Hrana je použita k propojení uzlů reprezentujících revize modelu metadat (uzly typu *REVISION_NODE*) s uzlem *REVISION_ROOT*.

Uzly i hrany mají dle svého typu několik specifických atributů, ty ale nebudeme blíže popisovat, protože nejsou pro analýzu zásadní.

V metadatovém úložišti je dále definováno několik typů indexů, konkrétně se jedná o standardní indexy:

[3] Je snaha tento princip upravit tak, aby byly objekty v metadatovém úložišti minimálně repklikovány [65].



Obrázek 4.2: Způsob verzování modelu metadat

- *indexy na kořeny*: indexy pro konkrétní uzly, kořeny jednotlivých stromů datového modelu - *SUPER_ROOT, SOURCE_ROOT, REVISION_ROOT*
- *indexy na atributy hran*: indexy zrychlující dohledávání atributů hran
- *indexy na typy hran*: indexy zrychlující dohledávání hran daného typu pro jednotlivé uzly

a externí *Apache Lucene* indexy, které slouží pro *fulltextové* vyhledávání uzlů dle jejich názvů a pro intervalové vyhledání revizí.

4.3 Popis komponent serverové části

Aplikace *Manta Flow* praceje s metadatovým úložištěm několika různými způsoby, přičemž různé moduly aplikace využívají jeden či více těchto přístupů (a často také provádí vlastní pomocné dotazy přímo do metadatového úložiště). Cílem této sekce je tyto způsoby manipulace s metadatovým úložištěm identifikovat a popsat (není tedy účelem detailní technický popis všech dotazů do metadatového úložiště, ale spíše popis obecných principů a specifických situací - například netradiční zacházení s transakcemi).

4.3.1 Connector

Modul, který je nejblíže metadatovému úložišti, tzv. **connector** má dvě hlavní zodpovědnosti - zajištění připojení aplikace k úložišti a provádění dotazů nad ním.

Modul obsahuje sadu základních dotazů, tzv. *operatinos*, mezi které patří například:

- získání předka uzlu
- získání atributů uzlu
- získání sousedních uzlů a hran
- získání cesty ke kořeni

- získání podstromu

Tyto operace přímo přistupují do databáze a pomocí programovacího jazyka *Gremlin*[4] a jsou na nich postaveny složitější operace nad grafouvou databází. U základních operací nejsou transakce řízeny explicitně, ale implicitně grafovou databází.

Další částí modulu *connector* jsou tzv. algoritmy, tedy komponenta, pomocí které jsou v metadatovém úložišti hledány samotné datové toky. Tato komponenta řetězí několik grafových algoritmů, přičemž první z nich získá z metadatového úložiště podmnožinu datových toků, která je dalšími algoritmy filtrována a omezována. Tímto způsobem vzniká tzv. *referenční view* - objekt obsahující kompletní graf datových toků pro zadané výchozí uzly a směr datových toků. To je pak využ íváno dalšími moduly aplikace, například *viewerem* (viz 4.3.3), který dle parametrů zadaných ve webové aplikaci graf datových toků vizualizuje. Jednotlivé algoritmy používají výše popsané základní operace (*operations*) a v některých případech také samy dotazují metadatové úložiště přímo pomocí jazyka *Gremlin*.

Posledním způsobem manipulace s metadatovým úložištěm, který modul umožňuje je přístup pomocí *traversalů*. Ty pracují na obecném principu *traversování* grafů popsaném v kapitole 2.4. V tomto případě ale celý průchod grafem není realizován samotnou grafovou databází, ale přímo aplikací, přičemž grafová databáze je dotazována pouze na dílčí informace - například na okolní uzly. *Traversaly* jsou používány v případech, kdy je manipulováno s větší částí grafové databáze, například při jejím exportu. Tyto operace jsou realizovány *vizitory* - každý uzel, který je procházen *traversalem* je následně obsloužen *vizitorem*, který provede požadovanou operaci (jedná se o návrhový vzor *Visitor* - viz [28]). Obě tyto části, tedy procházení grafu *traversalem* i obsloužení všech objektů *visitorem* jsou prováděny za pomocí základních grafových operací definovaných výše (*operations*). Zároveň je ale také z tohoto kontextu grafová databáze dotazována přímo pomocí jazyka *Gremlin*. Jedná se ale spíše o jednoduché dotazy na dohledání uzlů, jejich atributů apod.

4.3.2 Merger

Merger je modul, který je používán při analýze zdrojových systémů. Slouží k zanesení výsledků dílčích analýz jednotlivých částí (např. skriptů) zdrojových systémů do metadatového úložiště.

Vlastní operace *merge*[5] lze zjednodušeně popsat pseudokódem 1 (správa transakcí a synchronizace je blíže popsána v kapitole 4.5.1). Ten je uveden především kvůli složitému transakčnímu modelu, jehož účelem je umožňení provádění dotazů do metadatového úložiště jinými částmi aplikace, zatímco je prováděn *merge* analyzovaných částí zdrojových systémů.

Operace se chová různě v případě, kdy je umožňěno verzovaní metadatového úložiště (a to tak obsahuje více revízí) a kdy je vypnuto. V případě zapnutého verzování je *merge* prováděn vždy do nové revize, pokud je verzování vypnuté, je prováděn do hlavní (jediné) revize.

Samotné *merge* operace nad objekty grafové databáze (uzly, hranami, atributy, ...) jsou prováděny přímímy dotazy do databáze pomocí jazyka *Gremlin*. K přístupu do metadadtového úložiště se tedy nevyužívá modul k tomu předurčený - *Connector* (viz 4.3.1).

[4] Používá se *TinkerPop* ve verzi *2.6*.

[5] Operace *merge* má v kontextu aplikace *Manta Flow* obdobný význam, jako například v *SQL*: pokud objekt není uložen v persistentní vrstvě, je do ní uložen (*insert*), jinak je aktualizován (*update*).

Algorithm 1 Merger pseudocode

```
acquireGraphLock()
revision ← getNewestRevision()
if revision.isOpen() then
    acquireScriptsLock()
    for all scriptinscripts do
        beginWriteTransaction()
        merge(script)
        conditionalCommit()
    end for
    releaseScriptsLock()
    beginWriteTransaction()
    for all objectinobjects do
        merge(object)
        periodicalCommit()
    end for
end if
releaseGraphLock()
```

4.3.3 Viewer

Viewer je modul sloužící k poskytování dat uživatelskému rozhraní aplikace (klientské části webové aplikace). Jeho nejčastější interakce s metadatovým úložištěm je dotaz na *referenční view* dle parametrů zadaných uživatelem. To je prováděno pomocí algoritmů definovaných v modulu *connector* (viz 4.3.1), který obsahuje algoritmy pro hledání datových toků (resp. *referenčního view*).

Kromě toho *viewer* dotazuje metadatové úložiště o další informace, které následně propaguje do uživatelského rozhraní - především o informace o revizích metadatového úložiště a o objekty zdrojových systémů, pomocí kterých uživatel vybírá výchozí uzly pro hledání datových toků (*referenčního view*). Informace o revizích metadatového úložiště jsou dohledávány pomocí základních operací definovaných v modulu *connector* (viz 4.3.1) a pomocí přímých dotazů do metadatového úložiště pomocí jazyka *Gremlin* s explicitního řízení transakcí. Pro vyhledávání objektů zdrojových systémů (v metadatovém úložišti uzly typu *NODE*, viz 4.2) je použito vyhledávání pomocí fulltextového indexu implementovaného pomocí *Apache Lucene*. Ten indexuje uzly v metadatovém úložišti podle jejich názvu a umožňuje jejich rychlé vyhledávání.

4.3.4 Public API

Public API je modul, který by měl umožňovat vzdálené volání veřejné části funkcionality aplikace pomocí *REST API*. Konkrétně lze tímto způsobem volat například analýzu datových toků mezi různými objekty zdrojového systému. Modul přepoužívá část funkcionality poskytovanou modulem *connector*, část těchto funkcionalit ale duplikuje (s menšími úpravami) a přímo tak dotazuje metadatové úložiště.

4.3.5 Exporter

Posledním modulem, který přímo interaguje s metadatovým úložištěm je *exporter*, jehož úkolem je exportovat aktuální stav grafové databáze (ne nutně vše, může být exportován například jen interval revizí atd.) buďto do *CSV* souborů, nebo přímo do formátu používaného dalšími nástroji používanými na správu metadat. *Exporter* jako nástroj pro práci s metadatovým úložištěm nejčastěji používá *traversery* a *observery* poskytované modulem *connector*, díky kterým je možné provádět operace nad velkou částí metadatového úložiště bez zásadních paměťových požadavků. Dále jsou využívány základní základní operace (*operations*) poskytované stejným modulem a v některých případech je metadatové úložiště dotazováno přímo pomocí jazyka *Gremlin*.

4.4 Popis ostatních komponent

4.4.1 Manta Flow Client

Klientská část aplikace *Manta Flow* je *command line* aplikace implementovaná v programovacím jazyce Java sestavená z několika modulů. Jejím hlavním úkolem je extrakce skriptů ze zdrojových databázových systémů a repozitářů (modul *Exktractor*), a jejich následné parsování a analýza (modul *Analyzer*). Analýza skriptů probíhá v klientské části aplikace z toho důvodu, že využívá vyextrahované slovníky objektů zdrojového systému, které by v případě provádění analýzy serverovou částí aplikace musely být přenášena na server spolu se skripty. Poté, co proběhne zpracování (*merge* - viz. kapitola 4.3.2) zanalyzovaných skriptů serverem, výsledky jsou vyexportovány zpět do klientské části aplikace, odkud mohou být případně nahrány do externí metadatové databáze.

4.4.2 Configurator

Configurator je *standalone* webová aplikace implementovaná v programovacím jazyce Java (jako třívrstvá aplikace), jejímž úkolem je poskytnout *grafické uživatelské rozhraní (GUI)*, pomocí kterého může uživatel změnit komplexní konfiguraci aplikace *Manta Flow*. Konfigurace je typicky obsažena v *properties* souborech a to na různých místech. Serverová a klientská část *Manta Flow* má vlastní konfiguraci. [49]

4.4.3 Updater

Updater je *standalone* webová aplikace implementovaná v programovacím jazyce Java (jako třívrstvá aplikace), jejímž úkolem je poskytnout *GUI*, které provede uživatelem *updatem* aplikace *Manta Flow* (konkrétně její serverové části) na novější verzi. Aplikace umožňuje uživateli provést změny v komplexní konfiguraci aplikace a provede *merge* těchto změn s původním nastavením. [34]

4.5 Omezení stávající architektury aplikace

V následujících podkapitolách jsou popsány aktuální architektonická omezení a problémy aplikace *Manta Flow*. Cílem nově navržené architektury je, aby řešila problémy pramenící

z nevhodné manipulace s grafovou databází představující metadatové úložiště a umožnila řešení ostatních problémů popsaných v této kapitole.

4.5.1 Datová konzistence

Z popisu jednotlivých komponent aplikace v kapitolách 4.3 a 4.4 je patrné, že je používáno několik heterogenních přístupů k manipulaci s daty (především v grafové databázi) a je tak třeba pečlivě řídit konzistenci dat. Primárním principem zaručujícím konzistenci dat je verzování modelů datových toků na jednotlivé *major* a *minor* revize [65]. Cílem je, aby všechny moduly, které pouze provádějí analýzu datových toků a tu předávají (v některé z dostupných forem) dále, tedy *Viewer*, *Exporter* a *Public API*[6] mohly být spouštěny nad některou z uzavřených (*commitnutých*) revizí, zatímco *Merger*, který vkládá nové informace na základě analýzy provedené klientskou částí aplikace, má k dispozici jednu otevřenou *necommitnutou* pracovní revizi. Všechny zmíněné moduly používají pro přístup do grafové databáze primárně modul *Connector*, který obsahuje operace čtení i zápisu, obecně ale platí, že operace zápisu jsou používány pouze *Mergerem*. I přes systém revizí může vzniknout často konkurence při čtení a zápisu dat do grafové databáze:

- *Úprava uzavřené revize*: Při vytváření nové revize, která vzniká typicky jako *full update* celého modelu datových toků[7], může nastat několik situací. Připomeňme, že platnost vztahu mezi dvěmi uzly je parametrem hrany spojující tyto uzly. Pokud se libovolné dva uzly a jejich vztah nezmění mezi uzavřenou revizí A a vznikající neuzavřenou revizí B, pak jsou upraveny atributy hrany reprezentující tento vztah a může tak vzniknout konkurence při čtení revize A a zápisu do revize B.

- *Kombinace operací pro čtení a zápis při vytváření nové revize*: Modul *Merger* při vytváření nové revize modelu datových toků v grafové databází spouští několik komplexních postproccessingových algoritmů, které vyžadují přístup pro čtení i zápis do pracovní otevřené revize (a jsou v některých případech prováděny paralelně).

- *"Stop the world"operace*: Je také definováno několik takzvaně "stop the world"operací, tedy operací, při kterých je znepřístupněna velká část, nebo přímo celá grafová databáze. Mezi tyto operace patří odstraňování starých revizí modelu datových toků, import či export kompletního *dumpu* grafové databáze.

Vzhledem k výše uvedeným situacím je konkurence při přístupu k datům řízena dvěmi dalšími mechanikami, kterými jsou databázové transakce a synchronizace pomocí *reentrant* zámků.

[6] Pomocí *Public API* může být grafová databáze i upravována, všechny úpravy jsou ale prováděny pouze pomocí modulu *Merger*.

[7] Nová *major* revize vzniká jako *full update* modelu datavých toků, nová *minor* revize jako *incremental update*. Konkrétní implementace obou operací se v některých detailech liší, z pohledu datové konzistence ale řeší oba přístupy koncepčně stejný problém. Můžeme proto analýzu vzniku nové revize datového modelu zobecnit na *full update* přístup - tedy na *major* revize.

Transakce jsou řízeny podle transakčního modelu *Programatic Transaction Model (PTM)*[8] [47] za pomoci implementace *Spring TransactionTemplate*[9] kombinující *Titan* transakce a Java *reentrant* zámky. *Titan* v případě transakcí pouze přeposkytuje funkcionalitu používané podkladové databáze, kterou je v případě *Manta Flow* embedded *NoSQL* databáze typu klíč-hodnota *Persistit*[10]. Ta, stejně jako mnoho dalších grafových databází, implementuje *Multi-Version Concurrency Control (MVCC)* [46] využívající *Snapshot Isolation*[11] úroveň izolace transakcí. Jedná se o *optimistický* transakční model, žádné databázové objekty (ani záznamy) nejsou zamykány, takže transakce probíhají v plné rychlosti bez čekání. Může tak ale nastat situace, že ne všechny transakce je možné *commitnout* a jedna (nebo více) z těchto transakcí tak musí být zrušena (a zopakována). *Snapshot* izolace transakcí má v grafových databázích také další specifický důsledek. Zatímco v případě relačních databází vždy existuje unikátní identifikátor každého záznamu (byť i implicitní), v grafových databázích tomu tak není - tedy neexistuje žádná unikátnost uzlů a hran v grafové databázi za předpokladu, že není explicitně vynucena pomocí unikátních indexů. Ty nejsou v aplikaci *Manta Flow* používány, může tak nastat situace, kdy se dvě transakce souběžně snaží o vytvoření (z pohledu aplikační logiky) identického uzlu či hrany, obě transakce uspějí (nedojde k jejich kolizi) a objekt je tak vytvořen dvakrát (čímž je zanesena nekonzistence do metadatového úložiště). Aby k těmto konfliktům paralelních transakcí nedocházelo, používá aplikace při přidělování transakcí synchronizaci pomocí *reentrant* zámků, přičemž vlastní zámek je pro obecné transakce a vlastní pro *read-only* transakce. Nemohou tak současně existovat dvě transakce, které by mohly do databáze zapisovat zároveň - což může potenciálně velmi snižovat rychlost prováděných operací. Důsledkem tohoto systému zámků je serializace paralelních klientských požadavků pro zápis do metadatového úložiště (*merge*). Tento process v klientské části aplikace přitom probíhá paralelně, tipicky minimálně ve čtyřech vláknech.

Další používanou synchronizační mechanikou je zamykání grafové databáze (respektive uzlů reprezentující jednotlivé revize) *reentrant read-write* zámkem při několika operacích, které jsou součástí *Mergeru* a importu a exportu *dumpu* grafové databáze. Tyto zámky tak společně s verzováním modelu metadat slouží při *mergi* de-facto jako implementace *long living* transakcí - samotný *merge* objektů do grafové databáze je prováděn v několika transakcích[12].

Poslední synchronizace na straně serveru je zamykání metadatové databáze nesoucí informace a zanalyzovaných skriptech pro účely kontroly licencí. K této synchronizaci dochází při *mergi* v okamžiku, kdy jsou informace o skriptech do metadatového úložiště nahrávány.

[8] Kromě *PTM* je možné použít i deklarativní řízení transakcí (*Declarative Transaction Model (DTM)*). Ne pro všechny grafové databáze ale v tuto chvíli existují implementace, které by deklarativní transakce v rámci *Spring* frameworku umožňovaly. V tuto chvíli je podporuje například *Orient DB* (<https://github.com/orientechnologies/spring-data-orientdb>) a databáze přístupné pomocí *TinkerPop Gremlin 2.x* (<https://github.com/gjrwebber/spring-data-gremlin>). Podpora pro *Apache TinkerPop 3.x* je aktuálně ve vývoji.

[9] <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/support/TransactionTemplate.html>

[10] <https://github.com/pbeaman/persistit>

[11] Všechny záznamy v databázi jsou verzovány. Při čtení transakce vytvoří kopii poslední *commitnuté* verze čteného záznamu, ta je po ukončení transakce odstraněna. *Snapshot isolation* tak v zásadě odpovídá úrovni izolace transakcí *Read committed*.

[12] Transakce jsou objektem uloženým v operační paměti, přičemž velikost transakce narůstá s počtem úprav v databázi.

4.5.2 Výkon aplikace

Výkon je významným kvalitativním kritériem aplikace. U různých procesů jsou přitom požadavky na výkon různé:

- *Analýza datových toků*: Analýza datových toků probíhá v reálném čase na základě požadavku uživatele (pomocí *GUI* aplikace poskytované modulem *Viewer* - kapitola 4.3.3), nebo pomocí *REST API* - kapitola 4.3.4). Maximální *response time* požadavku je tedy v řádu jednotek vteřin. Toho je docíleno především optimalizací algoritmů počítajících datové toky na základě uživatelského požadavku a optimalizací dotazování metadatového úložiště pomocí interních a externích indexů (kapitola 4.2).

- *Update metadatového úložiště*: Narozdíl od výpočtu datových toků je update metadatového úložiště dávkový (typicky noční) proces, který probíhá v závislosti na procesech uživatelů *Manta Flow* jednou denně až jednou měsíčně[13]. Základním výkonnovým požadavkem pro tento proces tak je, aby byl dokončitelný řádu malých jednotek hodin a to (teoreticky) nezávisle na velikosti vstupů a aktuální velikosti metadatového úložiště. V současné době se update metadatového úložiště na některých instancích vstupních dat blíží k hranici maximálního *response time*, je proto nutné, aby navržená architektura tento stav reflektovala a umožnila zrychlení procesu optimalizací, nebo škálováním algoritmu.

  V kapitole 4.5.1 bylo nastíněno, že update metadatového úložiště je operace vykonávána částečně paralelně a částečně sériově. Konkrétně (vzhledem k diagramu B.2) jsou vykonávány paralelně fáze *extrakce* a *analýzy* zdrojových kódů prováděné klientskou částí aplikace *Manta Flow*. Při provádění následné operace *merge* (serverovou částí aplikace) jsou ale paralelní požadavky serializovány, aby nedocházelo k paralelnímu zápisu do metadatového úložiště. Vzhledem k tomu, že *extrakce* a *analýza* zdrojových kódů probíhá typicky minimálně ve čtyřech paralelních vláknech, je paralelním zápisem do metadatového úložiště dosáhnout potenciálně čtyřnásobného zrychlení operace *merge*.

- *Export*: Další výpočetně náročnou operací, kterou aplikace pravidelně provádí je *export*. Ten je nutno chápat jako operaci obsahující velké množství byznys logiky. Nejedná se o prostou transformaci informací obsažených v metadatovém úložišti do jiného formátu, naopak, informace o datových tocích jsou upravovány tak, aby byla zachována jejich sémantika i po jejich integraci s navazujícím nástrojem pro správu metadat. *Export* tak zahrnuje (často opakovaný) průchod podstatné části grafu datových toků a v některých případech také specifické analýzy datových toků. Stejně jako *update* metadatového úložiště je i *export* dávkovým procesem. Požadavky na výkon těchto dvou procesů jsou tak podobné, s tou výjimkou, že export do metadatového úložiště nezapisuje a není tak nutné řídit konzistenci dat[14].

[13] Předpokládá se, že v návaznosti na zavedení nové funkce inkrementálního updatu [65] bude prováděn update metadatového úložiště výrazně častěji, než je tomu v tuto chvíli. Kvůli povaze inkrementálního updatu je ale možné předpokládat, že tato operace bude výpočetně výrazně méně náročná, než úplný update, a bude tak na pomezí mezi dávkovým procesem a *online* procesem. Výkon operace je tak důležité sledovat a optimalizovat především v případě úplného updatu metadatového úložiště.

[14] *Export* je prováděn vždy pouze nad uzavřenou (*commitnutou*) revizí, je tedy možné se spoléhat na to, že procházené uzly a hrany se minimálně v této revizi již nezmění.

Důležitým důsledkem optimalizace výkonu je používání *embedded* databáze *Persistit*, jako podkladového úložiště pro grafovou databázi *Titan*. Bylo provedeno výkonostní testování různých grafových databází (v různých konfiguracích), v žádném případě se ale nepodařilo se výkonem přiblížit k aktuálně používané konfiguraci [41]. To je způsobeno především tím, že grafové algoritmy, které jsou součástí jednotlivých procesů *Manta Flow* jsou implementovány jako součást byznys logiky aplikace. Výsledkem tedy je, že grafová databáze vykonává pouze jednoduché dotazy, které se často omezují pouze na prohledání sousedních hran a uzlů počátečního uzlu.

4.5.3 Škálovatelnost

Škálovatelnost je jedním z problémů, se kterými se může aplikace *Manta Flow* v blízké budoucnosti potýkat. V současné době je možné pouze omezené škálování aplikace - zatímco klientská část aplikace existuje typicky jedna pro každý datový zdroj instance aplikace (což umožňuje její horizontální školování), serverová část aplikace je *de-facto* monolit a umožňuje tedy pouze vertikální škálování. Vzhledem k tomu, že jako podkladové úložiště grafové databáze je používána *embedded* databáze *Persistit*, není možné využít ani možnosti horizontálního škálování grafové databáze *Titan*, kterou (stejně jako většina ostatních grafových databází) standardně nabízí.

Potenciální potřeba horizontálního škálování serverové části aplikace má dva hlavní argumenty:

- *Objem dat*: Je zjevným trendem, že objem zpracovávaných dat napříč doménami stoupá. V důsledku toho se mění a rozšiřují i struktury, které umožňují ukládání a/nebo analýzu těchto dat - vzrůstá tedy i objem metadat. Objem dat zpracovávaných aplikací *Manta Flow* tedy jistě bude průběžně narůstat. Při současné architektuře je aplikace (včetně *embedded* grafové databáze) instalována na infrastrukturu uživatele aplikace, přičemž největší z těchto instancí obsahují řádově desítky *gigabytů* dat. Nejvytíženější instance *Manta Flow* zpracovává pravidelně desítky tisíc *DDL* skriptů obsahujících přibližně 5 milionů databázových sloupců. Důsledkem zpracování takového zdroje je grafová databáze obsahující desítky milionů uzlů a hran. Udávaný limit pro používání grafové databáze *Titan* spolu s *embedded* databází *Persistit* je přitom 100 milionů uzlů [8]. Lze tedy předpokládat, že stávající architektura nebude schopna zpracovat stále se zvětšující objem vstupních dat.

- *Výkon*: V sekci 4.5.2 jsou popsány procesy v rámci aplikace, u kterých je kladen největší důraz na výkon. Také bylo uvedeno, že výkon některých z těchto procesů se na některých instancích vstupních dat již blíží prahové hodnotě, po jejímž řešení by byl již výkon nedostatečný. Tento problém může být řešen mimo jiné horizontálním škálováním, tedy navýšením počtu procesorů zpracovávajích vstupní data procesu. Například u operace *update* metadatového úložiště jsou ale aktuálně paralelní požadavky s částmi vstupu pro operaci serializovány, stávající architektura tedy horizontální škálování v tomto případě nepodporuje.

4.5.4 Viditelnost

Přestože je klientská i serverová část aplikace *Manta Flow* rozdělena na jednotlivé moduly, viditelnost je v některých částech aplikace nízká. Nejzásadněji k tomu přispívá způsob práce s metadatovým úložištěm. V kapitole 4.5.1 je popsána implementace správy transakcí - transakce jsou implementovány podle modelu *PTM*, jako objekty jsou propagovány z modulu *Connector* do (všech) ostatních modulů serverové části aplikace. Důsledkem tohoto návrhu je, že aplikace umožňuje přímý přístup do metadatového úložiště z kteréhokoliv místa, kam je zpropagován objekt transakce. Velká část implementace modulů obsahujících byznys logiku aplikace tak obchází sadu operací definovaných pro práci s metadatovým úložištěm (často je důvodem nedostatečné pokrytí potřeb pro práci s úložištěm těmito operacemi) a provádí vlastní, specifické dotazy do metadatového úložiště. Toto prolínání byznys logiky a perzistenční logiky potom kód jednotlivých modulů značně komplikuje a dělá jej nepřehledným, obtížně modifikovatelným a v některých případech znemožňuje vytvoření jednotkových testů funkcionality (nelze oddělit testování byznys logiky od testování perzistenční logiky).

4.5.5 Modifikovatelnost

Požadavky na modifikovatelnost aplikace *Manta Flow* lze rozdělit na několik stěžejních problémů:

- *Přizpůsobitelnost zdrojovým systémům*: *Manta Flow* podporuje analýzu datových toků značného množství technologií. Z hlediska různorodosti těchto technologií se jedná především o *RDBMS* databáze, datové sklady, *Big Data* technologie a *ETL* nástroje. Je tedy zřejmé, že extrakce a analýza zdrojových kódů řešení využívajících tyto technologie je napříč technologiemi velmi rozdílná. Tento problém je vyřešen modularitou klientské části aplikaci, která provádí především právě extrakci a analýzu zdrojových kódů. Pro každou podporovanou technologii tak lze sestavit vlastní klientskou aplikaci[15].

- *Přizpůsobitelnost integračním systémům*: Stejně jako umožňuje *Manta Flow* analýzu různých technologií, tak také podporuje integraci několika technologiemi zaměřenými na správu metadat. Tyto integrace jsou realizovány pomocí exportů datových toků z metadatového úložiště *Manta Flow* a následného importu těchto informací do zvoleného nástroje. Tento proces je prováděn *Manta Flow serverem*, konkrétně komponentou *Exporter* (kapitola 4.3.5). Protože export pro každý nástroj pro správu metadata, se kterým aplikace integruje má jiný formát, existuje podobně, jako v předchozím případě, pro každý nástroj vlastní exportní modul.

- *Přizpůsobitelnost metadatovému úložišti*: Vzhledem k rychlému vývoji a neexistenci jasně definovaných standardů ve světě grafových databází je také žádoucí, aby architektura aplikace umožňovala výměnu technologie používané jako metadatavé úložiště a to bez zásadnějších dopadů na zbytek aplikace. Taková výměna by v tuto chvíli byla velmi komplikovaná, k metadatovému úložiště je přistupováno z mnoha míst v

[15] Modularitu klientské části aplikace zajišťují především komponenty *Extractor* (provádí extrakci zrdojových kódů), *Parser* (parsuje zdrojové kódy) a *Analyzer* (analyzuje zdrojové kódy).

aplikaci a to často velmi různorodým způsobem (popsáno v 4.3). Při změně grafové databáze, nebo jazyka, pomocí kterého je grafová databáze dotazována, by tak bylo nutné přepsat velkou část serverové části aplikace a to včetně byznys logiky, která je v některých extrémních případech taktéž implementována pomocí přímého zápisu do grafové databáze.

- *Rozšiřitelnost*: Kvůli nadměrné složitosti části implementace některých modulů serverové části aplikace (důvody popsány v kapitole 4.5.4) je v některých případech obtížné aplikaci rozšiřovat o další funkcionality (či upravovat stávající). Například nový požadavek na rozšíření serverové části aplikace o funkcionalitu, která bude omezovat přístup k informacím uloženým v metadatovém úložišti na základě konfigurace oprávnění jednotlivých uživatelů aplikace, je při stávající architektuře a implementaci serverové části aplikace velmi komplikované implementovat tak, aby byly pokryty skutečně všechny adekvátní procesy přistupující do metadatového úložiště.

4.5.6 Orchestrace aplikací

V rámci této kapitoly byly popsány čtyři samostatné aplikace: *Manta Flow Server* (sekce 4.3), *Manta Flow Client* (sekce 4.4.1), *Updater* (sekce 4.4.3) a *Configurator* (sekce 4.4.2). Komunikace mezi *Manta Flow Server a Client* je popsána diagramem B.2. *Updater* a *Configurator* jsou obslužné aplikace provádějící úpravy *Serveru* a *Clienta* zahrnující úpravy, přidávání či odebírání konfiguračních i zdrojových souborů těchto aplikací. V některých případech mohou být upravovány i konfigurační soubory aplikačního serveru *Tomcat*[16], který je součástí *Manta Flow Serveru*. *Updater* a *Configurator* tak musí být spouštěny na jiném aplikačním serveru, než *Manta Flow Server*. Zároveň mohou obě aplikace aktuálně provádět změny pouze na stejném fyzickém zařízení, na kterém jsou nainstalovány - pokud jsou serverová a klientská část aplikace nainstalovány na různá zařízení, musí být aplikace *Updater* a *Configurator* instalovány dvakrát.

Tento způsob orchestrace aplikací (popsaný *UML* diagramem nasazení B.3) má zřejmé nevýhody.

Samotný fakt, že obě aplikace (*Updater* i *Configurator*) mají dvě instance na dvou různých zařízeních přináší novou režii při správě aplikací. Původní instalace nástroje *Manta Flow* zahrnovala celkem dvě a více instancí (server a jeden či více klientů). S přidáním nových instancí *Updateru* i *Configuratoru* by se jednalo o šest a více instancí. Snížená režie pro správu nástroje, kterou tyto aplikace znamenají, by tak byla částečně kompenzována režií těchto čtyř nových instancí. Aplikace *Updater* a *Configurator* také obsahují konfiguraci (byť jednoduchou) a mohou být verzovány (byť méně dynamicky, než základní aplikace). To je také důvod, proč by měl *Updater* být schopen provádět updaty aplikace *Configurator* i sebe sama.

Dalším problémem jsou možné nekonzistence, které mohou kvůli dvěma instancím obslužných aplikací vzniknout. Tento problém je zvlášť patrný u aplikace *Updater*. Verzovací model *Manta Flow* je nastaven tak, že verze klienta by měla vždy odpovídat verzi serveru - v opačném případě nemůže být zajištěna kompatibilita verzí. Dvě instance *Updateru* ale znamenají, že *update* každé instance serveru a klienta je samostatným procesem - může tak

[16] <https://tomcat.apache.org/>

snadno dojít k *updatu* na nekompatibilní verze instancí. U aplikace *Configurator* tento problém není tak zásadní, protože konfigurace serveru a klienta jsou disjunktní a navzájem se neovlivňují. Provádění konfigurace pomocí dvou instancí *Configuratoru* ale není příliš uživatelsky přívětivé.

Posledním problémem na této úrovni architektury aplikace je způsob dodávání klientské části aplikace. Ta podporuje mnoho různých technologií a každá instance má tak často vlastní *build* v závislosti na tom, s jakými technologiemi bude využívána. To znamená značnou režii při nasazování aplikace.

4.5.7 API grafových databází

Manta Flow pro dotazování grafové databáze (v tuto chvíli *Titan*) používá programovací jazyk *Gremlin* ve verzi *2.6*. Lze tedy říci, že tento programovací jazyk slouží jako API mezi aplikací a aktuálně používanou grafovou databází. Z benchmarků dalších grafových databází [41] vyplývá, že bude-li současná grafová databáze nahrazena, jejím nástupcem bude pravděpodobně *JanusGraph*, nebo *OrientDB*[17]. Obě tyto databáze podporují programovací jazyk *Gremlin*, *JanusGraph* ve verzi *3.x*, zatímco *OrientDB* ve verzi *2.x*. Protože *Gremlin* a API, která mají grafovým databázím zaručit jeho podporu, prošli mezi verzemi *2.x* a *3.x* zásadními změnami (viz 2.9), budou pro přístup do datové databáze v práci nadále uvažovány obě verze jazyka *Gremlin*, respektive *Java API*, které obě verze jazyka poskytují (*Gremlin 2.x [73], Gremlin 3.x [5]*).

4.6 Požadavky na návrh architektury aplikace

Na základě rozboru omezeních aplikace je vytvořen seznam funkčních (tabulka 4.1) a nefunkčních (tabulka 4.2) požadavků na architekturu aplikace. Součástí seznamu jsou všechny požadavky vyplývající z rozboru výše, tedy i požadavky, jejichž řešení není součástí zadání této práce. Kromě závažnosti požadavků (*kritická*, *důležitá*, *střední*) je tak u každého požadavku definován také jeho vztah k této práci - cíl (*NI* - návrh a prototypová implementace řešení, *N* návrh řešení, *P* - při řešení ostatních požadavků je daný požadavek brán v potaz).

Tabulka 4.1: Funkční požadavky na architekturu aplikace

ID	Popis	Závažnost	Cíl
F1	Navržená arhitektura umožní update všech komponent - tedy *Manta Flow Server, Client, Configurator, Updater*.	Důležitá	N
F2	Navržená architektura umožní uživatelsky přívětivý update všech komponent aplikace v rámci jednoho procesu	Střední	N
F3	Navržená architektura umožní uživatelsky přívětivou konfiguraci všech komponent aplikace z jednoho místa.	Střední	N

[17] Obě zmíněné databáze jsou blíže popsány v kapitole 2.8.

Tabulka 4.2: Nefunkční požadavky na architekturu aplikace

ID	Popis	Závažnost	Cíl
N1	Grafová databáze bude dotazována přímo pouze z jediného modulu aplikace, který bude zpřístupňovat svou funkcionalitu ostatním modulům a/nebo komponentám pomocí privátního *API*. Žádný jiný modul neobsahuje dotazy do grafové databáze.	Kritická	NI
N2	Pro dotazování grafové databáze bude použit jazyk *Gremlin* ve verzi *2.x*. Architektura ale umožní změnu tohoto jazyka na verzi *3.x*, nebo na jiné jazyky sloužící pro dotazování grafových databází.	Kritická	NI
N3	Navržená architektura povede k lepšímu zajištění konzistence dat v grafové databázi.	Důležitá	NI
N4	Navržená architektura umožní horizontální škálování aplikace.	Důležitá	N
N5	Navržená architekura umožní lepší rozšířitelnost serverové částí aplikace.	Důležitá	N
N6	Navržená architektura zvýší viditelnost interakcí komponent, které jsou nyní součástí serverové části aplikace.	Střední	N
N7	Navržená architektura zachová stávající úroveň, nebo zlepší úroveň přizpůsobitelnosti aplikace na nástroje třetích stran, které pro aplikaci představují zdrojové nebo integrační systémy.	Kritická	P
N8	Navržená architektura zachová stávající úroveň, nebo zvýší výkon aplikace. Tento požadavek se týká především z pohledu výkonu aplikace kritických operací definovaných v sekci 4.5.2.	Důležitá	P

4.7 Existující řešení pro abstrakci grafových databází

Existují nástroje, které si kladou za cíl poskytnout aplikacím využívajícím grafové databáze podobné nástroje na mapování mezi doménovými objekty a databázovými objekty, jako je dostupná pro databáze relační. U relačních databází je tento princip označován jako *ORM (objektově-relační mapování)* a pro programovací jazyk *Java* jsou používanými například *JPA (Java Persistence API)*[18], nebo nadstavba této technologie - *Hibernate*[19]. Alternativa pro grafové databáze je označována jako *OGM (objektově-grafové mapování)*. Níže jsou popsány nástroje, které tuto funkcionalitu (nebo její část) poskytují. Porovnání výkonu jednotlivých nástrojů je uvedeno především na základě testů [68].

[18] <http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html>
[19] <http://hibernate.org/>

- *Frames*[20] jsou nativní součástí projektu *TinkerPop* a poskytují *DSL (Domain Specific Language)* pro dotazovací jazyk *Gremlin 2.x*. Novější verzi jazyka (*3.x*) ale nepodporují a vykazují několikanásobně horší výkon v porovnání s nativními *Gremlin* dotazy.
- *Totorom*[21] je alternativou *TinkerPop Frames*. Steně jako *Frames* poudporuje pouze verzi *2.x* jazyka *Gremlin*, navíc má dle dostupných testů horší výkon, než *Frames*.
- *Peapod*[22] je dalším nástrojem poskytujícím *DSL* pro jazyk *Gremlin*, v tomto případě ale pouze pro verzi jazyka *3.x*. Není tedy použitelný pro práci s aktuálně používanou grafovou databází *Titan*.
- *Ferma*[23] je aktuálně jediným *OGM* nástrojem podporujícím jazyk *Gremlin* ve verzi *2.x i 3.x*. Verze nástroje *Ferma* ale bohužel nejsou zpětně kompatibilní, celý problém se rozdílností programovacího jazyka *Grelin* v různých verzích *Ferma* tak jen posouvá o krok dále - *Ferma 2.x* podporuje *Gremlin 2.x* a *Ferma 3.x* podporuje *Gremlin 3.x*. Srovnání výkonu nástroje *Ferma* s nativními *Gremlin* dotazy není tak špatné, jako je tomu u ostatních nástrojů, stále je ale o zhruba 25% horší.

Žádný z existujících nástrojů neřeší jeden z podstatných problémů aplikace *Manta Flow*, kterým je špatná modikovatelnost aplikace. Použitím těchto nástrojů by se neodstranila závislost implementace byznys logiky aplikace na konkrétním dotazovacím jazyce. Zároveň by použití kteréhokoliv z uvedených nástrojů znamenalo zhoršení výkonu dotazů do grafové databáze. Výkon je přitom jedním ze zásadních kvalitavních kritérií pro nově navrhovanou architekturu.

[20] <https://github.com/tinkerpop/frames/wiki>
[21] <https://github.com/BrynCooke/totorom>
[22] <http://bayofmany.github.io/>
[23] <http://syncleus.com/Ferma/>

Kapitola 5

Návrh architektury

Cílem této kapitoly práce je navrhnout a popsat architekturu, která bude řešit, či umožní budoucí řešení problémů popsaných v kapitole 4.5 a vyhoví tak požadavkům na architekturu aplikace definovaným v kapitole 4.6.

Tyto požadavky jsou rozděleny na tři skupiny, přičemž řešení požadavků mezi jednotlivými skupinami na jsou na sobě bůďto navzájem nezávislá, nebo je případná závislost explicitně uvedena v následujícím textu.

První jsou problémy způsobené mícháním perzistentní a byznys logiky v rámci serverové části aplikace aplikace, mezi které patří především špatná *modifikovatelnost* aplikace a nízká *viditelnost* interakcí jednotlivých komponent serverové částí aplikace. Také jsou diskutovány dopady těchto omezení na *výkon* aplikace. Konkrétně do této skupiny patří požadavky *N1, N2, N3, N5, N6* a *N8*. Tyto požadavky jsou realizovány především změnou architektury komponenty *Connector* (sekce 4.3.1) popsanou v sekci 5.1 této kapitoly.

Do druhé skupiny patří omezení vyplývající z aktuální architektury *Manta Flow* na úrovni orchestrace jednotlivých aplikací, které jsou součástí celého řešení. Nově vzniklé podpůrné aplikace *Configurator* (kapitola 4.4.2) a *Updater* (kapitola 4.4.3) zatím nejsou ukotveny v architektuře celého řešení. Do této skupiny patří požadavky *F1, F2, F3* a *N7*. Návrh řešení této skupiny požadavků je popsán sekcí 5.2 této kapitoly.

Třetí část této kapitoly je věnována požadavku *N4* a navrhuje možnosti horizontálního škálování aplikace - sekce 5.3.

5.1 Úprava architektury komponenty Connector

Jak bylo uvedeno v sekci 4.5, jedním z klíčových problémů, kterým v současné době aplikace *Manta Flow* čelí je splývání byznys logiky aplikace s perzistenční logikou, která implementuje ukládání datových toků do grafové databáze a jejich dotazování. To je nejvíce patrné v modulu *Connector*, který zajišťuje připojení aplikace k grafové databázi, vkládání a dotazování dat do/z grafové databáze (perzistenční logika) a poskytuje způsoby analýzy dat uložených v grafové databázi (algoritmy a traversaly - 4.3.1), obsahuje tedy také značné množství byznys logiky serverové části aplikace. První částí návrhu architektury je tak změna architektury *Manta Flow Serveru* tak, aby byl tento modul nahrazen vícevrstvou architekturou, která:

- umožní připojení aplikace do grafové databáze,
- definuje operace, které tvoří persistenční logiku aplikace, pomocí *API* a zamezí přímému přístupu do grafové databáze z jiných komponent,
- umožní přidělení/zamezení přístupu ke konkrétním informacím obsaženým v grafové databázi na základě definovaných oprávnění uživatelů a
- poskytne funkcionalitu pro pokrytí všech požadavků na přístup k datům ostatních modulů serverové části aplikace.

Hlavní kvalitativní kritéria zvolené architektury, která vycházejí z těchto požadavků, jsou:

- *Jednoduchost*: Zvolená architektura musí maximalizovat princip separace zájmů. Tím bude výrazně snížena závislost (*coupling*) byznys logiky aplikace a persistenční logiky a bude docíleno jednoduchosti komponent.
- *Viditelnost*: Komunikace mezi jednotlivými komponentami musí být přímočará a musí být rozšířitelná o novou komponentu. Aktuálně existuje požadavek na začlenění komponenty, která bude řídit přístup uživatele k informacím uloženým v metadatovém úložišti na základě oprávnění uživatele.
- *Modifikovatelnost*: Jednotlivé komponenty musí být jednoduše rozšířitelné. Musí být snadné přidávání komponent. Implementace (některých) komponent musí být zaměnitelná bez dopadů na další komponenty (jedná se především o implementaci komponenty implementující perzistenční logiku aplikace - kvůli možné výměně grafové databáze a tedy i dotazovacího jazky).
- *Výkon*: V kapitole 4.5.2 jsou popsány požadavky na výkon (především) serverové části aplikace *Manta Flow*. Architektura musí umožňovat splnění těchto požadavků a musí být definovány možné způsoby optimalizace výkonu.

5.1.1 Transakční model a řízení konzisence dat

Jak je popsáno v kapitole 4.5.1, aplikace *Manta Flow* má velmi specické požadavky týkající se řízení datové konzistence metadatového úložiště. Jedná se o problém, který je koncepční a může mít podstatné dopady na architekturu aplikace.

Bylo uvedeno, že stávající řešení používá *PTM* transakční model, transakce jsou mezi jednotlivými komponentami serverové části aplikace propagovány jako objekty a jsou používány k přímému přístupu do databáze. Cílený stav je takový, aby byl přímý přístup do databáze možný pouze z komponenty k tomu určené a ostatní komponenty (obsahující byznys logiku aplikace) pro přístup do databáze vždy používali tuto komponentu. Současně ale musí být umožněno propagování transakcí mimo tuto komponentu, transakce jsou často rozsáhlé[1]. Je tedy nutné zavést mechanismus **abstrakce transakcí**. V aplikacích používajích

[1] Úroveň izolací *snapshot isolation* vyplývající z *MVCC* implementace transakcí používané grafovými databázemi je specifická v tom, že je v průběhu transakce duplikováno velké množství databázových objektů. Ty jsou navíc často ukládány do paměti klientské služby, nikoliv databáze. Pro optimalizaci přístupů do

relační databáze je k tomuto účelu standardně používán **deklarativní transakční model**. Ten umožňuje konfigurovat chování každé metody, která přistupuje do datového zdroje (v tomto případě databáze), nebo která takovou metodu volá. U každé takové metody je definováno, jak má být transakce propagována, jaký je stupeň izolace transakce, zda je *read-only* a v jakém případě dochází k *rollbacku* transakce. Implementace tohoto modelu frameworkem *Spring* je popsána v dokumentaci [57]. V kontextu grafových databázích ale pro tento model zatím není u řady databází podpora a o standardní řešení se nejedná. Součástí návrhové fáze tak bylo vytvoření dvou *PoC* implementací (viz příloha C), které ověřují použitelnost tohoto řešení pro dotazovací jazyk *Gremlin* ve verzi *2.x* a *3.x*. V obou případech bylo ověřeno, že použití *deklarativního transakčního modelu* v kombinaci s frameworkem *Spring* je pro aplikaci *Manta Flow* vhodným řešením. Podpora pro jazyk *Gremlin* ve zmíněných verzích, respektive pro databáze, které ho podporují, byla v rámci těchto *PoC* implementována vlastní - existující implementace (uvedené v kapitole 4.5.1) jsou dostupné pouze v experimentálních verzích a obsahují chyby. Důležitým omezením, které vyplývá ze zvoleného řešení je, že všechny komponenty, do kterých jsou propagovány transakce musí být součástí monolitické architektury - není možná propagace transakcí pomocí standardních komunikačních protokolů (například *HTTP/s*). Navržená pravidla pro práci s deklarativními trasakcemi jsou:

- Všechny metody přistupující do databáze musí mít nakonfigurované transakční chování.
- Všechny metody zapisující do databáze musí mít nastaven způsob propagace na *Mandatory* - všechny metody, které tyto využívají tak musí mít také nakonfigurované transakční chování.
- Všechny metody, které používají transakční metody se způsobem propagace *Mandatory* musí mít konfigoravaný stejný způsob propagace, pokud nejsou součástí komponenty implementující byznys logiku aplikace. Tím je zaručeno, že rozsah transakcí provádějících změny v databázi je řízen právě v komponentách realizujících byznys logiku aplikace (zpravidla se bude jednat o komponentu *Merger*).

Dalším faktorem ovlivňující architekturu serverové části aplikace, který se týká datové konzistence, je systém explicitních zámků. Ten zajišťuje synchronizaci (serializaci) paralelních přístupů do grafové databáze - na úrovni celé detabáze může v jednu chvíli existovat pouze jedna zapisovací transakce (popsáno v kapitole 4.5.1). Tento systém je velkým omezením pro architekturu aplikace, protože zabraňuje jejímu (efektivnímu) škálování. Systém zámků je nutný - pokud by nebyl používán, docházelo by k zanášení nekonzistencí do metadatového úložiště a to především k duplikacím objektů[2]. Je tedy nutné tento systém upravit tak, aby lépe vyhovoval požadavků aplikace. Konkrétně byl navržen algoritmus pro zamykání objektů uložených v metadatovém úložišti definovaný následujícím chováním:

grafové databáze je tak podstatné zvolení správné velikosti transakcí. V případě minimalistických transakcí zahrnujících jednotky operací je režie transakcí příliš velká a práce s databází není efektivní. Při příliš rozsáhlých transakcích dochází k vyčerpání operační paměti klientské služby kvůli duplikování databázových objektů.

[2]Potenciální alternativou k systémů zámků by bylo zavedení unikátních identifikátorů uzlů a vytvoření indexů hlídajících tuto vlastnost. V tom případě by nedocházelo k duplikaci uzlů a používání zámků by nebylo nutné. Používání unikátních indexů v grafových databázích ale není obecně příliš efektivní. Uzly typu *NODE* navíc žádné přirozené unikátní identifikátory nemají, respektive jejich vytvoření by vedlo často na řetězce obsahující stovky znaků. Tento přístup byl tedy zavržen.

- zamykány jsou uzly v grafové databázi,
- existují dvě úrovně zámků - pro čtení a pro zápis,
- zámky pro čtení jsou uplatňovány pouze v případě, že je čteno z *necommitnuté* revize,
- při úpravě vlastností uzlu (zpravidla úprava intervalu platnosti uzlu) je zamykán tento uzel,
- při vytváření novéhu uzlu je zamykán uzel, který je předkem nového uzlu,
- při mazání uzlu je zamykán uzel, který je předkem mazaného uzlu,
- při přidávání hran, které nejsou součástí hiearchie grafu datových toků (hrany typu *DIRECT*, *FILTER* a *MAPS_ TO*), je zamykán výchozí uzel nové hrany,
- v případě situace vedoucí k potencionálnímu *dead-locku* dojde ke *commitu* všech zůčastněných transakcí a zpracování dalších uzlů v nové transakci

Operace, které by i nadále měly zamykat celou databázi jsou promazávání starých revizí a export či import kompletního dumpu databáze.

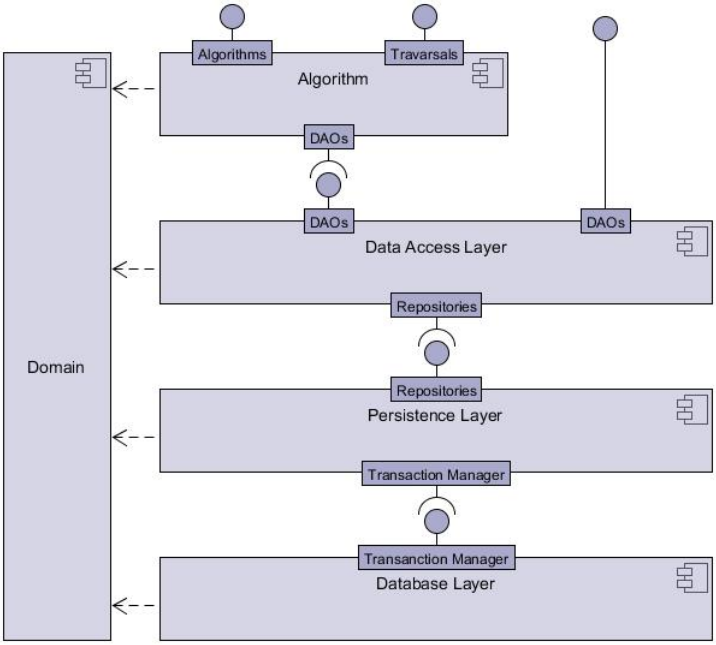
Takto navržený systém zámků umožní paralelní zápis do metadatového úložiště (byť zrychlení kvůli zamykání objektů a tedy potenciálnímu čekání v praxi nebude dosahovat počtu paralelních procesů).Existuje ale nadále silná závislost mezi architektonickými omezeními aplikace a omezeními na implementaci zámků. *Gremlin* (v žádné verzi) neposkytuje vlastní řešení zámků v grafových databázích. Zamykat objekty databáze pomocí databázových zámků je tak možné pouze u některých grafových databází. Nabízí se tak možnost sychnronizace pomocí *Java* konstruktů, jakou jsou *synchronized* metody/bloky, zámky, atomické proměnné atd. Tyto nástroje je ale možné použít pouze v případě zachování monolitické architektury serverové části aplikace. V případě její rozdělení na komponety spouštěné na různých *JVM* by bylo pro synchronizaci zámků nutné používat externí nástroje jako například relační databáze, nebo specifické nástroje jako je *Redisson*[3].

Vzhledem k tomu, že systém zámků řeší konzistenci dat z pohledu byznys logiky aplikace (integrita dat je zajištěna používáním transakcí), měl by být implementován komponentou obsahující tuto byznys logiku - konkrétně modul *Merger* (kapitola 4.3.2), který může jako jediná komponenta implementující byznys logiku aplikace pracovat s *necommitnutou* revizí metadatového úložiště.

5.1.2 Návrh komponent

V zádání diplomové práce je navrženo, aby byl jako architektonický styl pro návrh architektury splňující uvedená kritéria zvolena vícevrstvá architektura. Ta uvedeným kritériím zcela vyhovuje. Navržená architektura je popsána *UML*[4] *diagramem komponent* 5.1 a jednotlivé její komponenty jsou popsány v následujících podkapitolách.

[3] <https://redisson.org/>
[4] Unified Modeling Languege. Používaná notace 2.5.1 [53]



Obrázek 5.1: Upravená architektura modulu Connector

5.1.2.1 Doménový model

Jedním z důsledků splývání byznys a persistenční logiky je absence doménového objektového modelu serverové části aplikace. Místo něj je používán obecný model založený na třídách *Vertex* (vrchol) a *Edge* (hrana). Tato reprezentace dat v aplikaci má dva zásadní dopady:

- Model je příliš obecný (dokáže pojmout graf jakékoliv struktury, potažmo graf bez definované struktury), datový model metadatového úložiště je ale přesně specifikovaný (diagram 4.1) a jeho nedodržení znamená zanesení nekonzistence a tedy riziko selhání aplikace.
- Vzhledem k tomu, že zmíněný model poskytuje jako *SPI* dotazovací jazyk *Gremlin* (respektive *API Blueprints*) a to je implementováno *Java* knihovnou pro práci s grafovou databází *Titan*, umožňují instance tohoto modelu přímý přístup do grafové databáze a umožňuje tedy obcházení metod dedikovaných pro práci s grafovou databází.

Je evidentní, že tento doménový model není pro navrženou architekturu vhodný, byl tak vytvořen vlastní, **specifický doménový model**.

Základními omezeními návrhu doménového modelu jsou:

- *Rich model*: Doménový model je navržen jako tzv. *rich domain model*, obsahuje nejen definice entint a jejich parametrů, ale také základní logiku, kterou tyto entity a vazby mezi nimi představují.

- *POJO model*: Doménový model by měl být tvořen pouze *POJO třídami* [5].
- *Model neobsahuje perzistenční logiku*: Součástí doménového modelu by neměla být perzistenční logika. Toto omezení je v podstatě důsledkem omezení na *POJO* objekty - obsahoval-li by doménový model perzistenční logiku, musel by nutně obsahovat také transakční logiku, která je ale navržna tak (kapitola 5.1.1), že není realizovatelná pomocí *POJO* objektů. Perzistenční logika je tak realizovaná samostatnou komponentou (kapitola 5.1.2.3).

5.1.2.2 Databázová vrstva

Databázová vrstva slouží pouze pro připojení aplikace do databáze a pro dodání podpory pro externí indexovací nástroje, pokud je potřeba. Základní rozhraní této vrstvy tvoří *TransactionManager*, díky kterému jsou podporovány *deklarativní transakce* ve vyšších vrstvách aplikace a rozhraní reprezentující grafovou databázi[6] jako vstupní bod pro dotazy do databáze.

Tato funkcionalita je vyčleněna do samostatné vrstvy především kvůli principu separace zájmů. Také v některých případech umožňuje výměnu grafové databáze bez úprav vyšších vrstev - jedná se o případy, kdy je možné obě databáze dotazovat pomocí stejné verze jazyka *Gremlin*.

5.1.2.3 Perzistentní vrstva

Jak je zmíněno v kapitole 4.7, žádný ze zkoumaných nástrojů pro abstrakci objektově-grafového mapování nevyhovuje požadavkům *Manta Flow* a navržená architektura tak s žádný takový nástroj nepoužívá. Místo toho je navržena vlastní softwarová vrstva, která toto mapování provádí. Tou je právě *Perzistentní vrstva*. Jejím úkolem je poskytnout *API*, které pokryje všechny požadavky na dotazy do grafové databáze (a jeho implementaci). *Perzistentní vrstva* je *uzamčená vrstva* (viz 3.2.1), nemůže tedy být obcházena vyššími vrstvami při přístupu k nižší vrstvě.

API je tvořeno sadou *repository* objektů, které implementují (mimo jiné) *CRUD* operace a jsou inspirovány návrhovým vzorem *Repository pattern* [21]. Pro každou entitu reprezentovanou doménovým modelem existuje jeden *repository* objekt, přičemž každý z těchto objektů obsahuje (pokud je to pro příslušnou entitu relevantní) následující typy metod:

- **Create**: Metody slouží pro vytváření nových instancí entit (uzlů a hran) v grafové databázi. Parametrem je samotná instance obsahující parametry uzlu/hrany, interval platnosti uzlu/hrany a pokud existují, tak uzly, které jsou přímímy předky v hiearchii stromu grafových toků (v takovém případě je mezi těmito uzly vytvořena hrana). Například při vytváření nové instance entity *Node* je uzel spojen s rodičovckou instancí entity *Node* a/nebo s instancí entity *Resource*[7].

[5] *POJO (Plain Old Java Object)* je označení pro *obyčejný Java objekt*, tedy objekt, který není *J2EE Bean, Spring Bean, Entity Bean,* atd.

[6] V případě jazyka *Gremlin 2.x* poskytuje přístup do databáze rozhraní *com.tinkerpop.blueprints.Graph*.

[7] Již dříve bylo uvedeno, že graf datových toků ve skutečnosti nesplňuje stromovou strukturu. Je tak například možné, že entita *Node A* bude mít rodičovskou entitu *Node B* a *Resource C*, přičemž *B* a *C* nejsou součástí stejného podstromu grafu datových toků.

- **Find**: Vstupem *find* metod je jeden nebo více parametrů specifických pro danou entitu, přičemž kombinací těchto parametrů musí být vždy unikátně identifikovatelná maximálně jedna instance entity. Příkladem může být (jazykem *Gremlin* generované) id instance (jakékoliv entity), název entity *Resource*, nebo plně kvalifikované jméno entity *Node*. Metoda pak vrací nalezenou instanci entity - pokud existuje.

- **Update**: Operace *update* je u většiny entit redukována na úpravu intervalu platnosti entity, respektive na úpravu konce tohoto intervalu (začátek intervalu je po vytvoření instance neměnný). Ten je upravován při *updatu* (ať už úplném, nebo inkrementálním) metadatového úložiště operací *merge*. V případě, že se jedná o inkrementální *update*, je možné operaci volat s parametrem definujícím, že operace bude rekurzivní - bude uplatněna na celý podstrom entity (včetně entity samotné). Pokud by úpravou konce intervalu platnosti došlo k situaci, že by instance entity nebyla platná již v žádné revizi, je instance smazána pomocí operace *delete*.

  U entity *Flow* je navíc možné přidávat uživatelsky definované parametry entity. Tyto parametry nejsou součástí doménového ani datového modelu aplikace a nejsou ani využívány žádnými algoritmy byznys logiky aplikace. Všechny ostatní parametry (všech entit) jsou jasně definované v doménovém i datovém modelu aplikace, jsou nastavovány při vytváření instance entity operací *update* a dále jsou neměnné.

- **Delete**: Metody typu *delete* maže entity v grafové databázi bez ohledu na jejich interval platnosti. Operace může být (stejně jako operace *update*) rekurzivní.

- **Get***: Metody typu *get** slouží k dohledání entit, které jsou s vstupní entitou propojeny. Příkladem může být dohledání potomků (či předků) entit *Node* a *Resource*, dohledání parametrů (entit *Attribute*) entity *Node*, nebo dohledání entit, které jsou s vstupní entitou propojeny hranami *Flow*, nebo *MapsTo*.

  Metody typu *get** mají zpravidla další argumenty, které slouží k filtrování dohledávaných entit. Pokud kombinace těchto argumentů vytváří unikátní identifikaci instancí entity v rámci kontextu dotazu (vstupní entity), tak je podle jmenné konvence součástí názvů metody jednotné číslo dohledávané entity (například metoda *getChild*). Pokud argumenty unikátní identifikátor netvoří, je součástí názvu metody množné číslo dohledávané entity (například metoda *getChildren*).

- **Index Search**: Zatímco ostatní metody pro dotazování dat z grafové databáze (*find, get*, query*) implicitně využívají existujících interních indexů grafové databáze (a perzistentní vrstva od nich tak odstiňuje vyšší vrstvy), pro využití externích indexů (v případě *Manta Flow* se jedná o *Apache Lucene*) je nutné definovat vlastní metody. Argumenty těchto metod jsou definovány přesně dle definic těchto externích indexů tak, aby byl plně využit jejich potenciál. Případné další filtrování výsledků těchto dotazů (na základě parametrů neobsažených v externích indexech) tak musí probíhat již v komponentách realizujících byznys logiku aplikace.

- **Query**: Metody typu *query* slouží k obecnému dotazování entit. Argumenty těchto metod mohou být filtry na libovolné parametry entit, včetně parametrů entit s nimi spojenými (hranami a uzly). Na základě těchto argumentů je vygenerovaný databázový

dotaz, jehož součástí je uplatnění všech těchto filtrů - jedná se tak o výrazně efektivnější přístup než je například dotázání všech uzlů z grafové databáze a jejich následné filtrování ve vyšších vrstvách aplikace. Jedná se o nejobecnější nástroj, který API pro dotazování dat z grafové databáze nabízí.

Samotná perzistentní vrstva je dále rozčleněna na komponenty (viz diagram 5.2). Účelem tohoto rozdělení je striktní oddělení *API* komponenty a jeho implementace. Aplikace *Manta Flow* používá pro správu závislostí nástroj *Maven*[8], u jednotlivých modulů je tak definovány, jakým způsobem by měly být referencovány. Analogicky s tímto návrhem by měly být navržena také další vrstvy, které mohou být v budoucnosti do architektury přidávány. Jsou definovány tři moduly:

- *API*: Modul definuje rozhraní pro používání perzistentní vrstvy. Vyšší vrstvy využívající perzistenční vrstvu by měly referencovat právě tento modul.
- *Test*: Modul obsahuje třídy s definicemi testů pokrývajících funkcionalitou *API* - bez využití jakékoliv implementace *API*. Jednotlivé implementace *API* potom tyto testy implementují (poskytují implementaci *API*), čímž je zajištěno, že je vždy testováno chování *API*, a ne jen specifické detaily poskytnutých implementací. Modul závisí na modulu *API*.
- *Implementace*: Modul obsahující implementace vrstvy. Vyšší vrstvy využívající perzistenční vrstvu by tento modul měly referencovat, pouze ale s parametrem *scope=provided*. To zaručí, že není možné v kódu komponent používajích perzistenční vrstvu používat třídy, které jsou součástí této třídy a je tak zaručeno, že definované *API* není obcházeno. Modul je závislý na modulu *API*, modulu *Test* (*scope=test*) a na modulech představují *API* nižší vrstvy.

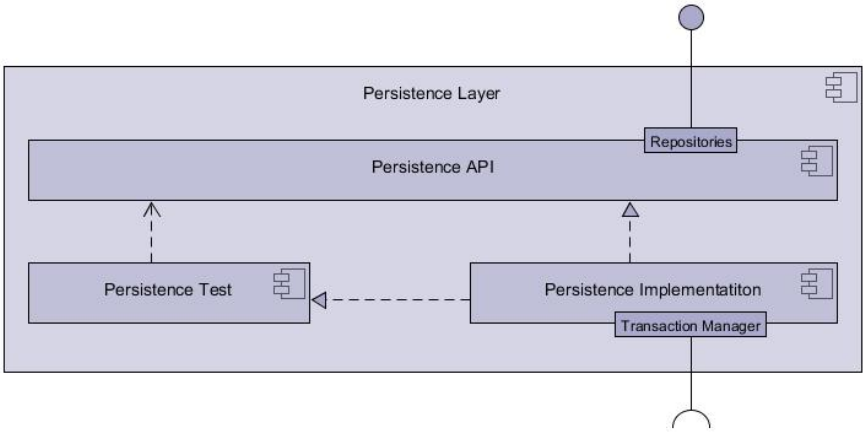
5.1.2.4 Vrstva datového přístupu

Vrstva datového přístupu je nejbližší vyšší vrstva perzistentní vrstvy, jejíž funkcionalitu rozšiřuje o kontrolu oprávnění uživatele na dotazovaná data[9]. *API* této vrstvy kopíruje *API* perzistentní vrstvy, přičemž každou metodu dotazující data[10] rozšiřuje o definici strategie, pomocí které dojde k validaci oprávnění uživatele. Z výsledků metod perzistentní vrstvy jsou tak v závislosti na zvolené strategii kontroly oprávnění případně odstraňována data, na která uživatel nemá oprávnění. Vrstva datového přístupu vrstva je *uzamčená vrstva*.

[8] <https://maven.apache.org/>

[9] Grafové databáze typicky neumožňují definovat oprávnění jednotlivých uživatelů tak, jak je tomu například u *RDBMS* databází.

[10] Kontrola oprávnění není nutná ve všech případech. Existuje premisa, že pokud má uživatel oprávnění na instanci entity, pak musí mít nuntě oprávnění na všechny předky této instance v hiearchii grafu datových toků.



Obrázek 5.2: Struktura perzistenční vrstvy

5.1.2.5 Algoritmy

Další vrstva - *Algoritmy* obsahuje funkcionalitu stávajícího modulu *Connector*, respektive jeho částí *Algorithm* a *Traversal*. Jedná se o první vrstvu realizující byznys logiku aplikace (funkcionalita je popsána v kapitole 4.3.1). Jedná se o *odemčenou vrstvu*, může být tedy obcházena vyššími vrstvami v případě, že chtějí použít přímo *API* vrstvy datového přístupu. Komponentami ve vyšších vrstvách serverové části aplikace jsou *Merger, Viewer, Exporter a Public Api*.

5.2 Orchestrace komponent Manta Flow

Tato sekce se zabývá návrhem orchestrace jednotlivých aplikací, které tvoří *Manta Flow*. Těmi jsou *Server, Client, Configurator a Updater*. Prioritním požadavkem v rámci této sekce je *F1*, tedy aby bylo možné pomocí *Updateru* aktualizovat všechny aplikace, které jsou součástí *Manta Flow*. Tomuto problému bude tedy věnována první část sekce. Pro zjednodušení budou v této sekci předpokládáno, že všechny tyto aplikace jsou nasazeny na jedno fyzické zařízení. Druhá část sekce bude věnována požadavkům *F2 a F3* a bude tedy již počítat s možností nasazení jednotlivých komponent na více fyzických zařízení.

5.2.1 Orchestrace na jednom zařízení

Primárním problémem v tomto případě je, jakým způsobem může být provedena aktualizace komponent *Configurator* a *Updater*. Aktualizace komponent *Manta Flow Server a Client* v rámci jednoho fyzického zařízení je *Updater* v současné době již bez problému schopen [34].

V případě komponenty *Configurator* je problémový fakt, že obě aplikace (tedy *Updater* i *Configurator*) by měly být nasazeny na stejném aplikačním serveru (pro jednoduchost a

efektivitu), přičemž k zaručení validní aktualizace aplikace musí být aplikační server restartován. To by vedlo k tomu, že v průběhu procesu aktualizace by byla zastavena i aplikace *Configurator*, kterou by uživatel v tu dobu používal.

U komponty *Updater* je problém analogický. Navíc je taková operace těžko proveditelná, protože aplikace nemůže používat své vlastní zdrjové soubory a konfigurace - v době běhu aplikace jsou používány a zamčeny.

V rámci návrhu řešení tohoto problému jsou definovány dvě nové komponenty:

- **Manta Flow Toolbox**: Vzhledem k tomu, že aplikace *Updater* a *Configurator* fungují velmi podobným způsobem, mají stejnou vnitřní architekturu a měly by být nasazovány společně, mohou být sloučeny pro snadnější operace s nimi do jedné aplikace - pracovně pojmenované *Manta Flow Toolbox*.[11]
- **Update Batch**: Vstupem komponenty *Updater* při aktualizaci některé komopnenty *Manta Flow* by měl být podle [34] archiv obsahující nové, či upravené soubory (zdrojové, nebo konfigurační) aktualizované komponenty a soubor obsahující instrukce k aktualizaci pro *Updater*. *Update Batch* je zobecněním tohoto konceptu, přičemž by měl obsahovat archiv pro každou z komponent, která má být aktualizována (*Server, Toolbox, Client*), a konzolovou aplikaci[12], která provede aktualizaci *Toolboxu* (popsáno sekvenčním diagramem B.4).

Server a *Client* mohou v rámci konkrétních nasazení obsahovat mnoho *customizací* zdrojového kódu i konfiguračních souborů. Jejich aktualizace může být tedy v závislosti na množšství těchto *customizací* velmi komplexní operace, která se neobejde bez účasti uživatele, který musí v některých případech zajistit správné sloučení některých souborů, které jsou při aktualizaci měněny. Na druhé straně u aplikací *Updater* a *Configurator* (respektive u aplikace *Toolbox*) žádná *customizace* nutné nejsou, jediná konfigurace, kterou obsahují, jsou odkazy na instance aplikací *Server a Client*. Jejich aktualizaci je tak možné provést účasti uživatele skriptem, nebo jednoduchou aplikací. Spuštění této aplikace je prvním krokem aktualizace všech komponent a jeho součástí je aktualizace *Toolboxu* a přesun vstupních dat pro aktualizace ostatních komponent na umístění, kde je *Toolbox* očekává[13].

Tato architektura zajišťuje, že v případě instalace všech komponent na jedno fyzické zařízení je možné všechny jednoduše aktualizovat.

5.2.2 Orchestrace po síti

V případě, že jsou instance serverové a klientské části aplikace nasazené na různá fyzycká zařízení, architektura popsaná v předchozí kapitole není dostačující k tomu, aby mohly být obě tyto komponenty aktualizovány najednou. Předpokládejme situaci, kdy je na jednom zařízení nasazen *Manta Flow Server* a *Toolbox*, a na druhém *Manta Flow Client*. Je tak možné aktualizovat (a konfigurovat) *Server* i *Toolbox* (dle návrhu v předchozí kapitole), není

[11] Přestože jsou aplikace sloučeny v jednu, měly by nadále fungovat jako samostatné moduly, které mezi sebou mají jasné rozhraní.

[12] Pro aktualizaci *Toolboxu* může být použit *shell* skript nebo jednoduchá *Java* aplikace

[13] Na rozdíl od popis procesu aktualizace v [34] už tak uživatel při aktualizaci ostatních komponent nemusí manuálně vstupy pro aktualizace dohledávat.

ale možné pomocí *Toolboxu* aktualizovat a konfigurovat *Client*. Primárním problémem je, že *Toolbox* nemá přístup na souborový systém zařízení, na kterém je *Client* nasazen.

Tento problém by bylo teoreticky možné vyřešit pomocí některého z protokolů umožňující vzdálený přístup k souborovému systému (například *FTP*). Tento přístup ale v současnosti není považovaný za standardní, ani za příliš bezpečný, není tak možné se spoléhat na to, že by byl v kontextu každého uživatele povolen. Nabízí se tak dnes již standardnější způsob síťové komunikace - protokol *HTTPS*.

V případě, kdy je *Toolbox* instalován společně se aktualizovanou komponentou na jednom zařízení, průběh aktualizace probíhá tak, jak je zachyceno na diagramu B.5. K tomu, aby komunikace mezi oběmi komponentami mohla probíhat obdobným způsobem i při použití protokolu *HTTPS*, je zapotřebí aby na obou stranách komunikace byly aplikace, které jsou v provozu nepřetržitě. Tomu neodpovídá současná architektura klientské částti aplikace, která je konzolovou aplikací spouštěnou *ad-hoc* jako dávková aplikace. Je tedy navržena úprava architektury aplikace *Manta Flow Client* tak, aby reflektovala výše uvedené požadavky.

Jak je zřejmé z diagramu B.6, který zobrazuje průběh aktualizace aplikace *Manta Flow Client* přes *HTTPS*, je nutné, aby se tato aplikace skládala ze dvou komponent:

- *Dávková aplikace*: Dávková *Java* aplikace, zcela odpovídá aktuální podobě klientské části aplikace, včetně způsobu použití.
- *Webová aplikace*: Webová aplikace bude obalovat dávkovou aplikaci a bude sloužit primárně pro komunikaci *Manta Flow Toolbox*. Tato aplikace by měla být navržena striktně dle architektonického stylu *REST* a měla by být pokud možno minimalistická co se týká vlastní konfigurace a poskytovaných funkcí. Vznikem této komponenty totiž vzniká problém s její aktualizací. Ten je řešitelný vygenerováním aktualizačního skriptu *Toolboxem* (na základě rozhodnotí uživatele o případném sloučení konfigurací).

Diagram B.7 ukazuje změna v nasazení jednotlivých aplikací.

5.3 Možnosti horizontálního škálování aplikace

V kapitole 4.5.3 je uvedeno, že *Manta Flow* může mít v budoucnu problémy s množstvím dat, které zpracovává. Aplikace může narazit na mezní hranici kapacity metadatového úložiště, která je v současné době omezena především používáním *embedded* databáze *Persistit* jako podkladové vrstvy pro *Titan*. Současně může být v některých kontextech problematický i výpočetní čas operací byznys logiky aplikace (viz sekce 4.5.2), který při současné architektuře narůstá lineárně v závislosti na vstupních datech.

V této kapitole jsou popsány možnosti, jak může být stávající architektura aplikace *Manta Flow* (nutně zahrnující změny popsané v kapitole 5.1 a volitelně zahnující změny popsané v kapitole 5.2) postupnými kroky upravována tak, aby průběžně vyhovovala zvyšujícím se požadavkům na horizontální škálování.

5.3.1 Škálování grafové databáze

Použití *embedded* databáze *Persistit* je zjevně potenciální úzké hrdlo aplikace a jeho nahrazení horizontálně škálovatelnou grafovou databází je tak první změna architektury směrem k vyšší škálovatelnosti celé aplikace. Jsou identifikovány dva hlavní důvody, které vedou k tomu, že je používána *embedded* databáze.

Prvním je výkon aplikace, který by při použití stávající architektury a jiné než *embedded* databáze výrazně utrpěl - síťová komunikace představuje při komunikaci aplikace se vzdálenou databází vždy režii navíc, v případě *RDBMS* databází (a většiny *NoSQL* databází) je tato položka ale pro celkový výkon aplikace zanetbatelná. Důvodem je, že dotazy vykonávané těmito databázemi jsou často velmi komplexní a jejich vykonání trvá zpravidla déle, než síťová komunkace mezi aplikací a databází. U grafových databází tomu ale tak být nemusí, jazyky pro dotazování těchto databází jsou stále prochází vývojem a ne vždy poskytují takové nástroje, aby pomocí nich mohli být realizovány všechny komplexní grafové algoritmy. To vede v některých případech k tomu, že granularita dotazů do grafové databáze je velmi vysoká (dotazy se v některých případech omezují pouze na okolní hrany/uzly) a čas vykonávání těchto dotazů je tak srovnatelný, nebo dokonce menší, než doba potřebná pro síťovou komunikaci mezi aplikací a grafovou databází.

Druhým omezením je závislost aplikace na dotazovacím jazyce *Gremlin 2.x*, který je vzhledem k možnosti přímého dotazování grafové databáze pomocí *PTM* transakcí (viz sekce 4.5.4) používán všemi moduly serverové části aplikace a je tak obtížně nahraditelný. Některé uvažováané databáze, které by mohly případně aktuálně používaný *Perzistit* nahradit jazyk *Gremlin* ve verzi *2.x* nepodporují.

Změna architektury komponenty *Connector* navržená v kapitole 5.1 oba tyto problémy řeší. Všechny dotazy do grafové databáze jsou definovány v nově navržené perzistentní vrstvě a jsou tak jednoduše upravitelné (včetně změny dotazovacího jazyk, kterým jsou realizovány) bez dopadů na ostatní komponenty (díky definovanému privátnímu *API*). To je navrženo tak, aby byla ve vhodných případech snížena granularita dotazů a byl tak omezen problém režie síťové komunikace aplikace se vzdálenou databází. Příkladem jsou metody typu *query*, které umožňují provádění komplexních dotazů na starně grafové databáze.

Díky tomu je otevřena možnost použití vzdálené grafové databáze, a je tedy možné **využít horizontální škálovatelnost grafových databází**, která je nativní vlastností většiny z nich. Je zřejmé, že lépší škálovatelnost budou vykazovat spíše komplexnější dotazy do grafové databáze, než dotazy atomické. Je proto možné, že pro maximalizaci využití horizontální škálovatelnosti grafové databáze bude nutné některé algoritmy, nebo jejich části, implementovat zcela pomocí komplexních grafových dotazů (například pomocí nadstavby jazyka *Gremlin - Pipes*). Obdobným způsobem, jako jsou navrženy metody typu *query* mohou být navrženy metody pro průchody grafem datových toků, mohly by tak vznikat komplexní průchody, kde by algoritmus průchodu nebyl vázán ná konkrétní dotazovací jazyk, zároveň by ale byly tyto průchody vykonávány kompletně na straně grafové databáze.

Horizontálně škálovatelná grafová databázy by měla být dostatečným řešením pro potřeby aplikace *Manta Flow* a zároveň se jedná o relativně neinvazivní změnu v architektuře aplikace. *Manta Flow* je software dodávaný tzv. *on premise*, uživatel tedy instaluje aplikaci na vlastní infrastrukturu a má nad ní plnou kontrolu. Uživatel si tak může sám rozhodnout, zda si vystačí s *embedded* grafovou databází, nebo zda potřebuje horizontálně škálovatelnou

grafovou databázi a na jaké infrastruktuře bude v tom případě nasazena. Díky navržené vícevrstvé architektuře není problém, aby aplikace podporovala obě možnosti a použití konkrétní implementace databázové (a perzistentní) vrstvy bylo konfigurovatelné.

5.3.2 Škálování Java komponent aplikace

Dalším krokem pro zlepšení škálovatelnosti aplikace může být potenciálně horizontální škálování samotných *Java* komponent aplikace. Protože tento krok již předpokládá použití horizontálně škálovatelné grafové databáze, je cílem v tomto případě již pouze zvýšení výkonu aplikace - schopnost analýzy zdrojových dat velkého objemu je již zaručena. Na elementární bázi to umožňuje i stávající architektura klient-server (v případě klientské části aplikace). Vzhledem k tomu, že klientská část aplikace obsahuje mimo samotné extrakce dat (databázových slovníků, zdrojových kódů, transformací, atd.) ze zdrojových systémů také logiku pro jejich analýzu, může několik instancí klientské části aplikace výkon zvýšit významně.[14] Teoreticky by také bylo možné paralelní nasazení několika serverových komponent, to by si ale vyžádalo úpravu některých algoritmů a především zavedení dalšíhi sdíleného zdroje (krom vstupů a grafové databáze), který by sloužil pro synchronizaci zámků v databázi (viz kapitola 5.1.1). Pro nadměrnou komplexitu a nejistou efiktivitu tak tento přístup není možné doporučit.

Efektivněji škálovatelná by serverová část aplikace byla, pokud by její architektura byla upravena postupným vyčleňováním jednotlivých služeb podle architektonického stylu *microservices*. Vzhledem ke komplexní a specifické infrastruktuře, kterou takové řešení vyžaduje, by ale taková architektura nebyla vhodná pro software dodávaný jako *on premise*. Přechod na software *on demand* je ale rozhodnutí zahrnující více faktorů než jen architektonická omezení aplikace a v současné době není vzhledem k prototypu uživatele aplikace možný.

[14] Paralelní instance klientské části aplikace ale není možné (či efektivní) použít ve všech případech, v případě silně propojených zdrojových systémů by musely jednotlivé instance sdílet téměř všechna data.

Kapitola 6

Implementace prototypu

Součástí práce je prototypová implementace architektury navržené v kapitole 5.1, konkrétně komponent *Doménový model, Databázová vrstva, Perzistentní vrstva a Vrstva datového přístupu*. Prototypová implementace také obsahuje část byznys logiky vyšších vrstev aplikace, na jejímž základě je provedena validace navržené architektury a navrženého *API* perzistentní vrstvy. Na obrázku B.10 je *UML* diagram komponent prototypové implementace. V této kapitole jsou blíže popsány důležité či nestandardní části implementace a je diskutována validnost vytvořeného návrhu.

6.1 Doménový model

Doménový model definuje typy entit, se kterými může aplikace pracovat a možné parametry těchto entit. Prakticky tak definuje také schéma (datový model) grafové databáze, přestože ta nutně nemusí koncept schémat podporovat. Hlavní entity doménového modelu jsou popsány *Business Domain Model (BDM) UML diagramem* B.8 (cílem diagramu je popsat entity a vztahy mezi nimy, ne konkrétní implementaci). Při porovnání s datovým modelem metadatového úložiště je patrné, že:

- Pro každý typ uzlu, který je součástí datového modelu existuje ekvivalent v doménovém modelu (s adekvátně definovanými parametry). Výjimku tvoří umělé typy uzlů, tedy *REVISION_ROOT, SOURCE_ROOT a SUPER_ROOT*.
- Doménový model neobsahuje tzv. *řídící hrany* datového modelu, tedy hrany, které tvoří hiearchickou strukturu uzlů. Jediným netechnickým parametrem těchto hran je interval platnosti uzlů. Ten je ale spíše vlastností uzlů samotných, nikoliv jejich řídících hran (jeho umístění na řídící hrany v datovém modelu je důsledek optimalizace výkonu aplikace). Nic tedy nebrání tomu, aby byl doménový model zjednodušen a tyto hrany z něj odebrány.
- Doménový model obsahuje ekvivalenty hran typu *DIRECT, FILTER a MAPS_TO* datového modelu. Tyto hrany mají vlastní interval platnosti, který je sice omezen intervaly platnosti uzlů, které spojují, ale může se od těchto intervalů lišit. Hrany také obsahují další parametry podstatné pro analýzu datových toků. Entita doménového modelu *Flow* zahrnuje hrany datového modelu *DIRECT i FILTER*.

6.2 Databázová vrstva

Databázová vrstva obsahuje dvě stežejní třídy (které musí obsahovat vždy) a to jsou implementace tříd `org.springframework.transaction.support.ResourceTransactionManager` a `com.tinkerpop.blueprints.Graph`. Tato rozhraní *de-facto* definují *API* databázové vrstvy. *Transaction manager* musí být implementován, aby mohly vyšší vrstvy využívat deklarativní transakční model, který je pro návrh celé vícevrstvé architektury velmi důležitý. *Graph* zpřístupňuje data uložená v grafové databázi dotazovacímu jazyku *Gremlin 2.x*. V tomto případě nevadí, že je *API* vrstvy závislé na konkrétním programovacím jazyku - implementace nejbližší vyšší vrstvy (perzistentní vrstvy) je (a vždy bude) z velké části tvořena právě tímto dotazovacím jazykem.

6.3 Perzistentní vrstva

Implementace perzistentní vrstvy je tvořena především implementací *repository* objektů definovaných v *API* vrstvy (popsáno v kapitole 5.1.2.3). Dokumentace celého API perzistentní vrstvy je dostupná na přiloženém CD (viz obsah CD v příloze C).

6.3.1 Vrstva *mapperů*

Modul obsahující implementaci perzistentní vrstvy obsahuje další vnitřní vrstvu tzv. *mapperů* - objektů provádějících mapování vrcholů a hran grafové databáze (reprezentovaných *Gremlin* třídami *Vertex a Edge*) na objekty doménového modelu. Pro účely prototypu byla zvolena implementace pro (aplikací aktuálně používaný) jazyk *Gremlin 2.x*.

6.3.2 Dotazy do metadatového úložiště

Jednoduché dotazy jsou implementovány nativními dotazy jazyka *Gremlin*. Příklad 6.1 ukazuje mapování entity *Flow* pomocí jazyka *Gremlin 2.x*.

```
// get id
String edgeId = ((RelationIdentifier) edge.getId()).toString();

// get flow type
FlowType flowType = ((EdgeLabel.getLabel(edge.getLabel()).getFlowType()));

// find & map start node
Node startNode = nodeMapper.map(edge.getVertex(Direction.OUT));
// find & map end node
Node endNode = nodeMapper.map(edge.getVertex(Direction.IN));

// get predefined and custom properties
Double validFrom = null;
Double validTo = null;
String callId = null;
Map<String, String> properties = new HashMap<String, String>();
for (String key : edge.getPropertyKeys()) {
    if (TRAN_START.p().equals(key)) {
        validFrom = edge.getProperty(key);
```

```
    } else if (TRAN_END.p().equals(key)) {
        validTo = edge.getProperty(key);
    } else if (CALL_ID.p().equals(key)) {
        callId = edge.getProperty(key);
    } else {
        properties.put(key, (String) edge.getProperty(key));
    }
}

RevisionInterval revisionInterval = new RevisionInterval(validFrom, validTo);

// create flow
Flow flow = new Flow(edgeId, startNode, endNode, revisionInterval, flowType,
    callId, properties);
```

Příklad 6.1: Mapování entity *Flow* (implementace pomocí *Gremlin 2.x*)

Pro větší efektivitu dotazů byly v některých případech nativní *Gremlin* dotazy nahrazeny dotazy pomocí další součástí *TinkerPop* projektu - *Pipes*[1]. *Pipes* slouží právě jako nástroj pro komplexnější průchody grafem pro jazyk *Gremlin 2.x*[2]. Průchodu grafu pomocí *Pipes* je ukázán na příkladu 6.2, kde uvedený kód nalzene instanci entity *Node* podle kvalifikovaného jména (jedná se tedy o průchod víceúrovňovou hierchií grafu datových toků).

```
GremlinPipeline<Vertex, Vertex> pipeline = new GremlinPipeline(resource);

for (NodeQualifiedName.QualifiedNamePart qnPart : qn.getParts()) {
    pipeline = pipeline
        .inE(HAS_PARENT.l())
            .has(TRAN_START.p(), lte, revisionInterval.getRevisionStart())
            .has(TRAN_END.p(), gte, revisionInterval.getRevisionEnd())
            .cast(Edge.class)
        .outV()
            .has(NODE_NAME.p(), qnPart.getNodeName())
            .has(NODE_TYPE.p(), qnPart.getNodeType().getId())
            .cast(Vertex.class);
}

Node node = nodeMapper.map((Vertex) pipeline.next());
```

Příklad 6.2: Nalezení uzlu dle kvalifikovaného jména (implementace pomocí *Pipes*)

6.3.3 Rozhraní pro *query* metody

V kapitole 5.1.2.3 je popsán návrh *query* metod. Ty by měly sloužit k obecnému dotazování dat z metadatového úložiště podle typu entity. Jedním z důvodů návrhu těchto metod je snaha o umožnění tvorby komplexnějších dotazů, které by měly být efektivnější, než kompozice atomických dotazů. Výsledkem toho je, že vstupem pro tyto metody jsou komplexní do sebe zanořené dotazovací objekty. Aby byl tento koncept reálně použitelný, bylo nutné vytvořit samostatné rozhraní pro tvorbu těchto objektů - jinak by tvorba dotazovacích objektů byla příliš komplikovaná, nutně by navíc musela obsahovat některé perzistenční detaily,

[1] <https://github.com/tinkerpop/pipes/wiki>

[2] *Gremlin 3.x* nahrazuje *Pipes* navazujícím řešením s názvem *Graph Traversal*

od kterých by měl být vývojář při používání těchto metod odstíněn. Návrh byl inspirován návrhovým vzorem *Builder Pattern* a stylem *Fluent Interface* [26]. Ukázka 6.3 obsahuje dotaz na všechny *nullable* sloupce ze specifické tabulky, schématu a databáze. Pomocí stejného návrhu by bylo možné provádět průchody grafem datavých toků.

```
nodeRepository.query(
    Query
        .nodes()
            .validIn(new RevisionInterval(1d, 1d))
            .hasType(COLUMN)
        .attributes()
            .hasName("COLUMN_NULLABILITY")
            .hasValue("nullable")
        .node()
        .parent()
            .hasName("DF_EDGE_ATTRIBUTE")
            .hasType(TABLE)
        .parent()
            .hasName("METADATA")
            .hasType(SCHEMA)
        .parent()
            .hasName("ORCL")
            .hasType(DATABASE)
    .q()
);
```

Příklad 6.3: Fluent interface query metod

6.4 Vrstva datového přístupu

API této vrstvy v zásadě kopíruje *API* perzistentní vrstvy. U metod, u kterých musí být ověřována práva uživatele na dotazovaná data, je ale navíc přidán parametr reprezentující strategii, pomocí které budou data ověřována. Asi tedy nepřekvapí, že je vrstva implementována pomocí návrhovému vzoru *Strategy* - aktuálně existují dvě strategie (v budoucnu může být ale tento seznam rozšířen):

- *Oprávnění na základě pohledů a rolí*: Je zaveden nový termín *pohled*. Ten reprezentuje část dat uložených v metadatovém úložišti - každý pohled se skládá ze sady zahrnutých a vyloučených objektů ve smyslů hiearchie grafu datových toků (entity *Node* a *Resource*). Pohledů může být teoreticky neomezené množství a mohou se navzájem překrývat. Uživatelům jsou pak přidělována oprávnění na jednotlivé pohledy na základě jejich *LDAP*[3] rolí.
- *Neomezená oprávnění*: Existují operace, u kterých není za žádných okolností žádoucí oprávnění kontrolovat. Například kompletní exporty metadatového úložiště. Proto existuje strategie přidělující všem uživatelům neomezená oprávnění.

Pro implementaci strategie kontroly oprávnění na základě pohledů je navíc za účelem rozdělení zodpovědností použit návrhový vzor *Chain of responsibility* (součástí implementace je také vlastní *cache* a další logika). Tato implmenetace je pospána diagramem tříd B.9.

[3] Lightwight Directory Access Protocol

6.5 Validace a testování

Funkcionalita všech implementovaných modulů byla pokryta jednotkovými testy. Tabulka 6.1 obsahuje údaje o pokrytí implementace jednotkovými testy.

Tabulka 6.1: Pokrytí prototypové impolementace návrhu jednotkovými testy

Třídy	Metody	Řádky
86% (150/174)	73% (726/994)	78% (3476/4438)

V rámci validace návrhu a prototypové implementace vícevrstvé architektury byly pomocí vytvořeného prototypu implementovány některé operace vyšších vrstev aplikace, konkrétně:

- *Merge*: Byla implementována operace *merge*, tedy stěžejní část operace updatu metadatového úložiště. Operace je blíže popsána v kapitole 4.3.2.
- *Flow Algorithm*: Byl také implementován základní algoritmus pro hledání datových toků. Tento algoritmus je součástí mnoha sloužitějších algoritmů, které *Manta Flow* používá.

Pro oba zmíněné algoritmy je v rámci stávající implementace definována řada jednotkových testů. Základním předpokladem validace vytvořeného prototypu je, že podaří-li se pomocí nově navržené architektury implementovat obě tyto funkcionality a budou-li nadále úspěšně procházet všechny testy pokrývající tyto funkcionality, podařilo se navrhnout architekturu a implementovat prototyp vyhovující zadání práce.

To se podařilo, lze tedy prohlásit, že **prototyp je validní**.

Další typy testů (např. *performance* testy) nebyly provedeny, protože v kontextu této práce nedávají smysl. Cílem validace je ověřit vhodnost navržené architektury pro konkrétní funkcionality *Manta Flow*. Vytvořená implementace (byť rozsáhlá - více než 22 tisíc *LOC*) je navíc pouze prototypem, neobsahuje tak všechny optimalizace výkonu, které zahrnuje stávající aplikace.

Kapitola 7

Závěr

TODO

Literatura

[1] *Microservices - A definition of this new architectural term* [online]. mar 2014. [cit. 02. 02. 2018]. Dostupné z: <https://martinfowler.com/articles/microservices.html>.

[2] AL-FURAIH, I. – RANKA, S. *Memory hierarchy management for iterative graph structures.* Orlando, FL, USA : IEEE, 1st edition, 1998.

[3] AMAZON. *Release: AWS Lambd* [online]. nov 2014. [cit. 02. 02. 2018]. Dostupné z: <https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13/>.

[4] ANGELS, R. – GUTIERREZ, C. *Survey of Graph Database Models* [online]. feb 2008. [cit. 21. 11. 2017]. Dostupné z: <https://www.cse.iitk.ac.in/users/smitr/PhDResources/SurveyofGraphDatabasesModels.pdf>.

[5] APACHE TINKERPOP. *Gramlin 3.3* [online]. aug 2017. [cit. 27. 11. 2017]. Dostupné z: <http://tinkerpop.apache.org/>.

[6] ARANGO DB. *Benchmark: PostgreSQL, MongoDB, Neo4j, OrientDB and ArangoDB* [online]. oct 2015. [cit. 28. 11. 2017]. Dostupné z: <https://www.arangodb.com/2015/10/benchmark-postgresql-mongodb-arangodb/>.

[7] AURELIUS. *Titan 0.4 Documentation* [online]. apr 2014. [cit. 26. 11. 2017]. Dostupné z: <http://titan.thinkaurelius.com/wikidoc/0.4.4/Home.html>.

[8] AURELIUS. *Titan 0.4 Documentation: Using Persistit* [online]. apr 2014. [cit. 17. 04. 2018]. Dostupné z: <http://titan.thinkaurelius.com/wikidoc/0.4.4/Using-Persistit.html>.

[9] BARNARD, S. – POTHEN, A. – SIMON, H. *A spectral algorithm for envelope reduction of sparse matrices* [online]. feb 1993. [cit. 21. 11. 2017]. Dostupné z: <https://www.nas.nasa.gov/assets/pdf/techreports/1993/rnr-93-015.pdf>.

[10] BATTITI, R. – BERTOSSI, A. *Greedy, prohibition, and reactive heuristics for graph partitioning.* Orlando, FL, USA : IEEE Transactions on Computers, 1999.

[11] BLOCH, J. *How to Design a Good API and Why it Matters* [online]. nov 2006. [cit. 30. 01. 2018]. Dostupné z: <https://www.infoq.com/presentations/effective-api-design>.

[12] BOLDI, P. – VIGNA, S. *The WebGraph Framework I: Compression Techniques* [online]. nov 2004. [cit. 21. 11. 2017]. Dostupné z: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.993&rep=rep1&type=pdf>.

[13] C. GHEZZI, M. J. – MANDRIOLI., D. *Fundamentals of Software Engineering*. Upper Saddle River, New Jersey, USA : Prentice-Hall, 2nd edition, 2003.

[14] CHAO, J. *Imperative vs. Declarative Query Languages: What's the Difference?* [online]. sep 2016. [cit. 27. 11. 2017]. Dostupné z: <https://neo4j.com/blog/imperative-vs-declarative-query-languages/>.

[15] CHEN, C. et al. 33rd International Conference on Very Large Data Bases. In *Towards graph containment search and indexing*, s. 926–937, 2007.

[16] CLARK, D. D. – TENNENHOUSE, D. L. ACM SIGCOMM'90 Symposium. In *Architectural considerations for a new generation of protocols*, s. 200–208, 1990.

[17] CORMEN, T. H. et al. *Introduction to Algorithms*. Cambridge, Massachusetts London, England : The MIT Press, 3rd edition, 2009.

[18] CUTHILL – MCKEE. Reducing the bandwidth of sparse symmetric matrices. *ACM Annual Conference*. 1969, s. 157–172.

[19] DEMLOVá, M. *Logika a grafy* [online]. nov 2017. [cit. 14. 11. 2017]. Dostupné z: <http://math.feld.cvut.cz/demlova/teaching/lgr/text_lgr_2017.pdf>.

[20] DIESTEL, R. *Graph Theory*. Spring Street, New York, USA : Springer International Publishing AG, 2nd edition, 2000.

[21] DORFMANN, H. *THE EVOLUTION OF THE REPOSITORY PATTERN - BE AWARE OF OVER ABSTRACTION* [online]. jun 2016. [cit. 21. 04. 2018]. Dostupné z: <http://hannesdorfmann.com/android/evolution-of-the-repository-pattern>.

[22] EVANS, E. *NoSQL: A Relational Database Management System* [online]. dec 2009. [cit. 25. 11. 2017]. Dostupné z: <http://blog.sym-link.com/2009/05/12/nosql_2009.html>.

[23] FIDUCCIA, C. M. – MATTHEYSES, R. M. SIAM Journal on Numerical Analysis. In *A linear-time heuristic for improving networkpartitions*, s. 175—181, 1982.

[24] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

[25] FOWLER, M. *Richardson Maturity Model* [online]. mar 2010. [cit. 31. 01. 2018]. Dostupné z: <https://martinfowler.com/articles/richardsonMaturityModel.html>.

[26] FOWLER, M. – EVANS, E. *Fluent Interface* [online]. dec 2005. [cit. 03. 05. 2018]. Dostupné z: <https://martinfowler.com/bliki/FluentInterface.html>.

[27] FUGGETTA, A. – PICCO, G. P. – VIGNA, G. Understanding code mobility. *IEEE Transactions on Software Engineering*. may 1998, s. 342–361.

[28] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Upper Saddle River, New Jersey, USA : Addison-Wesley, 1st edition, 1994.

[29] GAREY, M. R. – JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman and Co., 1st edition, 1990.

[30] GARLAN, D. – PERRY, D. E. IEEE Transactions on Software Engineering. In *Introduction to the special issue on software architecture*, s. 269–274, 1995.

[31] GARLAN, D. – SHAW., M. An introduction to software architecture. *School of Computer Science Carnegie Mellon University*. 1 1994, s. 1–39.

[32] GEORGE, A. – LIU, J. W. H. *An automatic nested dissection algorithm for irregular finite element problems*. Waterloo, Ontario, Canada : SIAM Journal on Numerical Analysis, 1978.

[33] GIUGNO, R. – SHASHA, D. In Ieee International Conference in Pattern Recognition. In *GraphGrep: A fast and universal method for querying graphs*, s. 112–115, 2002.

[34] GONDEK, P. Návrh a prototypová implementace nástroje pro instalace a aktualizace webových Java aplikací, 2016.

[35] HE, H. – SINGH, A. K. 22nd International Conference on Data Engineering. In *Closure-Tree: An index structure for graph queries*, s. 38–52, 2006.

[36] (IDC), I. D. C. *Data Growth, Business Opportunities, and the IT Imperatives* [online]. apr 2014. [cit. 27. 11. 2017]. Dostupné z: <http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm>.

[37] IEEE. *Information technology – Portable Operating System Interface (POSIX) – Part 1: Base Definitions* [online]. aug 2003. [cit. 27. 01. 2018]. Dostupné z: <https://www.iso.org/standard/38789.html>.

[38] JACKSON, D. *Software Abstractions: Logic, Language, and Analysis*. Cambridge, Massachusetts, London, England : The MIT Press, 1st edition, 2006.

[39] JIANG, H. et al. 23rd International Conference on Data Engineering. In *GString: A novel approach for efficient search in graph databases*, s. 566–575, 2007.

[40] KOUTRA, D. et al. *Algorithms for Graph Similarity and Subgraph Matching* [online]. dec 2011. [cit. 25. 11. 2017]. Dostupné z: <https://people.eecs.berkeley.edu/~aramdas/reports/DBreport.pdf>.

[41] KOVář, M. Benchmark grafových databází pro potřeby data lineage, 2018.

[42] L. BASS, P. C. – KAZMAN, R. *Software Architecture in Practice*. Upper Saddle River, New Jersey, USA : Addison-Wesley, 1st edition, 1998.

[43] LAHDENMäKI, T. – LEACH, M. *Relational Database Index Design and the Optimizers*. Hoboken, New Jersey, USA : A John Wiley and sons, Inc., 1st edition, 2005.

[44] LAL, M. *Neo4j Graph Data Modeling*. Livery Street, Birmingham, UK : Packt Publishing Ltd., 1st edition, 2015.

[45] LANEY, D. 3D Data Management: Controlling Data Volume, Velocity and Variety. *META Group*. 2001, s. 1–4.

[46] LINGAM, P. *Analysis of Real-Time Multi version Concurrency Control Algorithms using Serialisability Graphs* [online]. aug 2010. [cit. 22. 02. 2018]. Dostupné z: <https://pdfs.semanticscholar.org/efff/371acb8678353480e749f053a6782a368c7a.pdf>.

[47] LITTLE, M. – MARON, J. – PAVLIK, G. *Programmatic Transaction Model*. Upper Saddle River, New Jersey, USA : Prentice Hall, 1st edition, 2004.

[48] MELL, P. – GRANCE, T. *The NIST Definition of Cloud Computing* [online]. sep 2011. [cit. 29. 01. 2018]. Dostupné z: <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>.

[49] MOLITOR, G. Analýza a návrh konfiguračního nástroje pro BI analytiky v projektu Manta Tools. 01 2018.

[50] NEO TECHNOLOGY. *Cypher* [online]. [cit. 27. 11. 2017]. Dostupné z: <http://neo4j.com/docs/developer-manual/current/cypher/>.

[51] NEO TECHNOLOGY. *Meet openCypher: The SQL for Graphs* [online]. oct 2015. [cit. 27. 11. 2017]. Dostupné z: <https://neo4j.com/blog/open-cypher-sql-for-graphs/>.

[52] NITTO, E. D. – ROSENBLUM, D. The 1999 International Conference on Software Engineering. In *Exploiting ADLs to specify architectural styles induced by middleware infrastructures*, s. 13–22, 1999.

[53] Object Management Group. *About the Unified Modeling Language Specification: Version 2.5.1* [online]. dec 2017. [cit. 18. 04. 2018]. Dostupné z: <https://www.omg.org/spec/UML/>.

[54] ORIENT DB. *Consistency* [online]. [cit. 28. 11. 2017]. Dostupné z: <http://orientdb.com/docs/last/Graph-Consistency.html>.

[55] ORIENT DB. *Multi-Model* [online]. [cit. 28. 11. 2017]. Dostupné z: <http://orientdb.com/docs/last/Tutorial-Document-and-graph-model.html>.

[56] PAUTASSO, C. – ZIMMERMANN, O. – LEYMANN, F. 17th International World Wide Web Conference. In *RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision*, s. 805–814, 2008.

[57] Pivotal Software Inc. *Declarative transaction management* [online]. dec 2009. [cit. 20. 04. 2018]. Dostupné z: <https://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch11s05.html>.

[58] RICHARDS, M. *Software Architecture Patterns*. Sebastopol, CA, USA : O'Reilly Media, Inc., 1st edition, 2015.

[59] ROBERTS, M. *Serverless Architectures* [online]. jun 2016. [cit. 01. 02. 2018]. Dostupné z: <https://martinfowler.com/articles/serverless.html>.

[60] SADALAGE, P. J. – FOWLER, M. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Upper Saddle River, New Jersey, USA : Addison-Wesley, 3rd edition, 2013.

[61] SAKR, S. 14th International Conference on Database Systems for Advanced Applications. In *GraphREL: A decompositionbased and selectivity-aware relational framework for processing sub-graph queries*, s. 123–137, 2009.

[62] SHAW, M. Toward higher-level abstractions for software systems. *Data and Knowledge Engineering*. may 1990, s. 119–128.

[63] SHAW, M. – CLEMENTS, P. Twenty-First Annual International Computer Software and Applications Conference. In *A field guide to boxology: Preliminary classification of architectural styles for software systems*, s. 6–13, 1997.

[64] SIMON ST. LAURENT, J. J. – DUMBILL, E. *Programming Web Services with XML-RPC*. St George's Yard, Farnham Surrey, UK : O'Reilly, 1st edition, 2001.

[65] SýKORA, J. Incremental update of data lineage storage in a graph database. 12 2017.

[66] SOLID IT. *DB-Engines Ranking - Trend of Graph DBMS Popularity* [online]. [cit. 28. 11. 2017]. Dostupné z: <https://db-engines.com/en/ranking_trend/graph+dbms>.

[67] SPRING. *Understanding HATEOAS* [online]. nov 2018. [cit. 31. 01. 2018]. Dostupné z: <https://spring.io/understanding/HATEOAS>.

[68] Syncleus, Inc. *Ferma Benchmarks* [online]. jan 2015. [cit. 22. 04. 2018]. Dostupné z: <http://syncleus.com/Ferma/comparing_the_alternatives/>.

[69] TAYLOR, R. N. – MEDVIDOVIC, N. – DASHOFY, E. M. *Software Architecture: Foundations, Theory, and Practice*. Hoboken, New Jersey, USA : A John Wiley and sons, Inc., 1st edition, 2009.

[70] TIAN, Y. et al. *SAGA: A subgraph matching tool for biological graphs*. Oxford, England : Bioinformatics, 1st edition, 2007.

[71] TINKERPOP. *Blueprints* [online]. feb 2010. [cit. 27. 11. 2017]. Dostupné z: <https://github.com/tinkerpop/blueprints/tree/0.1>.

[72] TINKERPOP. *Gremlin 0.1* [online]. dec 2009. [cit. 27. 11. 2017]. Dostupné z: <https://github.com/tinkerpop/gremlin/tree/0.1>.

[73] TINKERPOP. *Gremlin 2.6* [online]. sep 2014. [cit. 27. 11. 2017]. Dostupné z: <https://github.com/tinkerpop/gremlin/tree/2.6.0>.

[74] UMAR, A. *Object-Oriented Client/Server Internet Environments*. Upper Saddle River, New Jersey, USA : Prentice Hall PTR, 1st edition, 1997.

[75] (W3C), W. W. W. C. *SOAP Version 1.2 Part 2: Adjuncts (Second Edition)* [online]. apr 2007. [cit. 29. 01. 2018]. Dostupné z: <https://www.w3.org/TR/soap/>.

[76] WANG, J. – NTARMOS, N. – TRIANTAFILLOU, P. *Indexing Query Graphs to Speedup Graph Query Processing* [online]. jul 2016. [cit. 27. 11. 2017]. Dostupné z: <https://openproceedings.org/2016/conf/edbt/paper-30.pdf>.

[77] WEBBER, J. – ROBINSON, I. *The Top 5 Use Cases of Graph Databases* [online]. feb 2017. [cit. 25. 11. 2017]. Dostupné z: <https://neo4j.com/resources/top-use-cases-graph-databases-white-paper/>.

[78] WIGGINS, A. *The Twelf-Factor App* [online]. jan 2017. [cit. 02. 02. 2018]. Dostupné z: <https://12factor.net/>.

[79] WILLIAMS, D. W. – HUAN, J. – WANG, W. 23rd International Conference on Data Engineering. In *Graph database indexing using structured graph decomposition*, s. 976–985, 2007.

[80] WINER, D. *XML-RPC Specification* [online]. jun 1999. [cit. 29. 01. 2018]. Dostupné z: <http://xmlrpc.scripting.com/spec.html>.

[81] WOODIE, A. *JanusGraph Picks Up Where TitanDB Left Off* [online]. jan 2017. [cit. 28. 11. 2017]. Dostupné z: <https://www.datanami.com/2017/01/13/janusgraph-picks-titandb-left-off/>.

[82] YAN, X. – YU, P. S. – HAN, J. ACM SIGMOD International Conference on Management of Data. In *Graph indexing: A frequent structure-based approach*, s. 335–346, 2004.

[83] YAN, X. – YU, P. S. – HAN, J. ACM SIGMOD International Conference on Management of Data. In *Substructure similarity search in graph databases*, s. 766–777, 2005.

[84] YOO, A. et al. *A scalable distributed parallel breadth-first search algorithm on BlueGene/L* [online]. may 2005. [cit. 26. 11. 2017]. Dostupné z: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.7691&rep=rep1&type=pdf>.

[85] ZHANG, S. – HU, M. – YANG, J. 23rd International Conference on Data Engineering. In *TreePi: A novel graph indexing method*, s. 966–975, 2007.

[86] ZHANG, S. et al. 12th International Conference on Extending Database Technology. In *A novel approach for efficient supergraph query processing on graph databases*, s. 204–215, 2009.

[87] ZIMMERMAN, H. OSI reference model — The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*. apr 1980, s. 425–432.

Příloha A

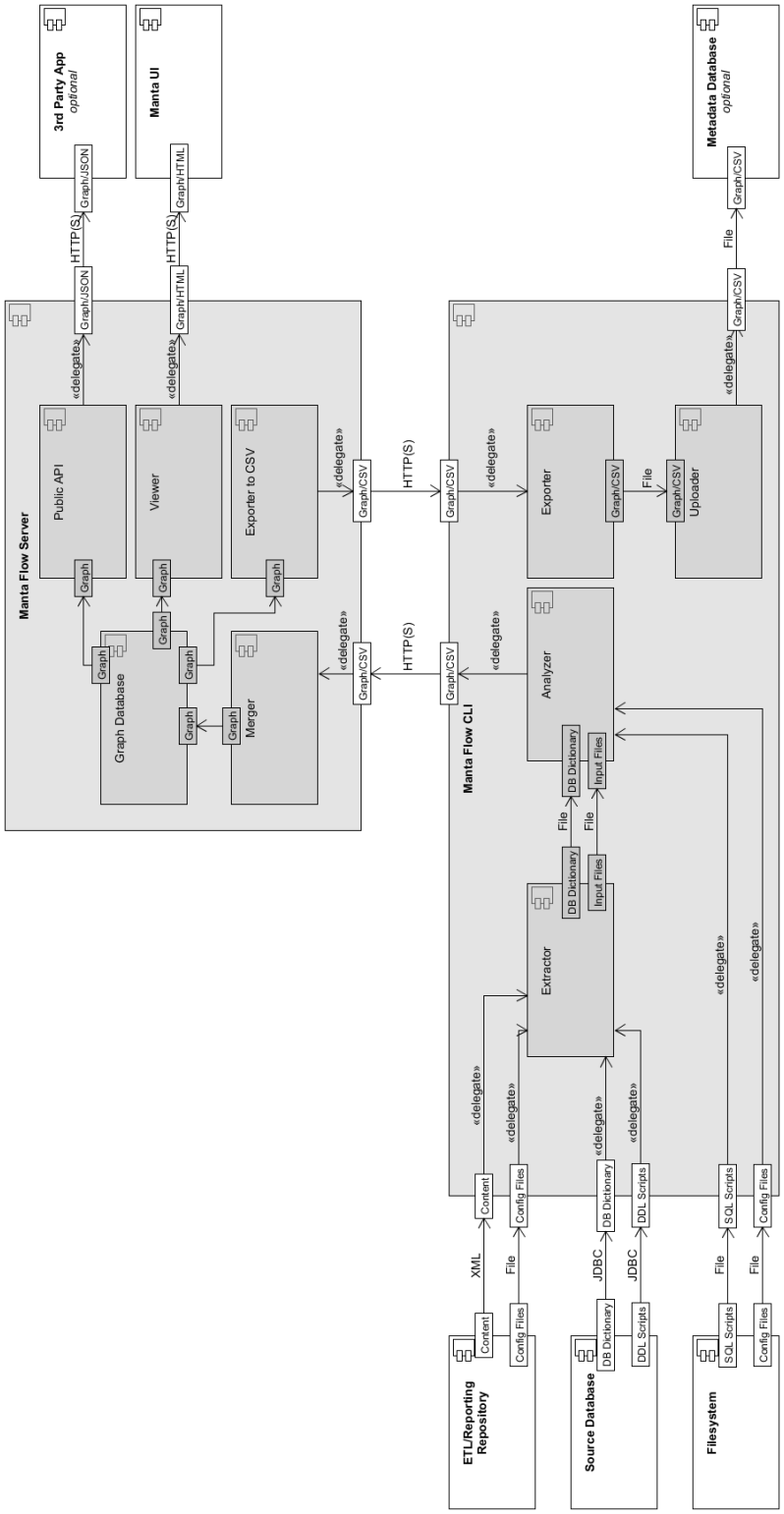
Seznam zkratek

ACID - *Atomicity, Consistency, Isolation, Durability*
API - *Application Programming Interface*
BASE - *Basically Available, Soft state, Eventual consistency*
BFS - *Breadth First Search*
CRUD - *Create, Read, Update, Delete*
CSV - *Comma Separated Variable*
DDL - *Data Definition Language*
DFS - *Depth First Seacrh*
DSL - *Domain Specific Language*
DTM - *Declarative Transaction Model*
ETL - *Extract, Transform, Load*
FaaS - *Functino as a Service*
FTP - *File Transfer Protocol*
GUI - *Graphical User Interface*
HTTP - *Hypertext Transfer Protocol*
HTTPS - *Hypertext Transfer Protocol Secure*
IaaS - *Infrastructure as a Service*
JSON - *JavaScript Object Notation*
JVM - *Java Virtual Machine*
LOC - *Lines of Code*
MVCC - *Multi-Version Concurrency Control*
NoSQL - *Not only SQL*
OGM - *Object Graph Mapper*
OLAP - *Online Analytical Processing*
ORM - *Object Relational Mapper*
PaaS - *Platform as a Service*
PoC - *Proof of Concept*

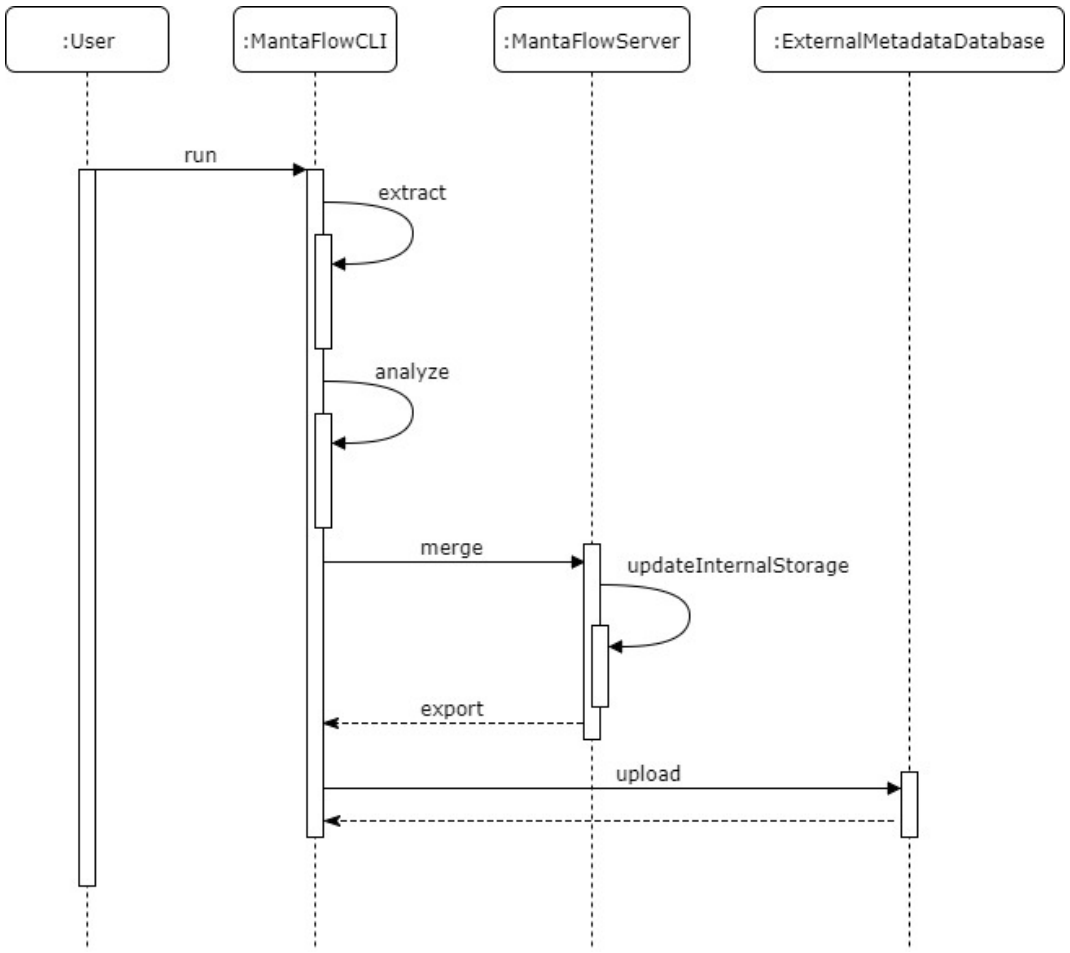
POJO - *Plain Old Java Object*
PTM - *Programatic Transaction Model*
RDBMS - *Relational Database Management System*
REST - *Representational State Transfer*
RPC - *Remote Procedure Call*
SaaP - *Software as a Product*
SaaS - *Software as a Service*
SOA - *Service Oriented Architectures*
SOAP - *Simple Object Access Protocol*
SPI - *Service Provider Interface*
SQL - *Structured Query Language*
URI - *Uniform Resource Identifier*
WSDL - *Web Services Description Language*
XML - *Extensible Markup Language*

Příloha B

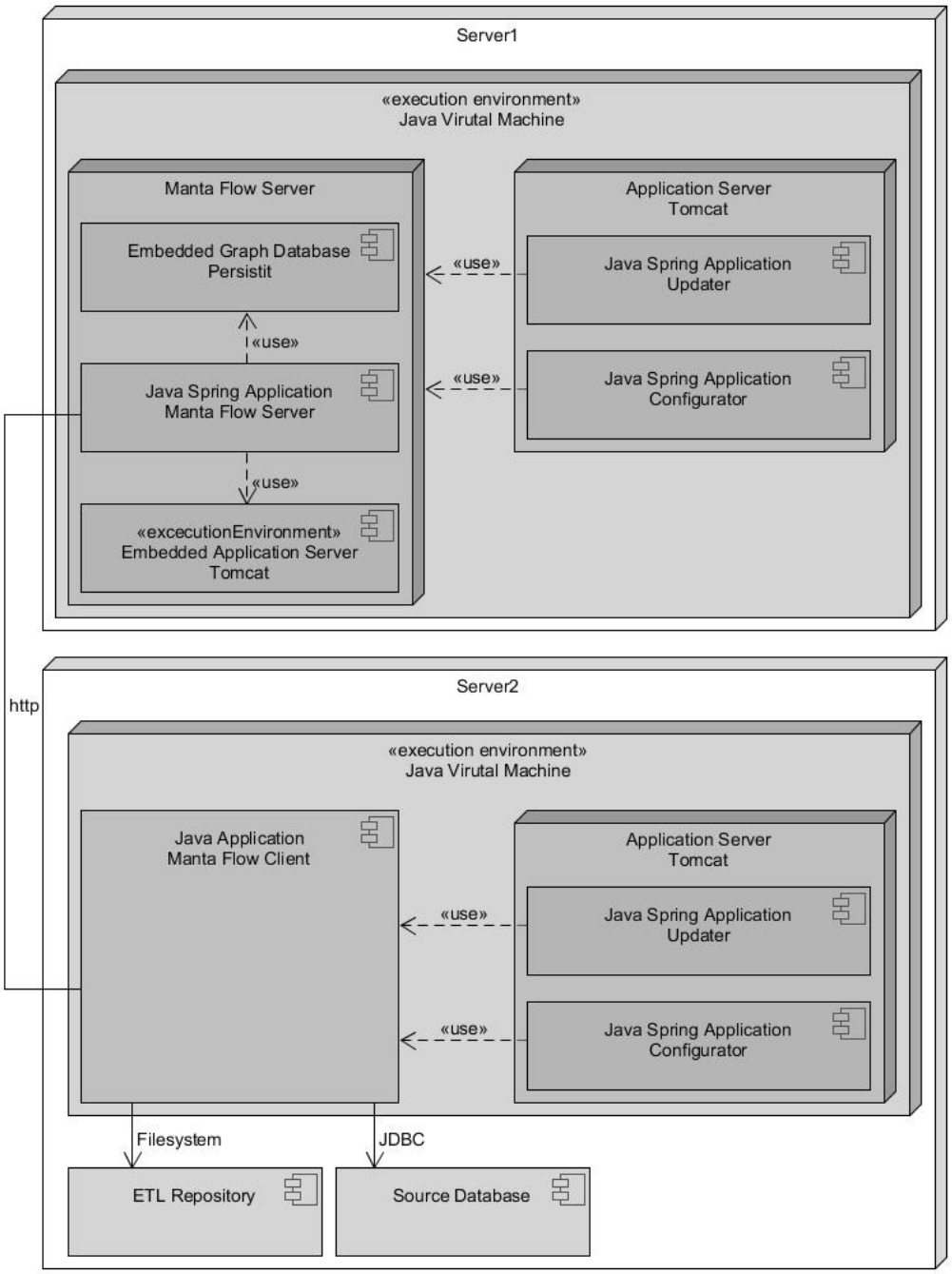
Diagramy



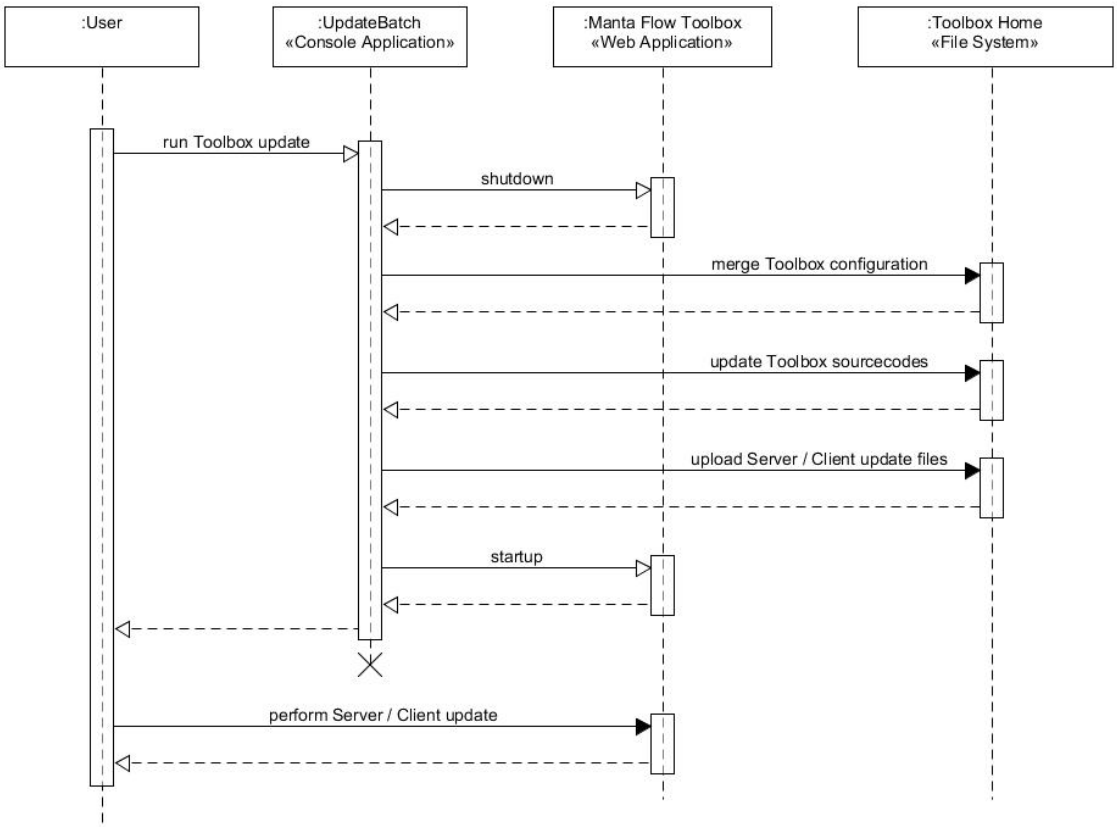
Obrázek B.1: Stávající architektura *Manta Flow*



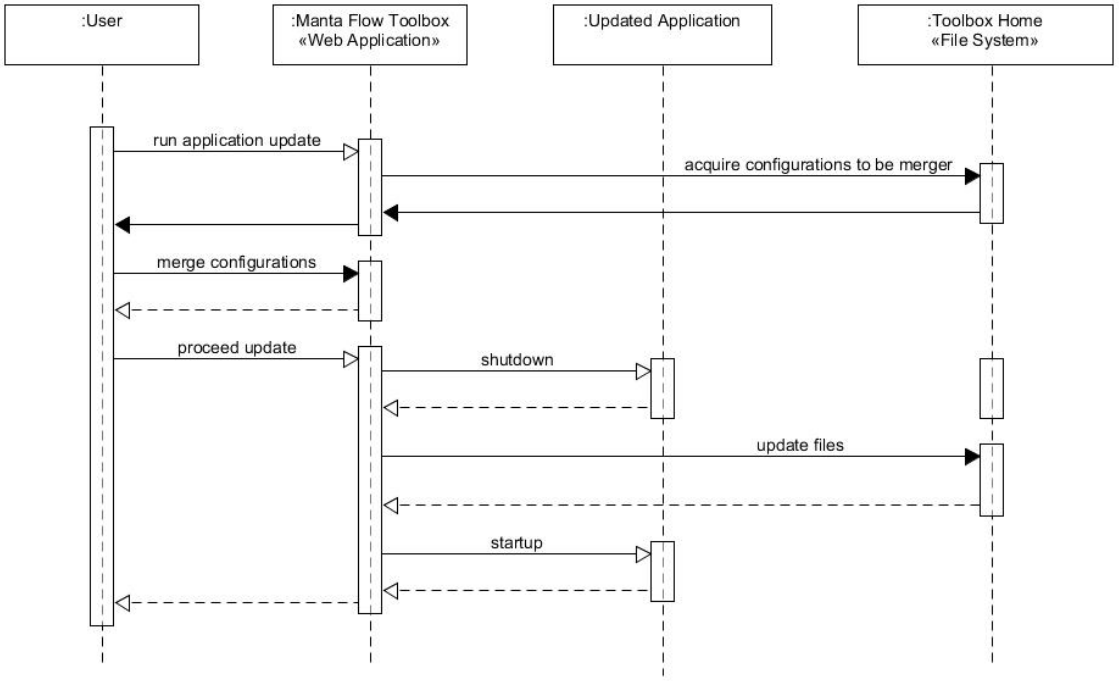
Obrázek B.2: Interakce mezi *klientskou* a *serverovou* částí *Manta Flow*



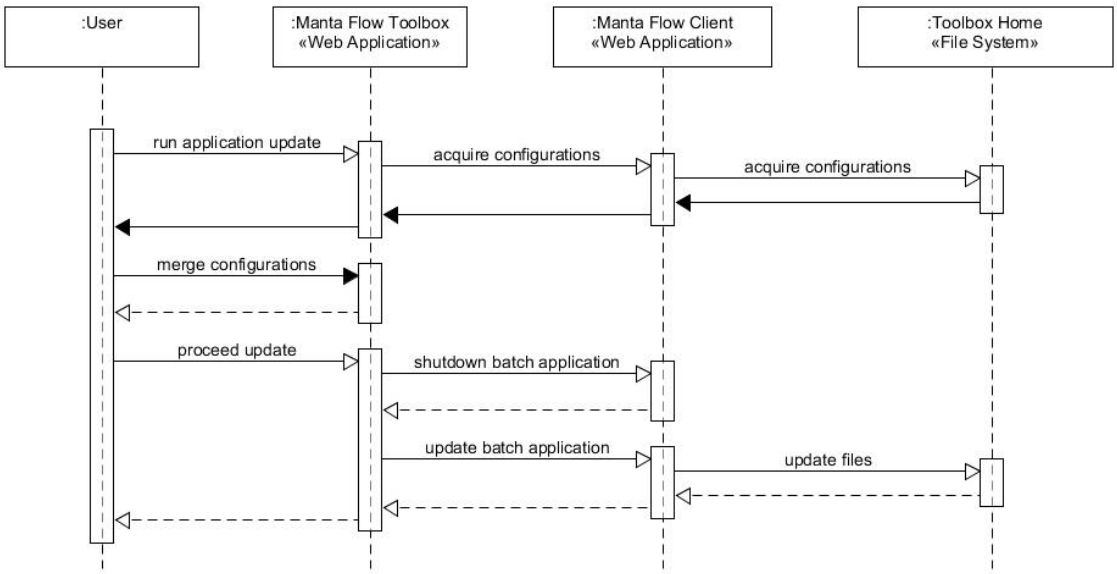
Obrázek B.3: Aktuální orchestrace aplikací *Manta Flow Server, Client, Updater a Configurator*



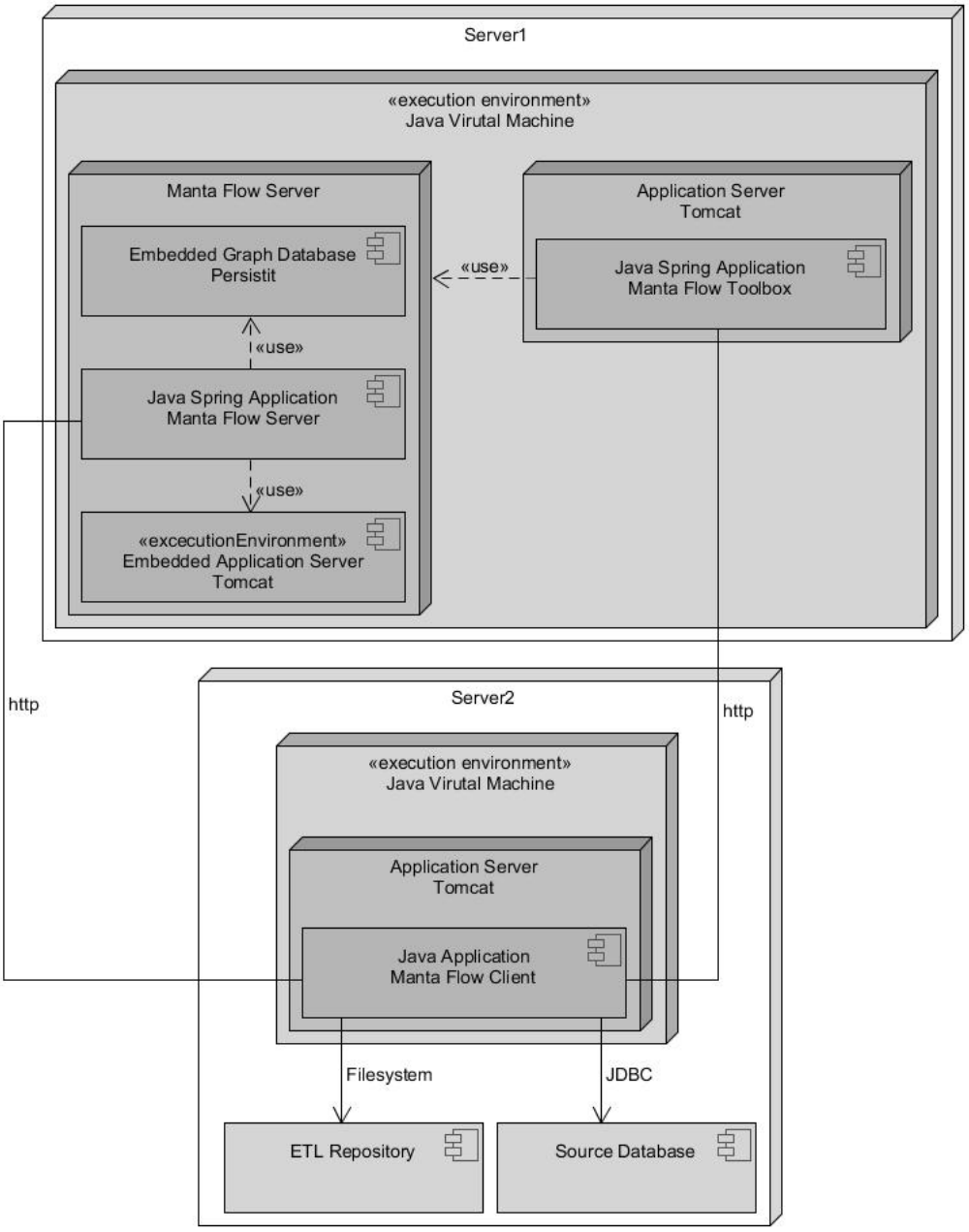
Obrázek B.4: První krok aktualizace všech komponent - aktualizace *Manta Flow Toolbox*



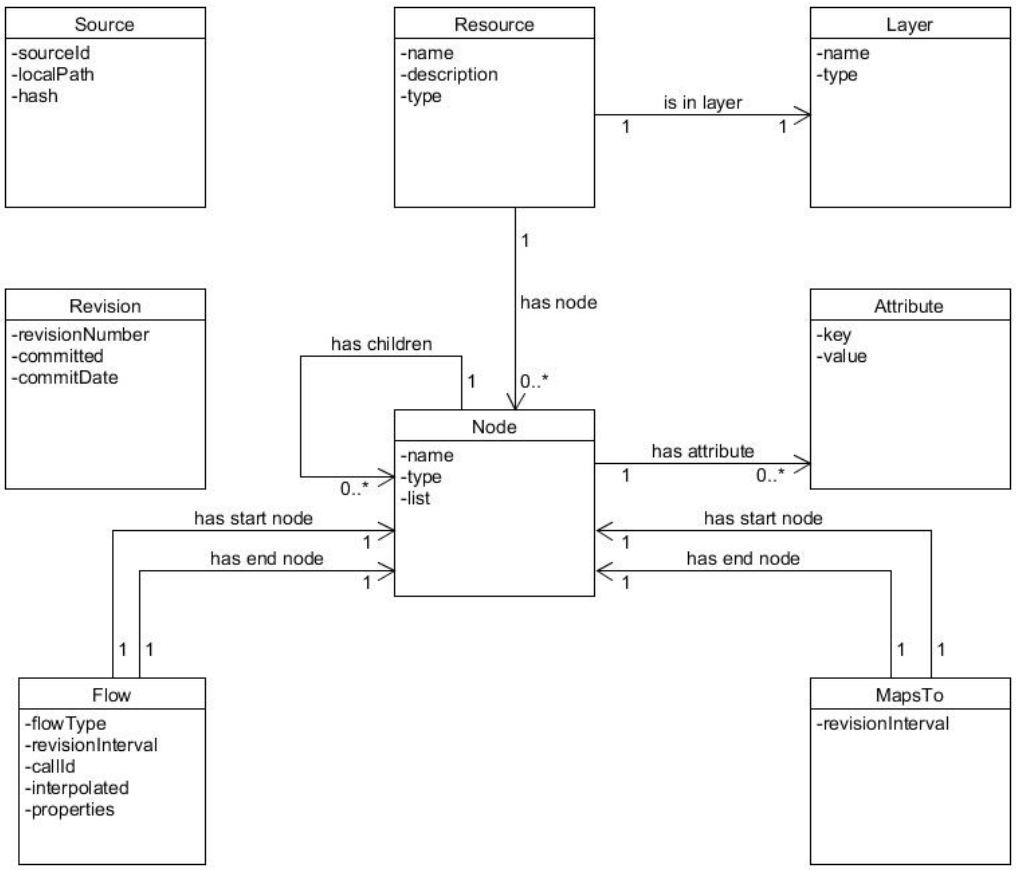
Obrázek B.5: Standardní aktualizace komponent na jednom zařízení



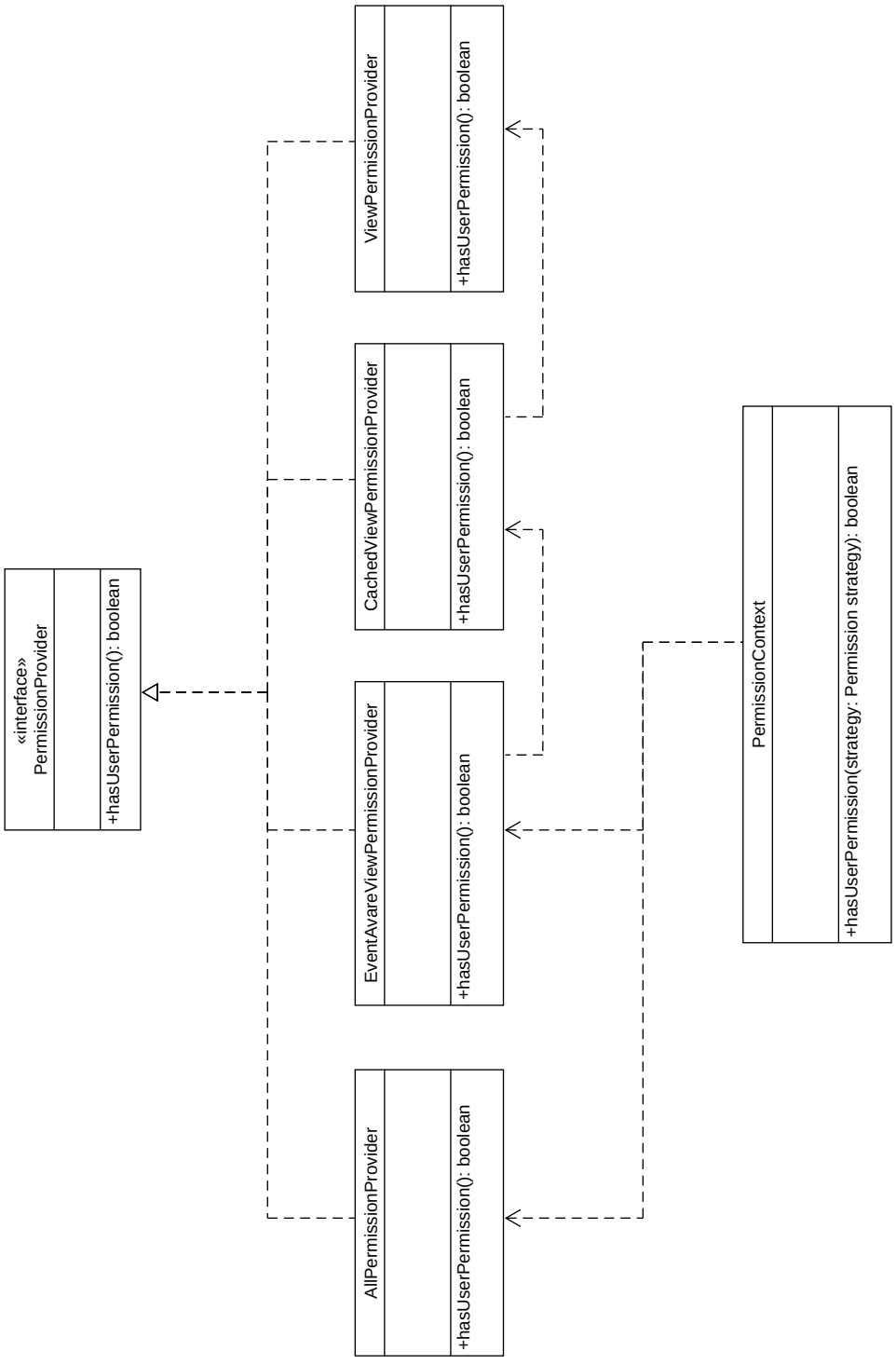
Obrázek B.6: Aktualizace komponenty *Manta Flow Client* přes *HTTPS*



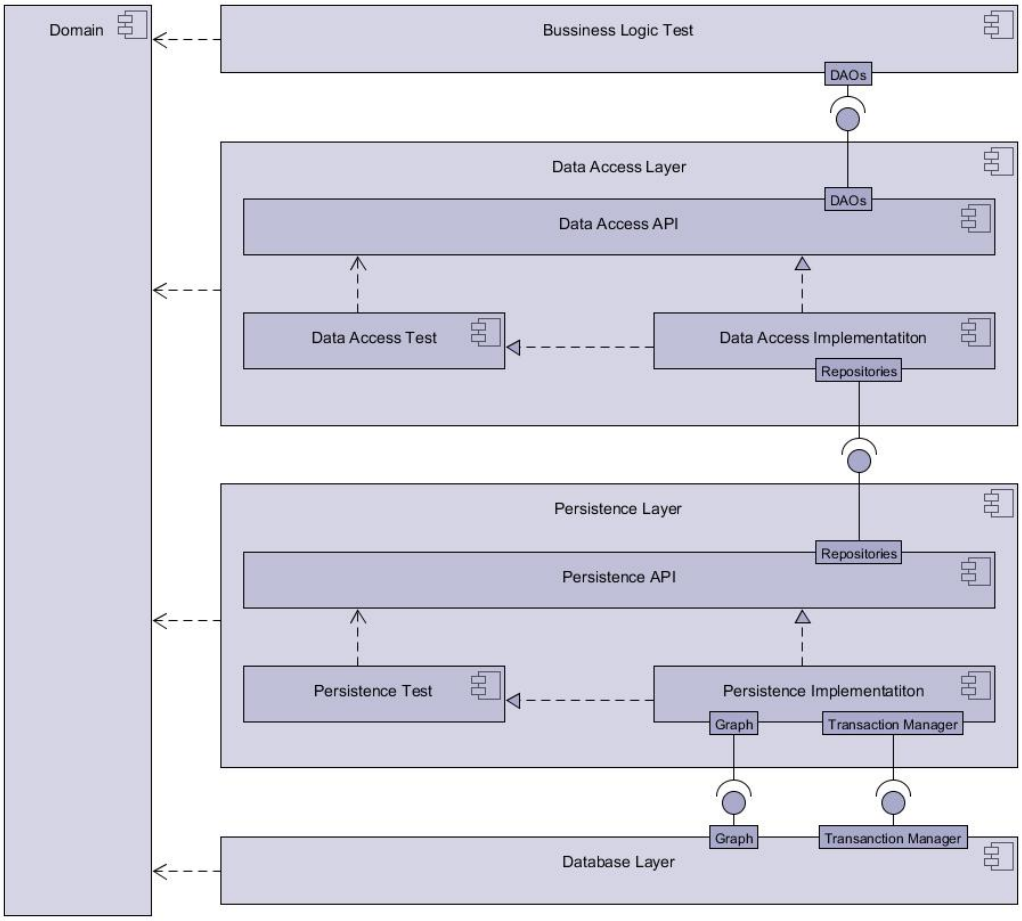
Obrázek B.7: Upravená orchestrace aplikací *Manta Flow Server, Client a Toolbox*



Obrázek B.8: Doménový model



Obrázek B.9: Implementace kontroly oprávnění



Obrázek B.10: Diagram komponent prototypové implementace

Příloha C

Obsah přiloženého CD

TODO - JavaDoc API (TODO) - PoCs (TODO)