ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Optimální struktura a indexy modelu metadatového úložiště v grafové databázi
Student:	Bc. Michal Peroutka
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2016/17

Pokyny pro vypracování

Na rozdíl od relačních databází není v grafových databázích ustálený převod mezi logickým a fyzickým modelem. Stejně tak grafové databáze poskytují nové druhy indexů, které mohou výrazně zrychlit některé dotazy. Produkt Manta Tools společnosti Profinit využívá právě grafové databáze pro své metadatové úložiště. Cílem práce je provést rešerši přístupů k implementaci fyzických modelů a využití indexů v grafových databázích, vytvořit prototypy fyzického modelu s indexy pro definovaný logický model a zadané dotazy, provést výkonové testy na dodaných datech a nejlepší řešení implementovat.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 8. září 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Optimální struktura a indexy modelu metadatového úložiště v grafové databázi

Bc. Michal Peroutka

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

8. května 2016

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu diplomové práce Ing. Michalu Valentovi, Ph.D. za jeho rady a vedení při návrhu řešení. Mnoho díků patří také lidem z projektu Manta, kteří mi dali dobré rady při implementaci a testování. Za všechnu podporu bych ještě rád poděkoval své rodině.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. 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 8. května 2016 .....

Odkaz na tuto práci

Peroutka, Michal. *Optimální struktura a indexy modelu metadatového úložiště v grafové databázi.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Práce se zabývá optimálním uložením různých hierarchií dat do grafové databáze. Zkoumá se zde definice ekvivalentních uzlů a jejich hledání v datech.

Klíčová slova Manta, grafová databáze, slučování hierarchických modelů, TitanDB, Java

Abstract

The work deals with optimally storing different hierarchies of data in a graph database. The work examines definition of equivalent nodes and their search.

Keywords Manta, graph database, merging hierarchy models, TitanDB, Java

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V průběhu posledních dvou desetiletí v enterprise aplikacích vzniklo velké množství systémů, které spolu komunikují a sdílí data. U velkých korporací může jít o desítky či stovky aplikací, mezi kterými probíhá komunikace. V posledním desetiletí narůstá obliba datových skladů. Ty sjednocují nejdůležitější data v celé společnosti. Struktura datových toků u takto velkých systémů nebývá triviálně zjistitelná. To může mít za následek ztrátu sémantické informace o datech, které přitékají do skladu. V horším případě může jít i o ztráty informací mezi jednotlivými systémy.

Produkt Manta Flow je nástroj na analýzu a zobrazení vývoje datových struktur databází a toků dat. Zároveň poskytuje pohled na tyto toky v čase. Tím umožnuje najít historický pohled na tok dat a snadnější zhodnocení, zda nedošlo k chybě. Manta k analýze využívá DDL skripty databází, definice reportů a ETL (exctract, transform, load). Tato vizualizace pomáhá pochopit složité datové toky mezi různými technologiemi a systémy. To pomáhá například při udržování a vytváření datových skladů. Podporované technologie jsou Teradata, Informatica, Oracle a Cognos. Celkový model je možné verzovat a díky tomu zpětně nahlížet na změny provedené v toku a definici dat. Tím lze odhalit změnu sémantického významu dat v čase. Verze tedy pomáhají při rozhodování o případných opravách.

Kapitola 1

Cíl práce

Současný datový model je vytvořen nad hierarchií (stromem) objektů podle stavu fyzického umístění. Uzel reprezentující sloupeček má v grafu za rodiče tabulku a ta má jako rodiče schéma a to zase databázi. Požadavkem je vytvořit takovou strukturu databáze, aby bylo možné mít více hierarchií. Více typů dat a více možných pohledů. Tyto pohledy mohou být založeny dle logického modelu, zodpovědných systémů nebo mohou být jakkoliv jinak definovatelné. Databáze musí být navrhnuta tak, aby bylo možné objekty stejného významu z různých hierarchií použít k přechodu mezi hierarchiemi. Shodné uzly budou hledány automaticky bez uživatelských zásahů.

Nynější model databáze se vyznačuje velice volnou definicí reprezentace stromu uzlů. Mimo kořenových uzlů jednotlivých stromů uzel o sobě neví, zda reprezentuje tabulku, sloupec nebo něco jiného. Tento model je flexibilní vůči změnám. Nevýhoda je zjevná - pokud uzel neví jakého je typu, tak nepomáhá k určení jeho ekvivalence s uzlem v jiné hierarchii. Databáze Manty v současné době neběží distribuovaně, ale při jakékoliv změně takového rozsahu je potřeba myslet na budoucnost. Je totiž pravděpodobné, že dříve či později bude Manta nasazena s distribuovanou databází. Když naposledy proběhla změna modelu přidáním revizí (verzí), tak se přidávání modelu dat zpomalilo na dvojnásobek. Přidání verzí znamenalo přidání dvou atributů hran ke každé hraně. Přidávání dalších hierarchií způsobí přinejmenším stejné zpomalení. Cílem je však tento růst dostat na co nejmenší možné minimum.

Projekt Manta využívá grafovou databázi Titan. V projektu jsou provedeny jisté optimaliazace specifické pro tuto databázi a nemusí být platné pro jiné. Vývoj Titanu je zastaven a databáze jejími vývojáři nebude nadále vyvíjena. Protože grafové databáze jsou si méně podobné než je tomu mezi relačními, a změna na model paralelních hierarchií je spíše většího rozsahu, tak je třeba nejdříve zhodnotit možnosti přechodu na jinou databázi.

Požadavky jsou:

1. Zhodnotit možnosti změny databáze vzhledem k současné situaci.
2. Analyzovat datový model projektu Manta.
3. Implementovat paralelní hierarchie.
   a) Rozlišit hierarchie.
      i. Tak, aby se mezi nimi dalo přecházet.
      ii. Zhodnotit možnost rozdílných revizí pro různé hierarchie.
   b) Musí se jednat o obecné řešení založené na heuristice.
   c) S co nejmenším zpomalením.
   d) Co nejvíce datově úsporné.
4. Myslet na potencionální distribuovatelnost.
5. Nastavit metriky měření rychlosti a provést testy.
6. Zhodnotit výsledky a určit možnosti dalšího pokračování.

Kapitola 2

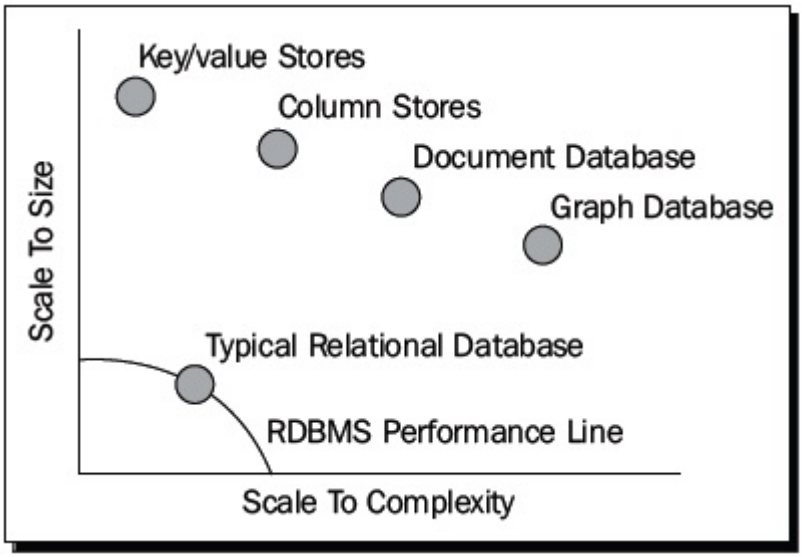
Seznámení s problematikou

2.1 Typy databází

Od začátku 80. let jsou relační databáze hlavní technologií pro ukládání aplikačních dat. Jedná se o ukládání dat do tabulek, kde jednotlivé záznamy jsou v řádcích. Sloupečky určují strukturu typovou a sémantickou. Tabulky se zpravidla spojují pomocí primárních klíčů. SQL dotazy jsou založeny na matematickém základu z relační algebry. Je zde tedy projekce, sjednocení, kartézský součin a jiné. Dotazovací jazyk SQL umožnuje mimo těchto základních operací i různé agregační a transformační funkce. Databáze umožnují jisté optimalizace založené především na indexaci.

Při tvorbě datových modelů relační databáze se mají vývojáři řídit podle normální formy. Normálních forem je šest a jsou značené 0NF až 5NF. Jednotlivé normy jsou více a více striktní a při jejich aplikaci dochází k rozpadu na větší množství tabulek. V praxi se využívá 3NF, která neobsahuje duplicitní data a přístup k datům je zajištěn cizími klíči a spojováním tabulek pomocí příkazu JOIN. Tento přístup je šetrný na množství dat, neobsahuje duplicity a umožňuje měnit uložené hodnoty. U složitějších databází může vznikat velké množství joinů v jednom dotazu. V reálu je možné se setkat s vývojáři, kteří zkoušejí pohledy na data i s několika desítkami joinů. Jejich databáze tak sice splňuje třetí normální formu a neobsahuje duplicity, ale je značně neefektivní co do výkonu. Takovýto přístup není možný v databázích s pár desítkami tisíc záznamů na hlavních tabulkách, natož kdyby se jednalo o databáze středně velkých podniků.

Množství ukládaných dat začalo s příchodem internetu více vyžadovat rozdělení dat až do několika desítek tisíc strojů. Jak se ukázalo, tak joiny nad velkým množstvím dat nejsou efektivní. Zároveň množství dat začalo vynucovat distribuci databází na více fyzických strojů. Jakýkoliv join mezi dvěma tabulkami, které jsou rozdělené na více strojů, znamená přenesení všech těchto dat po síti. Z toho plyne jediné - spojování distribuovaných tabulek je značně neefektivní. Řešením tohoto problému jsou NoSQL databáze. Ty porušují třetí



Obrázek 2.1: Graf škálovatelnosti databází

normální formu a ukládají duplicitní data do svých záznamů. Join je v NoSQL chápán jako anti-pattern a vývojáři by se ho neměli dopouštět. NoSQL dává přednost výkonu nad velikostí uložiště. Nejenže ukládá do svých záznamů duplicity, ale navíc i pro optimalizaci rychlosti editace, neediuje data. Místo změny dat jen přidá nové, s jiným časovým záznamem (číslem verze). Příkladem databáze, která je umístěna na desetitisících strojů v jednom clusteru je například Cassandra. [1] Jiným příkladem je část databáze Facebooku pro vyhledávání a emaily, která již v roce 2012 byla větší než 70TB a obsahovala více než sto miliónů emailových schránek. NoSQL se dělí na Key/value, Column a Document store.

Posledním typem NoSQL, který se používá pro speciální případ hustě provázaných „vztahových“ dat, jsou grafové databáze. Ty vynikají v dotazech typu „všichni kamarádi mých kamarádů“ před dotazy „všechny objednávky, které mají hodnotu větší než 10.000 s dodavatelem ze Středočeského kraje“ jak bývá zvykem u klasicky relačních dat. Tato data jsou logicky chápána jako obecný graf vztahů. Modelovým příkladem využití těchto databází je Facebook s jeho vyhledáváním všech „přátel mých přátel“, „akcí, které by se uživateli mohly líbit, protože se líbí X jeho přátelům“ a jiné. Grafové databáze a jejich snadné traverzování mezi uzly jsou v reálu jen příjemný a optimalizovaný interface nad jiným typem NoSQL databází. Již zmíněný Facebook například používá kombinaci relační a Wide-column store databáze. Na obrázku 2.1 je názorný graf škálovatelnosti databází co do složitosti a velikosti.

Poučení z tohoto obrázku není o tom, že relační databáze jsou nepoužitelné. Tento obrázek jen ukazuje, že nejsou dobře škálovatelné. Relační da-

tabáze jsou a budou převládat ve středně velkých systémech a až na speciální případy poskytují vše potřebné. Jedním ze speciálních případů je projekt Manta, kde datové toky a přirozeně stromová reprezentace dat nutí k použití grafové databáze. Většina databází v budoucnu zůstane relačních, avšak jakékoliv distribuované databáze větších rozměrů budou nutit vývojáře použít jeden ze čtyř typů NoSQL databází.

2.2 Grafové databáze

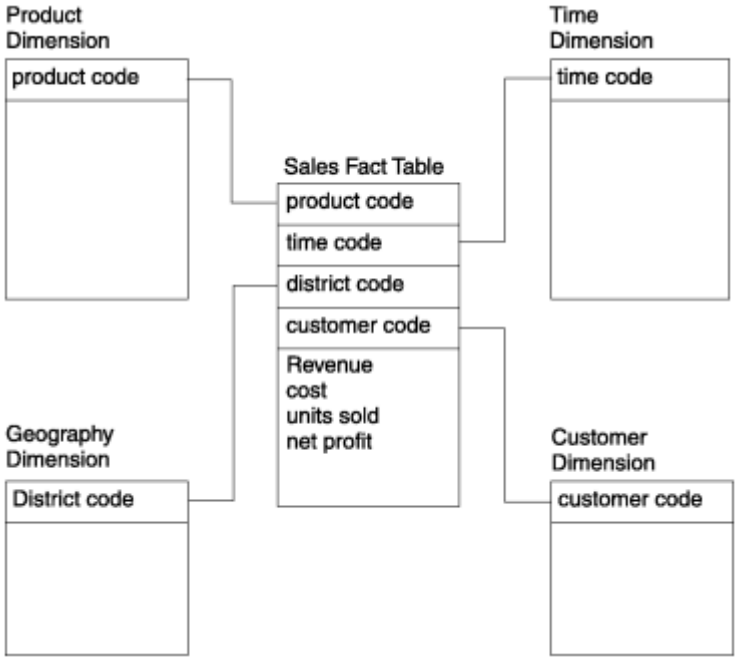
Grafové databáze jsou založeny na teorii grafů. Jedná se o reprezentaci dat jakožto graf vztahů mezi entitami. Jednotlivá data jsou uložena v uzlech, které představují entity. Entity jsou spolu spojeny hranami, které mají svůj label, který označuje vztah mezi těmito entitami. Hrany mohou být jednosměrné, obousměrné nebo bez určeného směru. Z jednoho uzlu může vycházet více hran stejného typu nebo se dá definovat jedna unikátní hrana. Uzly mohou obsahovat atributy (data) typu key-value tj. klíč => hodnota. Klíče mohou být indexované. Indexované klíče se mohou nastavit jako unikátní, ale nemusí tomu tak být. Hrany také mohou obsahovat key-value data. Ne každá grafová databáze má takto definovaná pravidla. Takto je to definované v Titanu.

Stejně tak, jak se ne všechna data ideálně reprezentují v klasické relační databázi, tak to samé se dá říci o grafové databázi. Práce s typicky relačními daty by nemusela být příjemná (např. diskusní příspěvky). Tento problém má dvě řešení: rozdělení modelu na více databází anebo použití multi-modelové databáze [2]. Mezi zástupce multi-modelové databáze patří FoundationDB, OrientDB a ArangoDB. Tyto tři databáze jsou dva až pět let staré a tudíž se nejedná o rozšířené databáze. Idea multi-modelové databáze je však velmi zajímavá do budoucna. Je otázkou, zda se jedná o slepou cestu vývoje, nebo ne. Příkladem technologie, která se z různých důvodů v praxi spíše neujala, je objektově relační databáze (například od firmy Oracle).

Relevanci multi-modelové cesty vývoje databází podporuje i koupě [3] Aureliusu (tvůrci databáze Titan) firmou DataStax, která stojí za komerční verzí Cassandry. Touto koupí nezískala firma DataStax databázi Titan, ale její vývojářský tým. S tímto vývojářským týmem chce přidat multi-model do Cassandry a jejího komerčího nasazení DataStax Enterprise [4].

Dalším kladem může být statistika [5], dle které přibližně půlka všech grafových databází je vlastně multi-modelová. Takže pro tuto myšlenku je nadchnuta velká část nových výrobců grafových databází.

Pevně věřím, že rozdělení modelu nebo multi-modelové databáze má budoucnost jak v běžných, tak i v distribuovaných databázích. Tento model by mohl v budoucnu tvořit významnější část databází, než je tomu doposud a to i ve středně velkých a nedistribuovaných systémech. Zvláště pokud k tomu bude vyvinuté příslušné API, které umožní plynulé dotazování mezi rozdělenými modely.



Obrázek 2.2: Faktová tabulka Sales spojená s dimenzionálními tabulkami

2.3 Rozdělení databáze v clusteru

Horizontální rozdělení databáze do více serverů se odborně nazývá shard, což v češtině znamená střep. Proces rozdělení se nazývá sharding. Každý server má svoji část databáze, svůj střep, a zároveň může mít i společnou část, kterou má každý server.

Příkladem společné části je třeba dimenzionální tabulka v datových skladech. Dimenzionální tabulka obsahuje data, pomocí kterých se nahlíží na faktovou tabulku. Zjednodušeně řečeno se jedná o atributy (databázové klíče), podle kterých je sestavena analytická faktová tabulka. Faktová tabulka obsahuje analyticky spočítaná data pro nějakou sadu atributů. Například čas, místo, zákazník, produkt aj. Schéma je například na obrázku 2.2 [6].

Množství dat ve faktových tabulkách je tedy násobek velikostí všech dimenzí. Protože JOIN je anti-pattern v rozdělených databázích z důvodu nutnosti velkého přenosu dat, tak se dimenzionální tabulky distribuují do všech serverů v clusteru. Tím se při nutném joinu předejde pomalému vykonání dotazu a výsledek dotazu nad faktovou tabulkou je vlastně jen sjednocení nad lokálními provedeními dotazu s lokálním joinem.

Tato teorie platí pro relační databáze. NoSQL tento problém obchází tím, že místo relačního rozdělení na více tabulek a provádění JOINu nad nimi, existuje jen jedna tabulka, která obsahuje duplicitní záznamy. V grafových databázích je sharding považován za anti-pattern a je velice složitý. A to z dů-

vodu traverzování mezi uzly. Pokud se například uzel ptá na všechny sousedy jeho sousedů, tak dotaz může při jednotlivém traverzování požadovat informace z mnoha serverů. Obecně existují dva přístupy k nutnosti rozdělené databáze. První je náhodný. Ten zajišťuje rovnoměrné rozdělení grafu po serverech co do dat. Zároveň se zde databáze samy snaží shlukovat uzly, které vedle sebe sousedí. Při přidávání dalších hran do grafu se tato snaha kazí. Druhý přístup je „user directed“ neboli uživatelem rozhodnuté rozdělení. To může být vhodné pokud je vývojář schopen správně určit části modelu grafu, které se objevují ve společných dotazech. Na příkladu sociální sítě to může být například nějaký větší uzemní celek jako společné bydliště.

Kapitola 3

Databázový engine projektu Manta

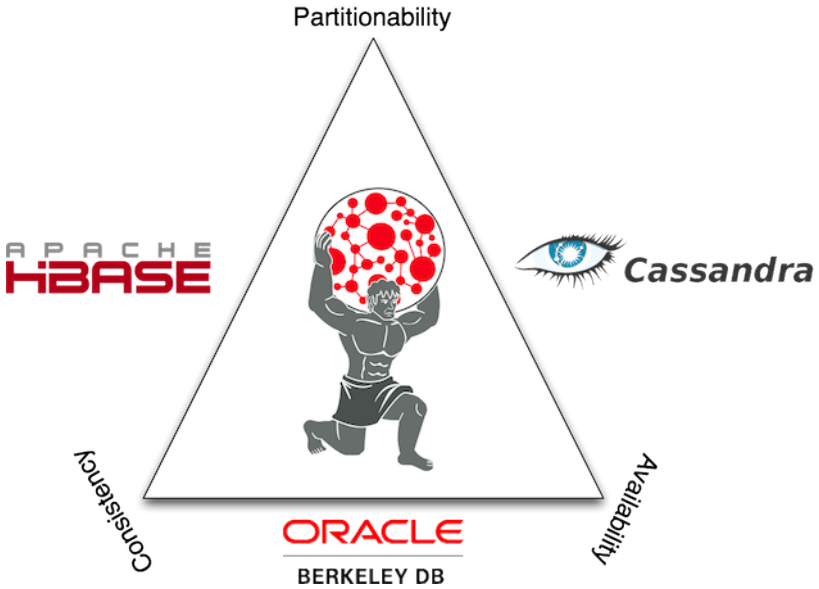
3.1 Vývoj volby databázového stroje

Projekt Manta používá k uložení dat grafovou databázi Titan. Titan je specifická databáze, která jako svoje interní uložiště používá jinou databázi. Tím se tedy jedná spíše o optimalizovaný interface pro dotazování. Manta používá verzi 0.4, která poskytuje na výběr storage backendu Persistit, Cassandru a Berkley DB.

Z těchto tří je Persistit pro projekt jasnou volbou. Berkley je placená databáze a Cassandra není tak rychlá pro malé množství dat. Persistit byl také doporučován autory Titanu pro malé až středně velké grafové databáze.[7] V relacích Titanu se za menší databázi považuje databáze do sta milionů uzlů. Na datech popisujících toky dat ve velkých českých bankách a komunikačních společnostech se ukazuje tato hranice jako dostatečná. V případě potřeby větší databáze se dá přejít na Cassandru.

Do budoucna má Titan pro projekt dva problémy. Persitit od verze 0.5 není podporován a Titan verze 0.4 není vůbec podporován. Tento problém je teoreticky řešitelný. To, že je Cassandra ve verzi 0.4 pomalejší pro malá data však nemusí znamenat, že je pomalá i v novějších verzích Titanu. Přechod na novější verzi by tedy mohl být řešením. Druhý, zato závažnější problém je, že Titan již není dále vyvíjen. Firma Aurelius, která stojí za touto databází, byla odkoupena výrobcem Cassandry a vývojářský tým se teď věnuje jinému projektu.

Z dlouhodobého hlediska je tedy velice pravděpodobné, že se databáze bude muset vyměnit kvůli vzrůstající komplexitě toho, co Manta umí. Množství změn v databázovém modelu, které změna na paralelní hierarchie vyžaduje je vhodná příležitost ke zvážení, zda by byla možnost tuto změnu provést zároveň. Grafové databáze se od sebe mohou značně lišit. Mají různě definované indexy nebo postrádají hranové indexy, uzlové indexy, vlastnosti



Obrázek 3.1: CAP theorém v Titanu

(property) a typy (labely). Jakýkoliv pokus o optimalizaci je z těchto důvodů také vždy databázově specifický. Proto je potřeba nejdříve zhodnotit možnosti ostatních grafových databází.

3.2 Titan

Jak již bylo zmíněno, Titan je velice specifická databáze. Díky tomu, že nemá svoje vlastní datové uložiště (storage backend), tak je možné si vybrat z různých backendů, které mají různé vlastnosti. Výhodou je, že přechod mezi nimi je vlastně zadarmo. Není potřeba upravovat kód aplikace a ani dotazování. Jediné v čem se z pohledu Titanu liší, je jejich nastavení. V poslední verzi nabízí tři možnosti uložišť - Berkley, HBase a Cassandra. Každá z těchto databází má své specifické nastavení v rámci Titanu, kterým je ovlivněna výkonnost databáze. Například u Cassandry je možné nastavit komprimaci dat. Komprimace může zpomalit vkládání, ale může zrychlit dotazování nad velkým množstvím dat zvláště pokud bude databáze rozdělená na více serverů v clusteru. Zároveň každá z těchto tří databází spadá do jiné části CAP teorému viz 3.1 [8]. Umístění v CAP teorému implikuje jiné základní vlastnosti jednotlivých backendů.

Z možných databází je Berkley placené a je vhodné pro jeden server. Její výhodou je, že se jedná o embedded databázi. Doporučený rozsah velikosti grafu je mezi deseti až sto miliony uzlů. Předností této databáze je rychlost, která je způsobená tím, že tato backend databáze i Titan, běží pod stejnou JVM.

Další dvě jsou vhodné pro obrovské distribuované grafové databáze, které nebudou mít problém s horizontálním škálováním. Obě jsou schopné pracovat v clusteru a poskytují možnost použití databáze o statisíci bilionů hran [9]. Co se týče operací, tak zápis je stejně dobrý jak čtení - převzatá vlastnost Cassandry/HBase. Výhodou Cassandry nad Hbase je, že umožnuje i embedded běh pod stejnou JVM. HBase nabízí jen připojení k lokálnímu serveru nebo vzdálenému serveru. Obě varianty vyžadují komunikaci pomocí socketů. Vzhledem k možnosti embedded varianty a většímu množství dokumentace ke Cassandře bych z těchto dvou databází považoval Cassandru za lepší volbu. Z pohledu potřeb Manty je Cassandra na tom lépe i z pohledu CAP teorému, kvůli tomu, jak jsou data vkládána najednou a jediný prováděný update je změna čísla poslední revize. Ostatní dotazy jsou vkládání nebo dotazování.

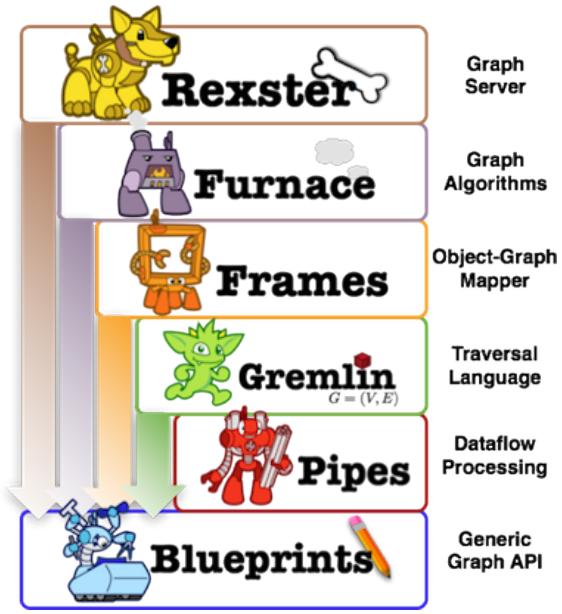
Obecně správná myšlenka Titanu je rozdělení do separátních vrstev. Ty umožnují například měnit storage backend a tím vlastně využívat cizí technologie, měnit cílovou výkonnost a rozsah databáze za běhu projektu. Zajímavou vrstvou je možnost externího indexovacího frameworku. Zde se nabízí Lucene, Solr a Elasticsearche. Lucene je nedistribuovatelný předchůdce Solr a oba projekty vychází pod stejnou Apache skupinou a mají i stejná data nových verzí. Ve skutečnosti je Solr jen nový framework s vylepšeným API. Je to distribuovatelný následník Lucene, který využívá jeho jádro na pozadí svého nového API. Elasticsearche je distribuovatelná konkurence Solr.

3.3 TinkerPop

Výhodou Titanu je propojení s celým TinkerPop Stackem. Ten obsahuje Rexter - grafový server, Furnance - grafové algoritmy, Frames - objektové mapování, Gremlin - dotazovací jazyk, Pipes - traverzování nad grafem, a celé je to zastřešeno generickou strukturou grafového API - Blueprints. Viz architektura Rexteru 3.2 z dokumentace [10]

Dalo by se říci, že se jedná o nezávislé moduly jedné technologie, která má více oddělených vrstev. Tato modulárně oddělená struktura je velmi podobná Titanu takže nepřekvapí, že se původně jednalo o projekt od tvůrců Titanu, aby měli čím dotazovat. Na rozdíl od Titanu se však TinkerPop stal součástí takzvaného Apache inkubátoru. Jeho vývoj si jde vlastní cestou i po skončení vývoje Titanu a snaží se definovat standardizovaný dotazovací jazyk pro grafové databáze. Vzhledem k tomu, že některé jiné databáze (například Arango) mají své implementace Blueprints, tak tato snaha se zdá být úspěšná.

Databáze jako taková, je pro projekt více než vhodná, ale jak již bylo zmíněno, tak její vývoj je zastaven a to z dlouhodobého pohledu opravy chyb a potřebných vylepšení není dobrý stav. V horizontu následujících pár let je však stále dostačující.



Obrázek 3.2: Architektura Rexteru

3.4 Neo4j

Neo4j je první a nejznámější zástupce grafových databází. Jeho nevýhodou je, že je částečně placený. Otázky vyvolává i horizontální škálovatelnost Neo4j pro velký počet serverů. [11] Tyto otázky jsou způsobené tím, že distribuovaný Neo4j využívá master-slave architekturu. Write bývá většinou pomalejší než operace read. Otázkou je, při jak velkém clusteru začne být tato architektura problémem. V kalkulacích kvality bych tuto vlastnost spíše považoval za riziko. I přes tyto výtky je Neo4j velmi dobrá DB. Jen se nehodí ke grafům, které mohou dosahovat až sto bilionů hran. [12] Pro projekt Manta by tato databáze byla vhodná, kdyby nebyla placená.

Databáze má svůj dotazovací jazyk zvaný Cypher. Lze však použít i dotazovací jazyk Gremlin, který je použit v Titanu. [13] Názory na porovnání Gremlinu a Cypheru jsou velice různorodé. [14] [15] [16] Osobně se mi více líbí syntaxe Gremlinu, která je, dle mého názoru, lépe čitelná. Pro představu dotaz v Cypher:

```
MATCH (start:Content)-[:RELATED_CONTENT]->(content:Content)
WHERE content.source = 'user'
OPTIONAL MATCH (content)-[r]-() DELETE r, content
```

Pro porovnání dotaz v Gremlinu

```
g.V().hasLabel('movie').as('a','b').
   where(inE('rated').count().is(gt(10))).
```

```
select('a','b').
    by('name').
    by(inE('rated').values('stars').mean()).
order().by(select('b'),decr).
limit(10)
```

3.5 OrientDB

OrientDB je jedna z prvních databází, která má implementovaný multi-model. Jako snad každá grafová databáze, i tato má svůj vlastní dotazovací jazyk. Na rozdíl od ostatních, je jazyk spíše nadstavbou SQL, což může být považováno za výhodu a samotná DB to vydává za svoji výhodu [17], [18].

```
SELECT name, out('ACTS').title FROM Person
WHERE name = 'Robin'
```

Výhodou je podpora TinkerPop Stacku, velkého množství programovacích jazyků a je OpenSource. Obchodní model má stejný jako všechny ostatní OpenSource databáze. Platí se zde za podporu. Otázkou je výkonnost. Z některých zátěžových a srovnávacích testů vychází nerozhodně s Neo4j, ale má mnohem lepší uživatelské rozhraní na správu a vývoj.[17] Protože se jedná o relativně novou databázi, tak je zde i více prostoru pro zlepšení pomalých dotazů. Ze srovnání grafových databází co do popularity, vychází jako druhá nejznámější, kdy v posledním roce předehnala Titan. Její meziroční nárůst je více jak 100. V celých číslech bodování je však nárůst menší jak u Neo4j. [19] Databáze se zdá být v trendu, avšak nepřináší rychlostně nic lepšího proti Titanu.

3.6 Projekt Trinity neboli Graph Engine

Trend grafových databází si uvědomují i ti největší softwaroví hráči. Microsoft Research v květnu 2015 představil první ukázku ze svého grafového projektu Trinity. Ten zároveň přejmenoval na Graph Engine. [20], [21] To, že je projekt stále pod Microsoft Research znamená, že ještě nemá vyhráno a Microsoft je jistě připraven ho zrušit ve prospěch již jiného grafového řešení. Stejně jak zrušil projekt Dryad [22]. Projekt sice není ve stádiu, kdy by byl použitelný, avšak má několik velice zajímavých vlastností. Vedle běžných slibů o distribuovatelnosti a optimalitě uložení dat, nabízí snadné nasazení v cloudu. Respektive v Azure, který je Microsoftem velmi propagovaný a zároveň patří v současné době k nejúspěšnějším divizím. Jeho velkou výhodou je také to, že zapadá do světa Microsoft technologií a že se v něm lze dotazovat pomocí LINQ dotazů, které patří mezi běžné při programování v C#. Tím odpadá jakákoliv nutnost pro vývojáře se učit něco nového. Zároveň jeho podpora bude obrovská

ze strany Microsoftu. Stejně tak, jak je v současné době těžko představitelný projekt v .NETu s něčím jiným než MSSQL, tak do budoucna je těžko představitelné, že by se v tomto prostředí prosadila jiná grafová databáze než ta od Microsoftu. Firma Microsoft byla vždy dobrá v držení si jednotné linie ekosystému technologií, které jsou spolu provázané a tím je velice omezený prostor pro jiné technologie.

Graph Engine je zajímavá databáze, protože umožnuje LINQ (defakto lambda funkce) dotazy a hlavně je ukázkou toho, že i velké společnosti se začaly zajímat o svět grafových databází. Z hlediska Microsoftu tedy nejde o to, udělat převratnou databázi. Jde jim spíše o to zůstat v kontaktu se současnými trendy a udržením jejich ekosystému. Pokud bude existovat API i pro Javu tak by tato databáze mohla být zajímavá pro projekt Manta.

3.7 Shrnutí

Ze srovnání grafových databází [19] je vidět, že všechny databáze, které nebyly zmíněné, mají jen malý zájem programátorské komunity. Z podrobného hledání lze zjistit, že většina grafových databází je inmemory nebo se snaží být vysoce distribuovatelná. V současné době projekt Manta potřebuje grafovou databázi, která je dobrá na jednom stroji a je zdarma. Výše zmíněné databáze jsou placené nebo jsou zaměřené na distribuovatelnost a nepodávají tak dobrý výkon jak Titan.

Nyní tedy nemá cenu měnit databázi. Moje doporučení pro projekt je počkat do doby, až Titan bude nevyhovující výkonem a v té době zvážit změnu API na Titan poslední verze i s tím, že již není dále vyvíjen nebo zvážit přechod na jinou DB a hledání začít od OrientDB. Ideální by bylo, kdyby tvůrci Titanu dříve než tento problém nastane vytvořili multi-modelovou databázi pro firmu Datastax. Ta poskytuje své databáze jako OpenSource s placenou podporou.[23]

Kapitola 4

Analýza a návrh

4.1 Výběr programovacího jazyka

Protože tato práce navazuje na již běžící projekt, tak žádná volba než Java verze 7 není možná. Z formálního hlediska je však vhodné zvážit možnosti výběru jazyka. Vedle osobních preferencí je dobré zmínit indexy. Existují různé indexy, které jsou založeny nad informacemi, které jsou shromážděné z dat od firem v anketách, propagačních materiálech, nabídkách, code review, cloud hostingu nebo code quality. Index, který využívá informace z analýzy zákaznických objednávek měření kvality kódu je Tiobe index.[24] Tento rating vedle popularity avšak nejvíce reflektuje jací zákazníci nakupují měření kvality kódu. Bude se jednat o větší, spíše korporátní společnosti. Prvních 5 nejoblíbenějších jazyků z Tiobe ratingu je zobrazených v tabulce 4.1.

Za zmínku určitě stojí i statistiky portálu StackOverflow jakožto největšího dotazovacího fóra pro programátory.[25] Velice zajímavý je přehled oblíbenosti dle vyhledávání tutoriálů na google.com [26] a v tabulce 4.2.

Zde z prvních 5 zmizelo C a C++ a prvních pět jazyků vlastně ovládají jazyky, ve kterých se dají psát webové stránky (s různou úrovní složitosti a cílovými zákazníky). Postupně vyřadíme z úvah jazyky dle různých nežádoucích vlastností. Nejdříve to budou Python, PHP, C a C++. C a C++ nejsou vhodné pro projekt takového typu a nemají rozumnou podporu tvorby webo-

pořadí	jazyk	podíl
1	Java	20,846%
2	C	13,905%
3	C++	5,918%
4	C#	3,796%
5	Python	3,330%

Tabulka 4.1: Tiobe index pro duben 2016

pořadí	jazyk	podíl
1	Java	24,0%
2	Python	12,3%
3	PHP	10,6%
4	C#	8,8%
5	Javascript	7,5%

Tabulka 4.2: Oblíbenost programovacích jazyků dle vyhledávání tutoriálů ve vyhledávači Google z dubna 2016

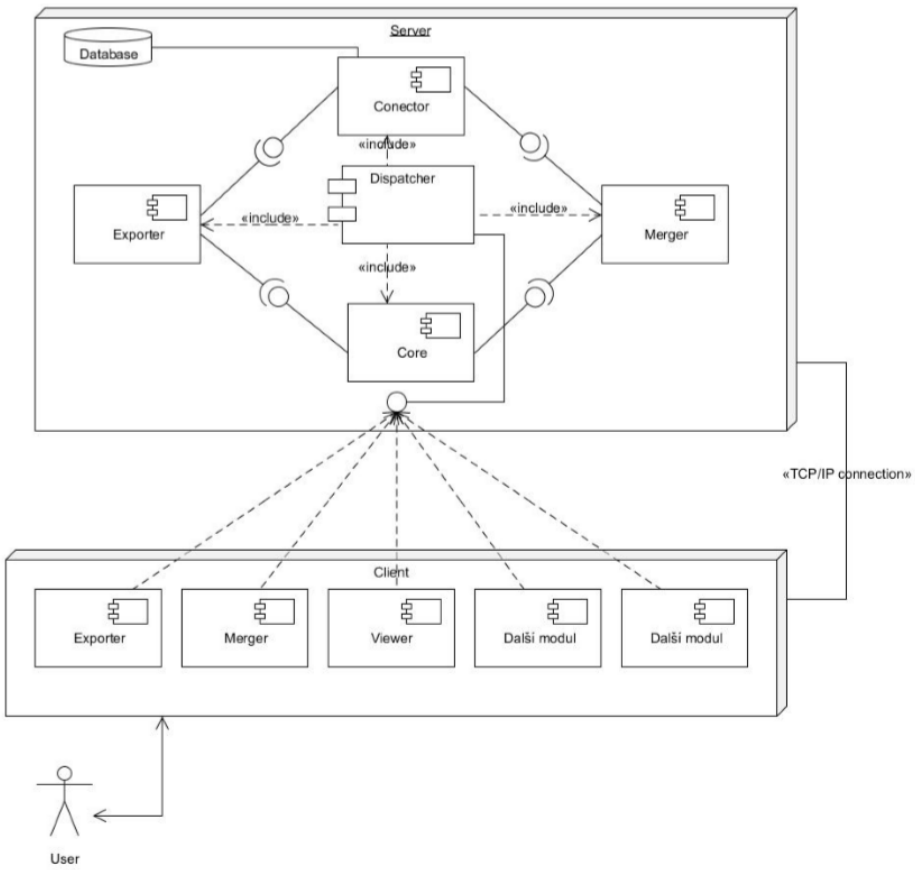
vých stránek. Python a PHP jsou slabě typové jazyky a pro tak rozsáhlý projekt nejsou vhodné.

Javascript je sice také slabě typový a nemá tak rozsáhlý ekosystém pro enterprise technologie, avšak v posledních několika letech je spousta nových, rychle se rozšiřujících technologií založena právě na Javascriptu. Ať už jde o single page aplikace na straně klienta, Node.js [27], nebo webové frameworky jakými jsou Angular [28] nebo React [29]. Oblíbenost Javascriptu dokládá i to, že je podporovaný některými grafovými databázemi - například OrientDB [30], ArangoDB [31] nebo Virtuoso [32]. Dalo by se říci, že každá nová technologie musí podporovat i Javascript. Jazyk, který byl navržen pro spouštění kódu webových stránek na straně klienta. Přes všechna tato pozitiva je nejspíše nejlepší počkat několik dalších let rozvoje ekosystému pro určení toho, zda tento ekosystém je již dozrálý.

Nakonec výběr mezi C# a Javou 8 je velice jednoduchý. Oba jazyky jsou si velice podobné. C# nabízí více „syntactic sugar“, který však může být ke škodě (viz. syntaxe Scaly). Rozdíl je spíše ve filosofii, zda chceme být v jednotném, přesně definovaném světě Microsoftu, který nám nenabízí moc jiných možností nebo úplným opakem. Protože projekt potřebuje specifické technologie, tak je vhodnější Java.

4.2 Architektura projektu Manta

Projekt Manta je v architektonickém řešení navržena jako aplikační server rozdělený na klientskou a serverovou část. Obě části jsou dále děleny na moduly. Klientská a serverová část má mezi sebou vystavené komunikační rozhraní a komunikace probíhá pomocí TCP/IP protokolu. Na diagramu architektury 4.1 je vidět, že komunikaci mezi serverovou částí a klientskými moduly zajišťuje modul Dispatcher. Serverová část obsahuje moduly zpracovávající business operace nad databází. Takovéto operace jsou vkládání, dotazování a export dat. Tyto operace jsou dále volány z Dispatcheru. Connector zajišťuje přístup k databázi spolu s rozhraním pro transakce. Modul Merger je pro tuto práci nejdůležitější. Zde dochází k provádění operací, které zajišťují vkládání a slučování dat v databáze. Exporter slouží k exportu databáze a Core obsa-



Obrázek 4.1: Architektura projektu Manta

huje základní operace nad databází typu vložení uzlu, hrany, definici datových struktur, vytvoření databázových indexů a podobně.

Celé řešení využívá framework Spring [33] a Maven pro sestavení aplikace, řízení závislostí mezi moduly a správu externích knihoven.

4.3 Datový model

Data jsou uložena v grafové struktuře - uzly propojené hranami. Jak již bylo zmíněno, tak Titan umožnuje vytvářet key-value atributy pro uzly i hrany. Navíc každá hrana musí být označena typem (label). Atributy mohou být indexovány a value má vždy libovolný datový typ nebo objekt (tak jak jsou v jazyku Java). I když jde do Titanu přidávat atributy a typy za běhu, tak se to z výkonnostních důvodů nedoporučuje. V projektu jsou definované typy hran. Něco jako typ uzlu v Titanu neexistuje. Uzel získává sémantický význam tím, jaké typy hran do něj či z něj vedou nebo jaké má atributy. Níže uvedený

přehled ukazuje typy a jejich využití.

4.3.1 Typy uzlů

1. Kořenový uzel - slouží k přístupu do grafu dat. V databázi existuje jen jeden unikátní.
2. Zdrojový uzel (Resource) - tento uzel obsahuje vstupní bod do stromu dat - hierarchie. Vždy se jedná o jméno technologie nebo typu modelu o který se jedná.
3. Datový uzel - obsahuje jména a typy datových uzlů v hierarchii. Datový uzel může mít například typ tabulka, schéma, databáze, soubor, server apod. Datový uzel neobsahuje žádné jiné atributy popisující data.
4. Atributový uzel - z důvodu optimalizace vkládání a dohledávání atributů datového uzlu existuje tento typ uzlu jako schránka pro typ atributu a jeho hodnotu.
5. Kořenový uzel revizí - uzel, který slouží k rychlému přístupu ke stromu revizí. V databázi existuje jen jeden a je unikátní.
6. Revizní uzel - reprezentuje jednotlivé revize - má atributy číslo revize a datum vytvoření revize.

4.3.2 Typy hran (labely)

- HAS_REVISION (hasRevision) - typ hrany spojující revizní uzel s kořenovým uzlem revizí. Obsahuje atribut číslo revize pro snadnější dotazování. Z každého revizního uzlu vychází jedna takováto hrana.
- HAS_PARENT (hasParent) - typ hrany spojující dva datové uzly. Určuje vztah dítě -> rodič v hierarchii. Například tabulka má jako rodiče schéma a sloupeček má tabulku. Každý datový uzel má jednu takovouto hranu.
- HAS_RESOURCE (hasResource) - určuje do jaké zdrojové hierarchie uzel patří. Tyto hrany vždy vedou z datového uzlu do zdrojového uzlu nebo ze zdrojového do kořenového. Může se stát, že uzel má za zdroj jinou hierarchii než rodič. V reálných datech se ukazuje, že toto je případ u méně než 0,1 procenta datových uzlů.
- HAS_ATTRIBUTE (hasAttribute) - spojuje atributový uzel s jeho datovým uzlem. Tím datovému uzlu přidává jeho význam.
- DIRECT (directFlow) - značí datovou transformaci mezi dvěma listovými datovými uzly ve dvou hierarchiích. Příkladem může být sloupec, který se stává atributem v proceduře.

- FILTER (filterFlow) - Ze zdroje hrany vede nepřímý datový tok do jejího cíle. Což znamená, že je hodnota jednoho listového datového uzlu filtrována hodnotou z jiného listového uzlu.

4.3.3 Hranové atributy

- CHILD_NAME (childName) - používá se pro HAS_PARENT hranu pro rychlejší vyhledávání při traverzování.
- TRAN_START (tranStart) - určuje první revizi, ve které byl datový nebo zdrojový uzel přidán.
- TRAN_END (tranEnd) - určuje poslední revizi, ve které byl datový nebo zdrojový uzel přidán. Pokud bude v následující revizi uzel znovu přidán, tak se změní jen tato hodnota na hraně.

4.3.4 Uzlové atributy

Na rozdíl od předchozích tří typů jsou uzlové atributy nejméně důležité v datovém schématu. Výjimku tvoří jen atributy označující kořenový uzel a kořenový uzel revizí.

- SUPER_ROOT (superRoot) - označuje kořenový uzel.
- REVISION_ROOT (revisionRoot) - označuje kořenový uzel revizí.

Zbylé atributy jsou:

- RESOURCE_NAME (resourceName) - jméno zdrojového uzlu.
- RESOURCE_TYPE (resourceType) - typ zdrojového uzlu.
- RESOURCE_DESCRIPTION (resourceDesc) - popisek zdrojového uzlu.
- NODE_TYPE (nodeType) - typ datového uzlu. Například server, tabulka, sloupec.
- NODE_NAME (nodeName) - jméno datového uzlu.
- ATTRIBUTE_NAME (attributeName) - klíč atributového uzlu.
- ATTRIBUTE_VALUE (attributeValue) - hodnota atributového uzlu.
- REVISION_NODE_REVISION (revisionNumber) - číslo revize revizního uzlu.
- REVISION_NODE_COMMITED (revisionCommited) - označení, zda bylo vkládání celé revize dokončeno.

- REVISION_NODE_COMMIT_TIME (revisionEnd) - čas dokončení operace sloučení (merge).

Z této definice je jasné, že uzly v databázi nijak „neplavou“. Každý uzel má hranu směrem k nějakému typu rodiče. Jedinou výjimkou jsou dva kořenové uzly, které jsou kořeny dvou oddělených stromů definovaných *HAS_PARENT* a *HAS_ATTRIBUTE* hranami. Hrany *HAS_PARENT*, *DIRECT* a *FILTER* jsou výsledkem analýzy databázových skriptů a transformačních skriptů. Všechny ostatní hrany jsou zde pomocné pro zajištění optimálního chování a snadnému přístupu k datům. Validní databázi dle výše zmíněného datového modelu je vidět na diagramu 4.2 a 4.3.

4.4 Analýza požadavků

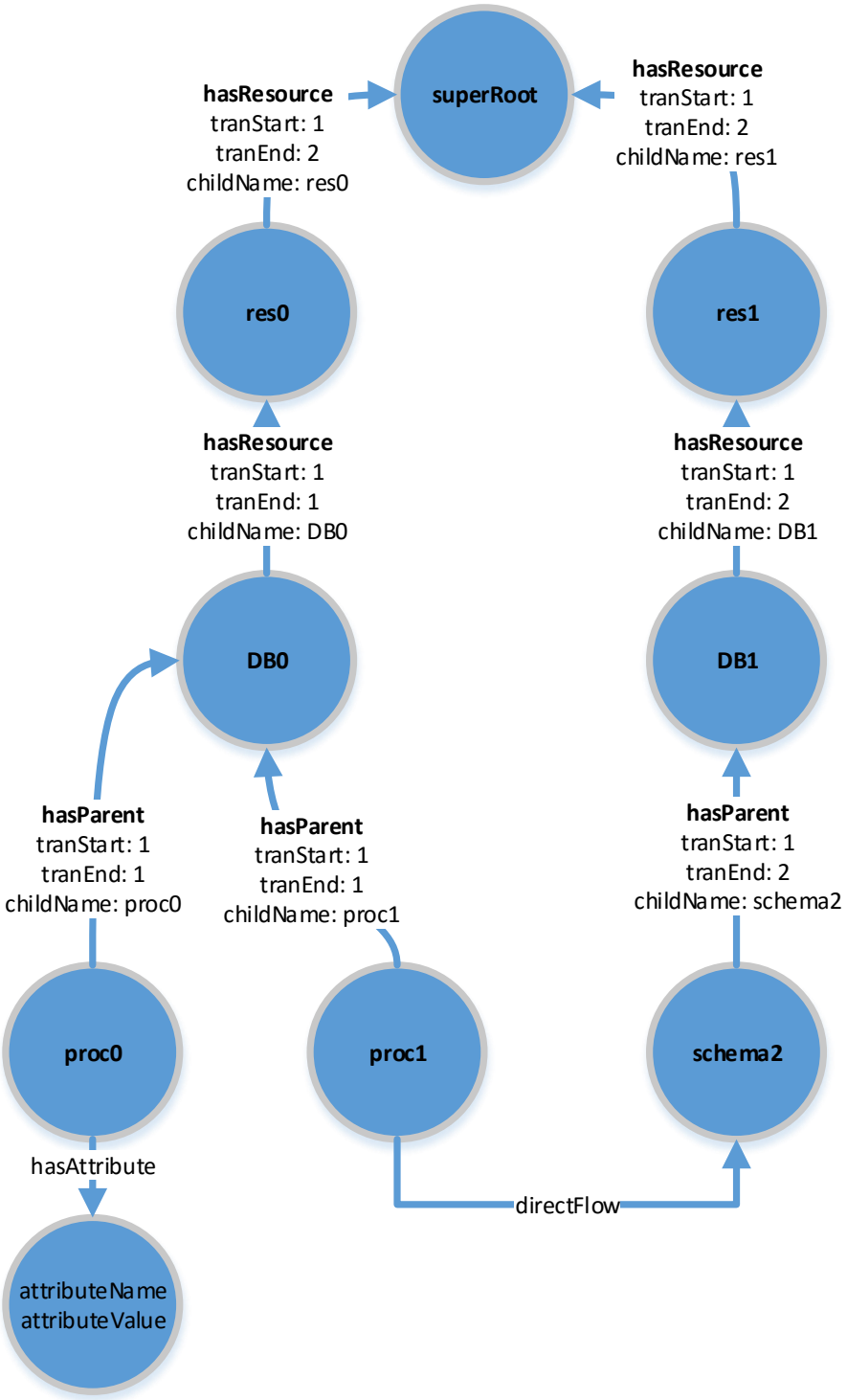
Cílem práce je umožnit uložení více pohledů na data a to pomocí nějakého obecného rozdělení na hierarchie. Samotné uložení však nestačí. Je nutné vymyslet způsob, pomocí kterého bude možné správně určit významově ekvivalentní uzly ve dvou či více hierarchiích (modelech). Do modulu, který vkládá uzly do databáze přichází informace o id uzlu z analýzy dat, id jeho rodiče, zdroje do kterého spadá, jména uzlu a jeho typu. Na diagramu 4.4 jsou k vidění dvě různé hierarchie. Barevně jsou zvýrazněné dvojice u kterých můžeme mít pocit, že by měly být shodné. Jenže jsou opravdu shodné? A co kdybychom v jednom modelu neměli typ DB, ale říkalo by se tomu databáze nebo anglicky database? Co kdyby v jedné byly vztahy mezi sebou jinak definované?

Vedle těchto otázek je třeba myslet i na různé aspekty problematiky. V ideálním případě by se velikost databáze neměla zvětšit o nějaký násobek při stejném objemu dat, ale jen o nějaké procento. Řešení musí umožnit nejen nalezení stejné dvojice v rozumném čase, ale musí poskytovat možnost snadného traverzování mezi sémanticky stejnými uzly v jiných hierarchiích. Největším problémem však zůstává rozeznávání stejných uzlů.

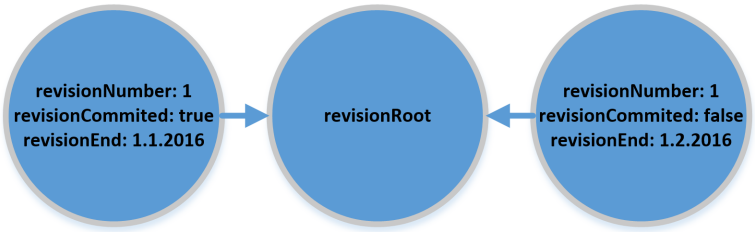
Dále je ještě třeba dodat, že se zabýváme jen modely, které mají ekvivalentní elementy 1:1 a mají stejný název. Jinak definované ekvivalentní elementy by mohly znamenat velké změny v celkovém návrhu.

4.5 Metamodel

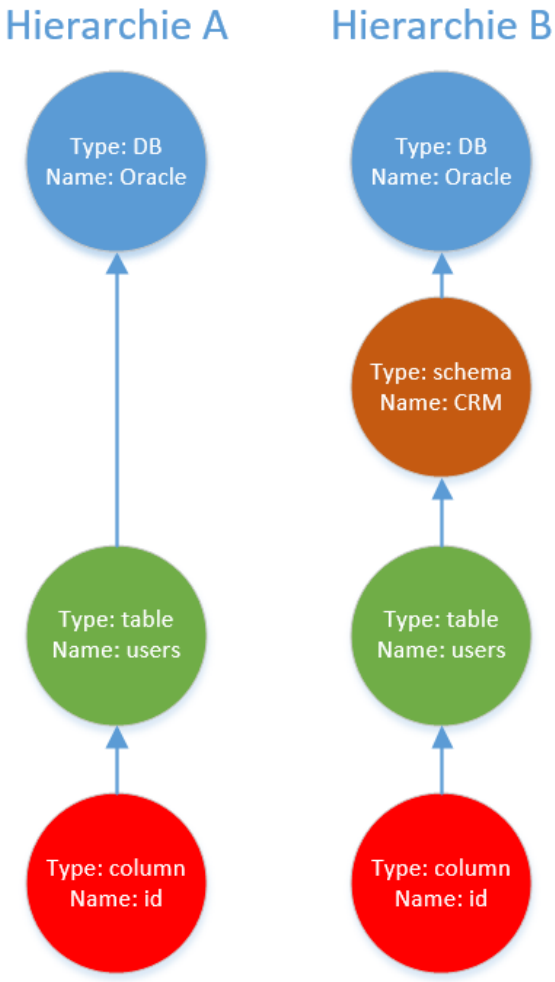
Řešením problému efektivního rozpoznávání stejných struktur, které je možné sloučit do stejných podstromů pojmenujeme metamodel. Za myšlenkou metamodelu stojí zjevná nutnost definovat vztahy mezi jednotlivými uzly v grafu dat. A tyto vztahy následně vynucovat při importu dat. Metamodel jako takový je jen šablona toho jaké vztahy mají mít typy uzlů v grafu dat. Díky tomu budeme mít jistotu, kde se určitý typ uzlu může nacházet. To nám pomůže při hledání ekvivalentního dvojčete. Jednoduchý metamodel můžeme vidět na diagramu 4.5.



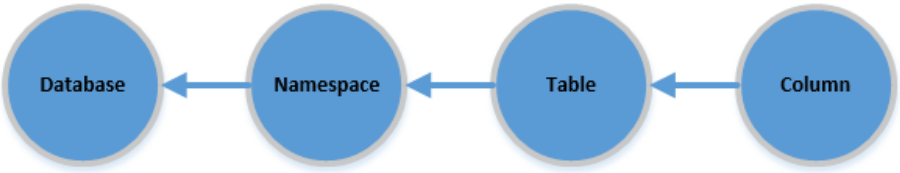
Obrázek 4.2: Ukázka databáze dle datového modelu



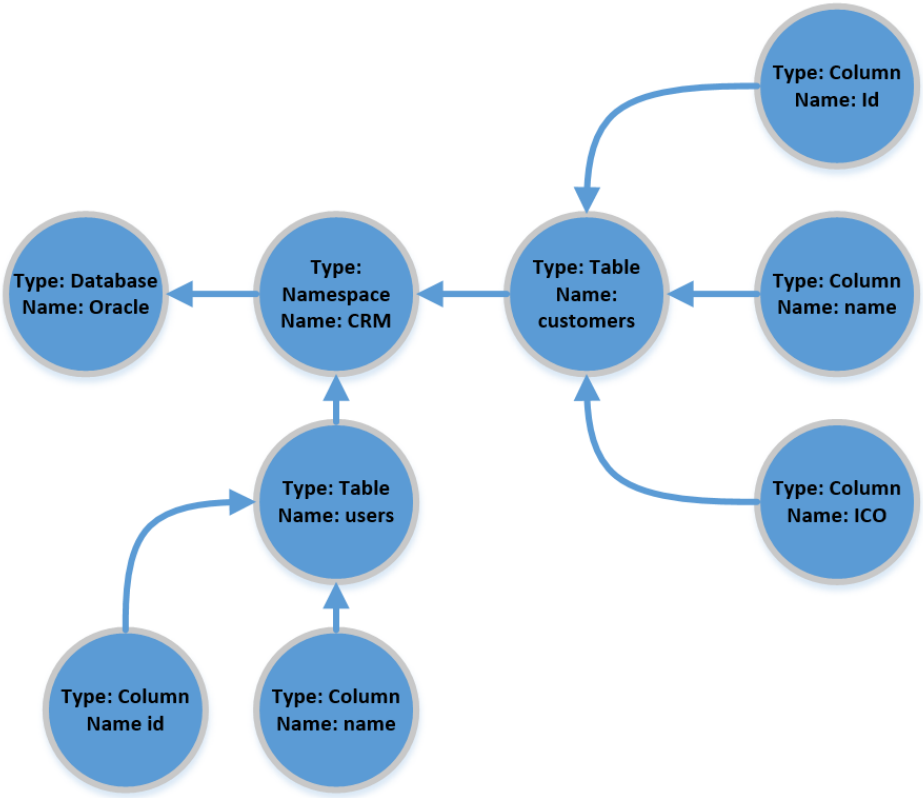
Obrázek 4.3: Ukázka databáze dle datového modelu - část revizních uzlů



Obrázek 4.4: Dvě rozdílné hierarchie - modely se zvýrazněnými ekvivaletními uzly



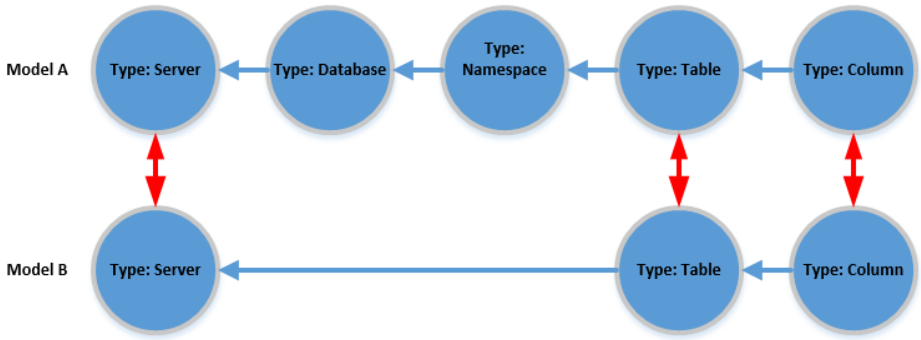
Obrázek 4.5: Jednoduchý metamodel popisující strukturu databáze



Obrázek 4.6: Ukázkový graf dat k metamodelu 4.5

Tento obrázek nám říká, že uzel typu tabulka může v grafu dat existovat jen pokud je jeho rodičem typ namespace. Obdobně namespace musí být pod databází a sloupeček pod tabulkou. Dle takovéhoto metamodelu může jednoduchý graf dat vypadat třeba jak na diagramu 4.6.

Struktura vztahů uzlů odpovídá šabloně metamodelu. Výše zobrazený metamodel odpovídá jen jedné hierarchii. Každá hierarchie musí mít v metamodelu alespoň jeden takový graf. K čemu je vlastně takováto struktura dobrá? Z definice posloupnosti typů rodičovství uzlů bude možné určit uzly, které mohou být sloučeny. Formálně metamodel definujeme jako stromovou strukturu.



Obrázek 4.7: Metamodel s 2 modely a znázorněnými ekvivalencemi

Každý uzel bude mít žádného nebo právě jednoho předka. Všechny děti mají unikátní jméno. V metamodelu se nebude rozlišovat mezi uzly typu resource a běžnými uzly. Logicky však vyplývá, že uzel, který v metamodelu nemá předka reprezentuje resource. A to z důvodu, že každý uzel, který není typu resource musí mít hned pod kořenem předka typu resource. Typu takovéto hrany od dítěte k předku v metamodelu budeme nazývat *HAS_PARENT*.

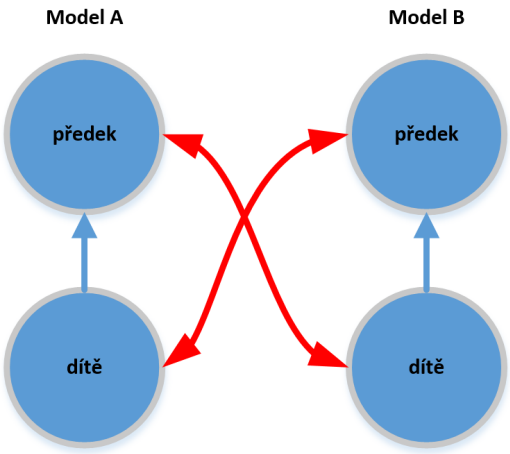
Toto nám však samo o sobě nepomůže co do hledání ekvivalencí mezi hierarchiemi. Obecně řečeno ne vždy musí být v každé hierarchii stejně pojmenovaný sémanticky stejný typ a obráceně řečeno také nemusí vždy stejně pojmenovaný typ být sémanticky stejný. Z tohoto důvodu zavedeme ještě jeden typ hrany a to METAMODEL_EQUAL. Díky této hraně přesně víme, které dva typy uzlů mohou být stejné. Tím, že jsou spojené se dokonce mohou jmenovat jinak. Navíc díky tomu můžeme definovat shodnost dvou uzlů, které jsou někde hluboko ve stromě.

- Dva datové uzly ze dvou různých hierarchií jsou shodné, pokud mají stejné jméno, jejich typy jsou v metamodelu spojené hranou typu *METAMODEL_EQUAL* a jejich první předci v datech, kteří mezi těmito dvěma hierarchiemi mají také ekvivalentní hranu v metamodlu, jsou také shodní.

Díky této definici tedy můžeme vyžadovat, aby dva shodné uzly měly shodné všechny předky, ve kterých je možná ekvivalence a zároveň díky rekurzivitě definice je potřeba kontrolovat jen dvě dvojice. Aby definice byla kompletní, tak je ještě třeba dodat část o chování zdrojového uzlu.

- Dva zdrojové uzly ze dvou různých hierarchií jsou ekvivaletní pokud mají stejné jméno a existuje hrana mezi nimi v metamodelu.

Na diagramu 4.7 si můžeme prohlédnout dvě hierarchie reprezentované dvěma modely metamodelu a červené ekvivalentní hrany mezi některými uzly.



Obrázek 4.8: Chybně definovaný metamodel

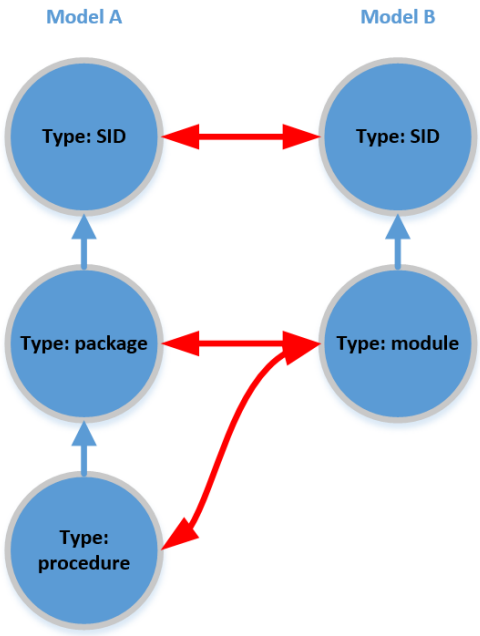
4.6 Speciální příklady metamodelu

Uzly ve stejném modelu nemohou mít mezi sebou ekvivalentní hranu. Principiálně by mohly mít, ale myslím si, že by to spíše vedlo k chybám v kódu a jedná se o zbytečnou funkcionalitu.

Striktně stromová definice metamodelu neposkytuje prostor pro nevalidní metamodely. Jediné co dává prostor pro chyby jsou ekvivalentní hrany. Není možné, aby se dvě ekvivalentní hrany křížily. Pokud máme čtyři uzly ve dvou modelech - dva předky a dvě jejich děti, tak není možné, aby dvě ekvivalentní hrany v jednom modelu byly vedeny z předka do uzlu dítěte a z dítěte do uzlu předka v druhém modelu. Viz diagram 4.8. Programově tento problém nebude ověřován, protože se nepředpokládá, že by uživatelé vytvářeli vlastní metamodely a vyhodnocování by tedy bylo zbytečně náročné.

Zvláštní případ metamodelu je zobrazený na diagramu 4.9. Zde si můžeme všimnout pár zajímavých věcí. Prvně se zde snažíme naznačit dvě ekvivalence, které vedou do stejného uzlu. Tomu budeme říkat zhušťovací problém. V diagramu chybí tranzitivně logická hrana mezi package a procedure. V metamodelu by všechny ekvivalentní hrany měly být tranzitivní s výjimkou hran mezi uzly ve stejném modelu. Poslední zajímavost je to, že procedure se nikdy nepodaří navázat na nějaký module. To je způsobené tím, že nikdy nemůže být splněna podmínka stejného rodiče, jak je to popsáno v definici. Aby toto bylo možné, tak by se musela pravidla metamodelu jinak definovat.

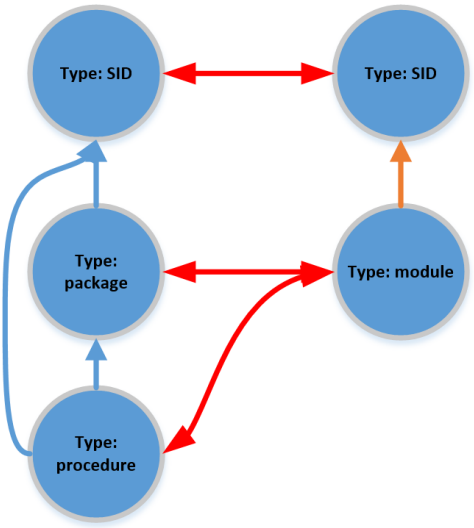
Metamodelový uzel bude mít za předka metamodelový uzel nebo uzel nového typu - *METAMODEL_ROOT*. Uzly budou spojeny hranou, která bude mít typ (label) *HAS_PARENT*.



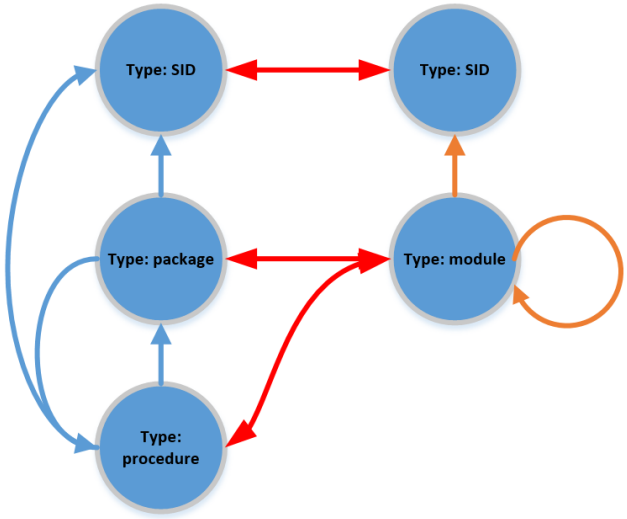
Obrázek 4.9: Metamodel zobrazující zhušťovací ideu

4.7 Modelovací síla metamodelu

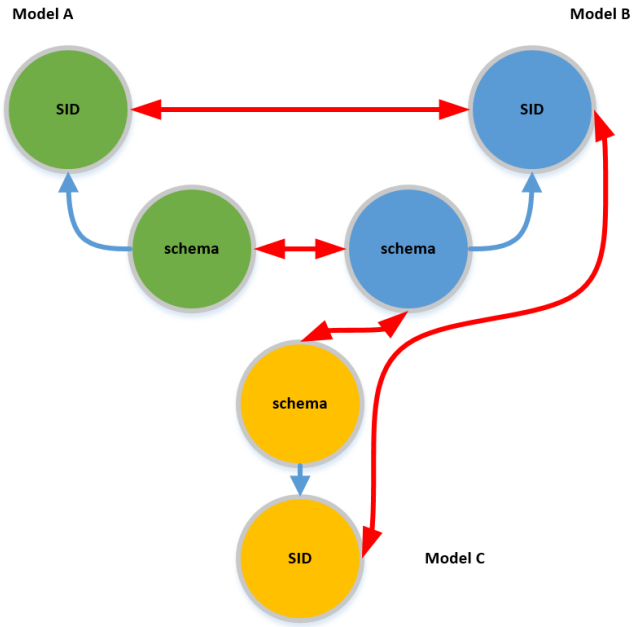
Jak bylo vidět na příkladu v 4.6, tak modelovací síla metamodelu je omezena striktně stromovou strukturou. Jako první se nabízí možnost dědit od více předků ze stejného podstromu. Příklad je na diagramu 4.10. Již je tedy možné mít proceduru, která je modulem. Protože je zde zhuštění, tak by na tomto místě bylo vhodné umožnit uzlu být předkem sám sobě. Tím bude moci uzel generovat až nekonečnou řadu svých potomků. Pokud se nám podaří vymyslet algoritmus, který nebude mít poblém s nalezením uzlů, které chceme vkládat, tak z rekurzivní definice shodnosti dvou uzlů by principiálně nemělo vadit ani rodičovství do svého potomka. Tento případ je zobrazen na diagramu 4.11. Algoritmus, který by byl schopen vkládat data dle takového metamodelu, by nejspíše byl schopný vkládat uzly s rodiči v různých podstromech nebo s cykly do jiných podstromů. Jedná se spíše o teoretickou otázku, kterou nebudeme od algoritmů vyžadovat, protože tento případ nemá reálný podklad ve vkládaných datech. Algoritmicky může být složité splnit požadavky rozšířené definice metamodelu a proto dále budeme rozlišovat i to, zda algoritmy tyto rozšířené požadavky splňují.



Obrázek 4.10: Metamodel umožňující více předků



Obrázek 4.11: Metamodel umožňující více předků, předka ve svém podstromu a předka sám sobě



Obrázek 4.12: Tři metamodely ve stavu před doplněním tranzitivních hran ekvivalence

4.8 Dopočet metamodelu

K metamodelu se váže ještě jeden požadavek. Metamodelů může být mnoho a proto by mohlo být nepříjemné zadávat vždy všechny ekvivalence s ostatními modely. Předpokládám, že většina modelů bude mít velice podobné ekvivalence mezi sebou. V tomto případě by bylo vhodné umožnit uživateli zadat jen nutné ekvivalence a zbylé hrany přidat, tak aby byla splněna tranzitivita. Na diagramu 4.12 můžeme vidět minimální nutné ekvivalentní hrany. Zbylé hrany mezi modelem A (zelený) a modelem C (žlutý) se samy vytvoří. V případě, že bude modelů mnoho a budou mít i desítky uzlů, tak toto bude velká úspora při vytváření správných ekvivalencí mezi modely.

4.9 Ukládání datových uzlů

Struktura metamodelu je jasná. Otázkou zůstává jak ukládat datové uzly. Řešení, které se doslova nabízí, by bylo ukládání uzlů každé hierarchie zvlášť a shodné uzly by byly propojeny hranou znázorňující ekvivalenci. Navíc by každý uzel měl unikátní id typu dle metamodelu. To by pomohlo v rychlém nalezení metamodelového vzoru daného uzlu. Poté, co bychom měli metamodelový vzor datového uzlu již snadno zjistíme, zda může mít nějaké ekvivalence a v jaké množině uzlů je hledat. Toto řešení má několik výhod. Je snadno představitelné a logika revizí by zůstala nezměněna. Spíše teoretickou výhodou je,

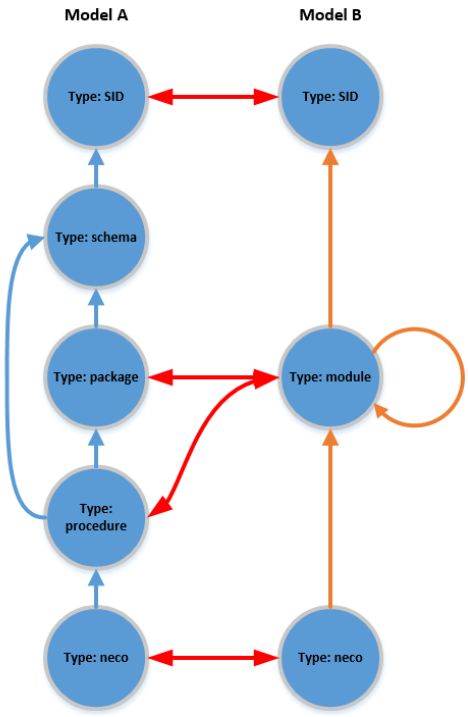
že v případě rozdělení databáze do clusteru by bylo možné dělit databázi po celých kusech hierarchií. Výhoda je však jen teoretická, protože v současné době databáze nedosahuje velikosti kde by se dalo uvažovat o rozdělení po clusteru. Na druhou stranu je možné, že využití mnoha hierarchií by tento stav mohlo změnit na nutnost a proto je dobré vědět o této možnosti. Problémem tohoto způsobu je, že když je n uzlů shodných, tak datově zaberou místo n uzlů. Za menší nevýhodu se dá považovat to, že k zjištění sousedních uzlů ekvivalentního uzlu potřebujeme do něj nejdříve traverzovat.

Existuje řešení, které je mnohem složitější na představení, ale výrazně šetří místem a umožňuje snadný přechod z jedné hierarchie do druhé. V tomto řešení jsou všechny sémanticky shodné uzly uloženy jako jeden uzel. Informace o tom, v jakých hierarchiích se tento schizofrenní uzel nachází, musí logicky být obsažena na hranách. Každá hrana tedy bude mít typ *HAS_PARENT* nebo *HAS_RESOURCE* a k tomu tyto atributy - identifikátor modelu, první a poslední revizi. Uzel navíc bude muset obsahovat identifikátor uzlu dle metamodelu. Protože je uzel ve více stavech najednou, tak identifikátor bude odpovídat prvně vloženému uzlu. Všechny ostatní jsou logicky platné z daných ekvivalencí. Ukázkový metamodel s rozšířenou funkcionalitou je zobrazen na diagramu 4.13. Vkládaná data dle modelu A 4.14 a modelu B 4.15. Výsledek datové části grafu po vložení modelu A a modelu B je vidět na diagramu 4.16. Na výsledném grafu stojí za zmínku mimo sloučených uzlů například to, jak je uzel „schema0“ přeskočeno, protože není v modelu B.

Pro úplnost je třeba říci, že existuje alternativa k určení hierarchií podle hran. Řešení by muselo mít tyto informace na uzlu. Tj. uzel by musel obsahovat seznam netriviálních objektů obsahující tři informace - jméno hierarchie, číslo první a poslední revize. Toto řešení je však nevhodné z několika důvodů. Jednak grafová databáze umožnuje traverzování po hranách což znamená dotazovaní se na atributy hran. Toto řešení by vynucovalo při hledání sousedů ve stejné hierarchii přejití do všech sousedních uzlů a v nich vždy projít zmíněný netriviální list. Zároveň by zde nebylo možné žádné rozumné indexování. Zkrátka toto řešení by nedávalo smysl.

4.10 Uložení ostatních typů

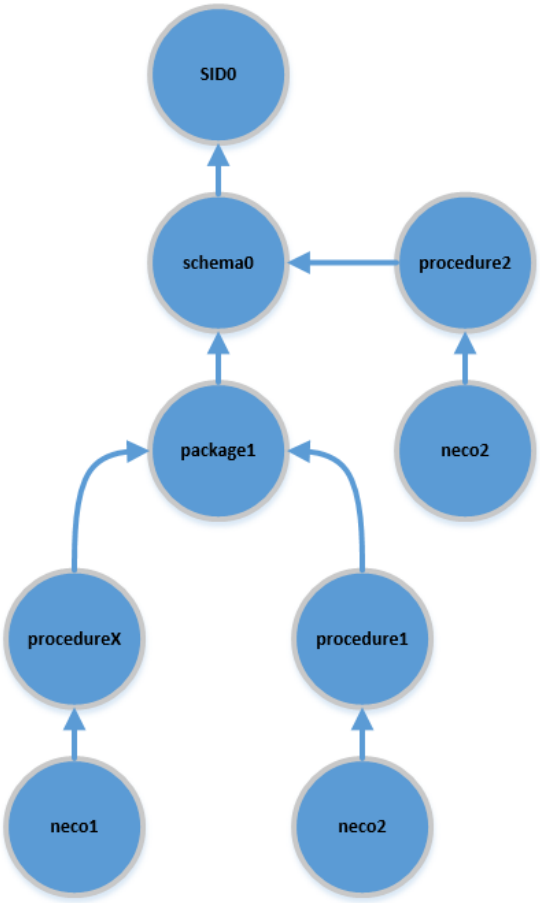
Obě možná řešení ukládání dat nevyžadují další složité změny v dalších strukturách. Jediná potřebná změna je přidání atributu označujícího model do hran vedoucích k atributovým uzlům a při vkládání hran typu directFlow nebo filterFlow. Tato změna není úplně nutná, ale myslím si, že je lepší vynutit rozlišování jaké atributy má uzel v daném modelu. To sice vynutí větší množství hran těchto typů, ale neomezí to funkcionalitu. Vždy je možné vynechat kritérium příslušnosti do metamodelu v dotazu vyhledávajícím atributové uzly.



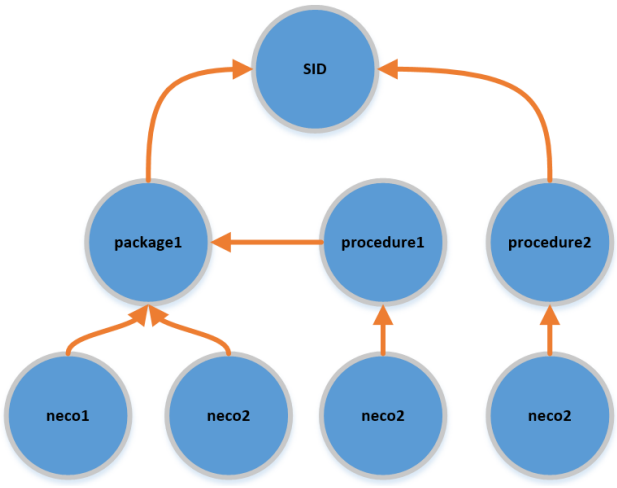
Obrázek 4.13: Ukázkový metamodel dvou hierarchií

4.11 Porovnání možností ukládání datových uzlů

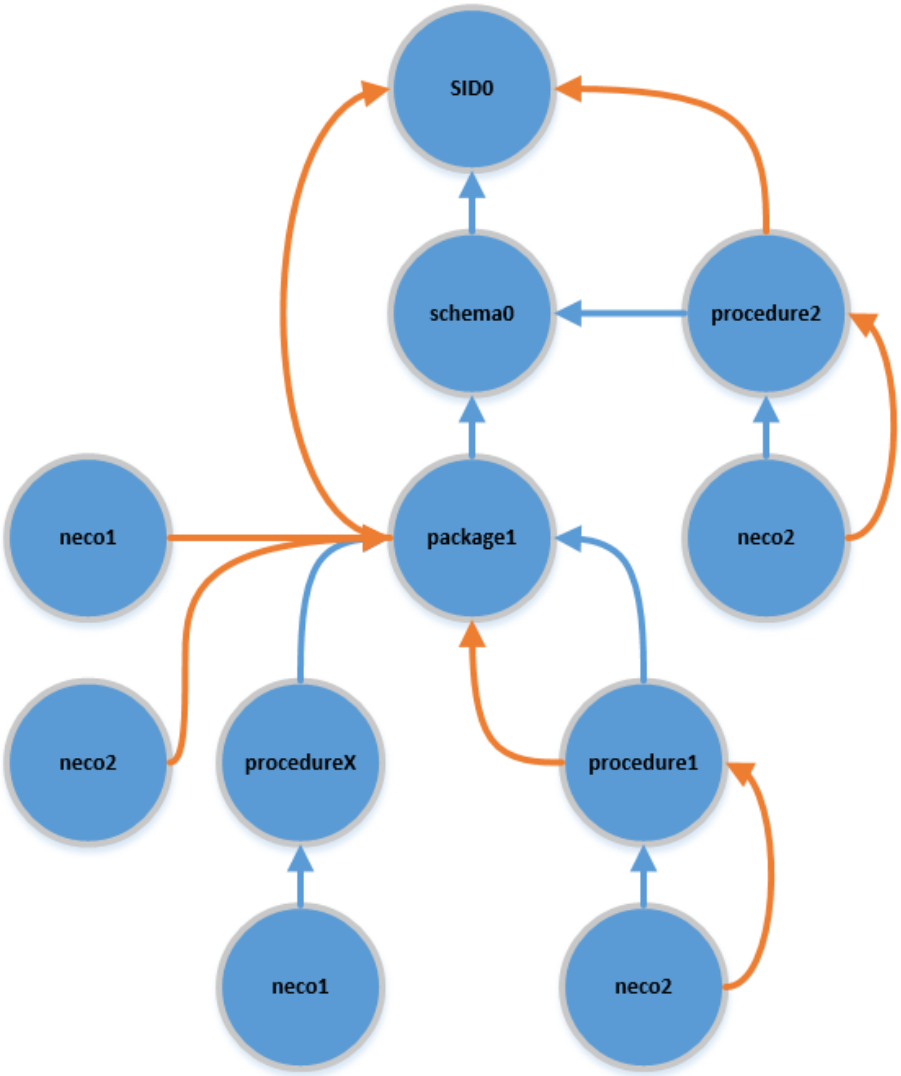
Jak řešení oddělených hierarchií, tak varianta rozlišení hranou, má své výhody i nevýhody v rámci různých aspektů. Oboje řešení nevyžadují žádné speciální změny v logice revizí a ani ostatních typech uzlů a hran. Řešení s oddělenými hierarchiemi je sice jednodušší, lépe představitelné a má větší potenciál při rozdělení databáze v clusteru, ale jeho velikost je zbytečně veliká. Rozlišení hranou je tedy v současné době lepší řešení. Navíc i přes velký rozdíl v uložení dat v jednotlivých řešení není tato volba zavazující. Velikou výhodou je, že obě řešení jsou mezi sebou snadno převoditelné. Protože pokud bychom měli algoritmy, které provádí vkládání do obou typů, tak vždy lze současnou databázi ať už je jakéhokoliv typu vyexportovat a znovu vložit. Dále tedy budeme pokračovat cestou oddělení hierarchií hranou.



Obrázek 4.14: Graf dat vkládaných dle modelu A



Obrázek 4.15: Graf dat vkládaných dle modelu B



Obrázek 4.16: Výsledný stav dat v databázi po vložení dat

Kapitola 5

Realizace

5.1 Cíle

Cílem této části je popsat realizaci zvoleného řešení hranově oddělených hierarchií. Dále jsou zmíněny klíčové implementační změny v jednotlivých modulech. Pořadí modulů je dáno jejich závislostmi. Tudíž změny jsou zmíněny v pořadí Core, Connector, Dispatcher a Merger-server-logic. Zásadní změny v Merger-server-titan nebyly provedeny. Nejvíce změn bylo provedeno v modulu Merger, kde byly implementovány dva algoritmy slučování uzlů. Tyto dva algoritmy budou následně porovnány v kapitole 6. Při implementaci jsem se rozhodl nepřepisovat staré ukládání dat. Místo toho jsem vytvořil nové třídy a metody ve stejném stylu. Interface některých metod se liší jen jménem modelu. Toto řešení jsem zvolil ze dvou důvodů. Hlavním důvodem je, že se může stát, že uživatel z nějakého důvodu nebude chtít používat metamodely. Důvodů může mít spoustu. Nemusí mít data vhodná pro tuto funkcionalitu, bude si přát rychlejší vkládání anebo může mít jakýkoliv jiný důvod. Přijde mi tedy správné mu tuto funkcionalitu nenutit. Druhá výhoda je výhodou pro mne a to, že při testování mohu snadno porovnat výkonnost starého způsobu vkládání dat a nového. Navíc smazat starý kód je vždy snadnější než zpětně tuto funkcionalitu přidávat.

Všechny uvedené části kódu v této práci jsou napsány v jazyce Java 7.

5.2 Postup

Vzhledem k jasné specifikaci a nutném důrazu nejen na úspěšné provedení operací vkládání, ale i na správnost výsledku dle definic byl postup realizace veden stylem test-first. Nejdříve byly vytvořeny testy na správnost vkládání metamodelu. Pak byla naprogramována část vkládání metamodelu. Stejný postup byl uplatněn u funkcionality vkládání dat. Testy jsou si podobné. Vždy jsou vložena data speciálně připravená na konkrétní případ funkcionality a následně je zkontrolován počet, sousednost a shodnost uzlů. Velikost vkládaných

grafů se mění dle požadované funkcionality. Některé obsahují jen pár uzlů a kontrolují základní funkcionalitu. Jiné testy obsahují desítky vkládaných uzlů. Pro testování je použita knihovna JUnit. Příklad inicializace testu je vidět na následujícím kusu kódu. Tento kus kódu se spustí vždy před začátkem testu díky anotaci *Before*.

```
protected OneFileAdder modelAdderHelper;
protected MetamodelHelper metamodelHelper;
protected StandardModelProcessor modelProcessor;
protected MetamodelMergerProcessor mergerProcessor;
protected OneFileMerger mergerHelper;

@Before
public void setUp() {
cleanGraph();
modelProcessor = new StandardModelProcessor();
modelProcessor.setMetamodelRootHandler(getMetamodelRootHandler());

modelAdderHelper = new OneFileAdder(getDatabaseHolder(),
    getMetamodelRootHandler(), getSuperRootHandler(),
    getRevisionRootHandler(), modelProcessor);

metamodelHelper = new MetamodelHelper();
metamodelHelper.setMetamodelRootHandler(getMetamodelRootHandler());
metamodelHelper.setDatabaseService(getDatabaseHolder());

mergerProcessor = new MetamodelMergerProcessor();
mergerProcessor.setSuperRootHandler(getSuperRootHandler());
mergerProcessor.setMetamodelRootHandler(getMetamodelRootHandler());
mergerHelper = new OneFileMerger(getDatabaseHolder(),
    getSuperRootHandler(), getRevisionRootHandler(),
                getMetamodelRootHandler(), mergerProcessor, 2000);
mergerHelper.setMergerProcessor(mergerProcessor);
}
```

Následný test pak je tvořen vložením metamodelu, ověřením počtu uzlů, vložením dat, ověřením počtu uzlů a následnou kontrolou dat v databázi. Někdy může probíhat více fází vkládání dat. Velice se mi osvědčilo vytvoření stejného testu nad stejnými daty jen s prohozením pořadí vkládaných hierarchií. Tento dodatečný test je snadný na vytvoření, ale jeho přínos je opravdu veliký, protože výsledek musí být vždy stejný. Ukázka CSV formátu vkládaných dat pro grafy 4.14 a 4.15:

```
resource,1,SID0,SID,,1
node,2,1,schema0,schema,1,1
node,4,2,procedure2,procedure,1,1
node,5,4,neco2,neco,1,1
node,6,2,package1,package,1,1
```

```
node,7,6,procedureX,procedure,1,1
node,8,7,neco1,neco,1,1
node,9,6,procedure1,procedure,1,1
node,10,9,neco2,neco,1,1
resource,20,SID0,SID,,2
node,23,20,package1,modul,20,2
node,24,23,neco1,neco,20,2
node,25,23,neco2,neco,20,2
node,26,20,procedure2,modul,20,2
node,27,26,neco2,neco,20,2
node,28,23,procedure1,modul,20,2
node,29,28,neco2,neco,20,2
```

Dále následuje ukázka spuštění dvou testovacích scénářů - metody s anotací *Test*. Oba testy kontrolují stejnou strukturu i počet uzlů v databázi. Liší se jen pořadí uzlů ve vkládaném CSV souboru.

```
@Test
public void text15() throws FileNotFoundException {
    executeTest15(METAMODEL_DATA_15_FILE);
}

@Test
public void text15AnotherOrdering4() throws FileNotFoundException {
    executeTest15(METAMODEL_DATA_15_FILE_ANOTHER_ORDERING);
}


private void executeTest15(String datafile) throws
    FileNotFoundException {
    OneFileAdder modelAdderHelper = new
        OneFileAdder(getDatabaseHolder(), getMetamodelRootHandler(),
            getSuperRootHandler(), getRevisionRootHandler(),
                modelProcessor);

    // v~db by mel byt pouze revizni uzel, root, revision root,
        metamodel root
    getDatabaseHolder().runInWriteTransaction(new
        TestCallback<Object>() {
        @Override
        public Object callMe(TitanTransaction transaction) {
            Assert.assertEquals(INIT_GRAPH_VERTEX_COUNT,
                getVertexCount(transaction));
            return null;
        }
    });

    // nacteni metamodelu z~CSV
    modelAdderHelper.process(new FileInputStream(new
```

```
            File(METAMODEL_15_FILE)), false);
        // ... dalsi kontrola poctu uzlu
        // nactu CSV
        mergerHelper.process(new FileInputStream(new File(datafile)), 0);
        checkMetamodel15Data();
        // ... dalsi kontrola poctu uzlu
    }
```

Metoda, která kontroluje strukturu databáze obsahuje transakční callback. Nejdříve načte všechny uzly pomocí traverzování mezi nimi a následně zkontroluje jejich shodnost a různost. Tím se ověří zároveň sousednost, tak i spojení dvou modelů. Následuje část metody *checkMetamodel15Data* již jen část kódu bez transakčního callbacku.

```
TitanVertex root = ((TitanVertex)
    getSuperRootHandler().getRoot(transaction));
//MODEL1
Vertex SID0m1 = root.query().direction(Direction.IN)
        .labels(DatabaseStructure.EdgeLabel.HAS_RESOURCE.t())
        .has(DatabaseStructure.EdgeProperty.METAMODEL_NAME.t(), "1")
        .has(DatabaseStructure.EdgeProperty.CHILD_NAME.t(),
            "SID0").vertices().iterator().next();

Vertex schema0m1 = SID0m1.query().direction(Direction.IN)
        .labels(DatabaseStructure.EdgeLabel.HAS_RESOURCE.t())
        .has(DatabaseStructure.EdgeProperty.METAMODEL_NAME.t(), "1")
        .has(DatabaseStructure.EdgeProperty.CHILD_NAME.t(),
            "schema0").vertices().iterator().next();
//.. nacteni dalsich 15 uzlu
// overeni shodnosti
Assert.assertEquals(SID0m1, SID0m2);
Assert.assertEquals(pack1m1, pack1m2);
//.. atd.
// overeni ruznosti
Assert.assertNotSame(schema0m1, SID0m1);
//.. atd.
```

5.3 Core

Core obsahuje definice struktur v databázi. Do třídy *DataflowObjectFormats* jsem do enumu *ItemTypes* přidal nový typ a to uzel metamodelu. Zároveň jsem vytvořil enum který definuje v jakém formátu mají být metamodelové uzly v CSV. Tento zápis je shodný se zavedenými konvencemi.

```
public enum ItemTypes {
   // ... predesle
   METAMODEL_NODE("metamodel_node", 5),
```

```
}

public enum MetamodelNodeFormat {
   /** ID uzlu */
   ID(1),
   /** ID rodice, prazdny string pokud predka nema. */
   PARENT_ID(2),
   /** semanticky typ uzlu(to co ma reprezentovat - tabulka, uzel,
       server)*/
   META_TYPE(3),
   /** jmeno metamodelu = jmeho hierarchie kterou ve ktere je*/
   METAMODEL_NAME(4),
   /** popis typu metamodelu */
   DESCRIPTION(5);
   private int index;
   // ... dale definice konstruktoru a pomocnych metod
}
```

Do třídy *DatabaseStructure* jsem přidal nové hodnoty do enumů definujících labely a typy atributů. *METAMODEL_ROOT* slouží k přidání unikátního indexu na jediný kořenový uzel metamodelu v databázi. Díky tomu je možné získat tento uzel velice rychle a dále z něj traverzovat. Ostatní atributy uzlů až na *METAMODEL_NODE_ID* patří novému typu uzlů - uzel v metamodelu. Metamodelový uzel má za předky metamodelový uzel nebo uzel typu *METAMODEL_ROOT*. Uzly jsou spojeny hranou typu *HAS_PARENT*. *METAMODEL_NODE_ID* je atribut přidaný na každý datový uzel. Slouží k určení všech ekvivalentních typů (dle metamodelu) ve kterých se může uzel nacházet. Na hraně tento atribut znamená, že vede přesně z tohoto typu uzlu.

```
public enum NodeProperty {
  // ... predesle
  /** Atribut oznacujici korenovy uzel pro metamodel */
  METAMODEL_ROOT("metamodelRoot"),
  /** Atribut oznacujici typ metamodel do ktereho uzel patri*/
  METAMODEL_NAME("metamodelName"),
  /** delsi popis semantickeho vyznamu uzlu metamodelu */
  METAMODEL_DESCRIPTION("metamodelDescription"),
  /** semanticky typ uzlu metamodelu (co maji uzly takoveho typu
      predstavovat)*/
  METAMODEL_TYPE("metamodelType");
  /** databazove id uzlu v~metamodelu */
  METAMODEL_NODE_ID("metamodelNodeId"),
}

public enum EdgeLabel {
  //... predesle
  /** zdroj i cil maji stejny vyznam v~ramci metamodelu */
  METAMODEL_EQUAL("metamodelEqual","METAMODEL_EQUAL");
```

```
}

public enum EdgeProperty {
  //... predesle
  /** jmeno metamodelu do ktereho patri pocatecni uzel*/
  METAMODEL_NAME("metamodelName"),
  /** databazove id uzlu v~metamodelu */
  METAMODEL_NODE_ID("metamodelNodeId");
}
```

5.4 Connector

V modulu Connector sice proběhlo větší množství změn, ale zajímavých jich je jen pár. Změny přidání argumentu práce s metamodelem proběhly ve třídách *RevisionUtils*, *RevisionRoothandler*, *GraphOperation*, *SimpleDatabaseHolder* a jiných. Byla přidána řídicí třída pro přístup ke kořenu metamodelu - *MetamodelRootHandler*. Třída je v podobném duchu jako *SuperRoothandler* a *RevisionRootHandler*. Tyto třídy jsou injektovány do tříd (pomocí anotace *Autowired*), které s nimi chtějí pracovat. V projektu je využit Spring framework k poskytování této singleton funkcionality. Aby injektování fungovalo tak se navíc musí třída zaregistrovat v XML souboru *applicationContext* tímto způsobem:

```
<bean id="metamodelRoothandler" class="eu.profinit.manta.dataflow
    .repository.connector.titan.service.MetamodelRoothandler" />
```

Důležité změny byly provedeny v třídě *GraphCreation*. V té se nacházejí definice použitých indexů. Na příkladu níže je vidět index pro kořen metamodelu. Dále byl přidán index na jméno modelu a id uzlu v metamodelu. Kód těchto dvou indexů je obdobný tomuto:

```
if (!indexTransaction.getIndexedKeys(Vertex.class)
    .contains(NodeProperty.METAMODEL_ROOT.t())) {
    try {
        indexTransaction.makeKey(NodeProperty.METAMODEL_ROOT.t())
            .dataType(Boolean.class)
                .indexed(Vertex.class).unique().make();
        LOGGER.info("Create index for " +
            NodeProperty.METAMODEL_ROOT.t());
    } catch (UnsupportedOperationException e) {
        LOGGER.error("Error during creating index.", e);
    }
}
```

Přibyly i nové metody pro práci s metamodelem na úrovni datových uzlů, tak i metamodelových uzlů. Tyto metody slouží k vytváření uzlů a hran růz-

ných typů. Příklad takové metody je níže.

```
/**
 * vytvori uzel v~metamodelu. tento uzel neobsahuje revizni hrany
 * @param transaction transakce do db
 * @param parent predek, ke kteremu se uzel pripoji, muze byt i
     resource
 * @param metaType semanticky typ uzlu (co maji uzly takoveho typu
     predstavovat)
 * @param metamodelName jmeno metamodelu do ktereho uzel patri
 * @param description delsi popis toho co meta_type znamena
 * @return nove vytvoreny vertex
 */
public static Vertex createMetamodelNode(TitanTransaction
    transaction, Vertex parent, String metaType,
        Integer metamodelName, String description) {
    if (parent == null) {
        throw new IllegalArgumentException("The argument parent was
            null.");
    }
    Vertex newNode = transaction.addVertex(null);
    newNode.setProperty(NodeProperty.VERTEX_TYPE.t(),
        VertexType.NODE);
    newNode.setProperty(NodeProperty.METAMODEL_NAME.t(),
        metamodelName);
    newNode.setProperty(NodeProperty.METAMODEL_TYPE.t(), metaType);
    newNode.setProperty(NodeProperty.METAMODEL_DESCRIPTION.t(),
        description);

    EdgeLabel parentEdgeType = EdgeLabel.HAS_PARENT;
    Edge newEdge = newNode.addEdge(parentEdgeType.t(), parent);
    newEdge.setProperty(EdgeProperty.CHILD_NAME.t(),
        GraphOperation.processValueForChildName(metaType));
    newEdge.setProperty(EdgeProperty.METAMODEL_NAME.t(),
        metamodelName);
    return newNode;
}
```

5.5 Dispatcher

V Dispatcheru neproběhly žádné důležité změny. Jediná změna, která stojí za zmínku je opětovné přidání Spring beany do *applicationContext.xml* k umožnění injectování *MetamodelRootHandleru* do tříd.

5.6 Merger-server-logic

5.6.1 Funkcionalita Mergeru

Merger je jeden z nejdůležitějších modulů. Stará se o vkládání a slučování uzlů v databázi. Před hierarchiemi probíhalo slučování v rámci revizí. S každou novou revizí se nahrála nová data. Uzly, které měly svůj ekvivalent v předchozích verzích byly nalezeny a opravilo se jim jen číslo poslední revize (tato informace je na hraně od uzlu k rodiči). Merger zajišťuje sekvenční transakční zpracování příchozích dat v CSV formátu. Kde každý řádek je jedním datovým uzlem, zdrojovým uzlem, atributovým uzlem nebo jednou z možných hran.

Merger pracuje jedno-vláknově. Toto omezení je zde z důvodu prevence vkládání duplicit. V rámci této práce se tato logika neměnila.

5.6.2 Změna struktury

Rozdělil jsem logiku starého a nového vkládání dat. Konkrétně jsem do package *eu.profinit.manta.dataflow.repository.merger.server.helper.merger* přidal podpackage *metamodel* a *standard*. Všechny abstraktní třídy jsem nechal v tomto page. Starou, standard, logiku jsem přesunul do *standard* a do *metamodel* jsem jí překopíroval a začal upravovat. Pro jasnější oddělení jsem přejmenoval *StandardMergerProcessor* v package metamodel na *metamodelMergerProcessor*. V této třídě se nachází nejdůležitější část této práce - slučování uzlů. Pro každý řádek z CSV se zavolá metoda, která rozhodne co se dále vlastně vkládá.

```
@Override
public MergerProcessorResult mergeObject(String[] itemParameters,
    ProcessorContext context) {

    ItemTypes itemType = ItemTypes.parseString(itemParameters[0]);

    if (itemType != null) {
        switch (itemType) {
        case NODE:
            return new
                MergerProcessorResult(processNode(itemParameters,
                context), itemType);
        case EDGE:
            return new
                MergerProcessorResult(processEdge(itemParameters,
                context), itemType);
        case RESOURCE:
            return new
                MergerProcessorResult(processResource(itemParameters,
                context), itemType);
        case NODE_ATTRIBUTE:
```

```
            return new
                MergerProcessorResult(processNodeAttribute(itemParameters,
                context), itemType);
        case EDGE_ATTRIBUTE:
            return new
                MergerProcessorResult(processEdgeAttribute(itemParameters,
                context), itemType);
        default:
            LOGGER.warn("Unknown item type " + itemParameters[0] +
                ".");
            return new MergerProcessorResult(ResultType.ERROR, null);
        }
    } else {
        LOGGER.warn("Unknown item type " + itemParameters[0] + ".");
        return new MergerProcessorResult(ResultType.ERROR, null);
    }
}
```

Nejzajímavější úpravy v této práci jsou v metodách *processResource* a *processNode*. Aby algoritmus mohl fungovat a nedělal zbytečné operace, tak před začátkem ukládání datových uzlů je načten celý metamodel do paměti. Aby bylo možné v něm rychle vyhledávat, tak byly vytvořeny dvě pomocné třídy. První se jmenuje *MetamodelNode* a reprezentuje uzel v metamodelu. Obsahuje tyto důležité hodnoty:

```
// rodice
private List<MetamodelNode> parents;
// id uzlu v~databazi
private final Object dbId;
// deti
private Set<MetamodelNode> children;
// deti ve tvaru dbIds
private Set<Object> childrenIds;
// ekvivalentni uzly
private Set<MetamodelNode> modelEquivalents;
// ids ekvivalentnich uzlu
private Set<Object> modelEquivalentsIds;
// jmeno modelu
private Integer modelId;
// lazy cesta k~rodici, ktery ma do stejneho (jineho) modelu take
    equals hranu
private Map<Integer, List<Object>> routeToModelEquivalentParent;
// typ uzlu, ktery metamodelovy uzel predstavuje
private Object type;
```

Z důvodu existence lokálních id v rámci dat v CSV existovalo mapování lokálních id na databázová. Nyní bylo potřeba mít rychle informaci i o typu dle metamodelu. Proto vznikla druhá pomocná třída (*DbNode*), která je jen

kontejner pro id uzlu v databázi a referenci na *MetamodelNode*.

Načtení metamodelu do paměti je ve třídě *OneFileMerger* a jedná se o rekurzivní projití stromu metamodelu od metamodelového kořene po hranách s labelem *HAS_PARENT*.

5.6.3 Algoritmy slučování uzlů

Principiálně existují jen dva možné typy algoritmů, které mohou být použity k hledání ekvivalentních uzlů. Jakékoliv další možnosti musí logicky být nějakou optimalizací jednoho z těchto dvou přístupů. Algoritmy nazveme *UpDown* a *ByName*. Obecně se nedá předem určit, který přístup bude ve výsledku rychlejší nebo zda existuje nějaká variace, která by mohla být lepší. Jakákoliv variace by byla založena na úpravě algoritmu na základě nějaké možnosti optimalizačního postupu úpravy datového modelu. Taková variace může být specifická pro současnou databázi a v jiné již nemusí být platná a naopak by mohla způsobovat zpomalení. V případě metody *processResource* je celý kód zjednodušený, protože na rozdíl od *processNode* není potřeba provádět kontroly typů a jmen předků.

Oba algoritmy mají společnou část kde se snaží najít svého rodiče, ověřit shodu s metamodelem, ověřit zda již uzel neexistuje a pokud existuje, tak jen upraví číslo revize. Dále se rozhoduje podle toho, zda uzel může nebo nemůže mít ekvivalence. Tato společná část kódu je vidět níže. Pokud by byla v projektu použita Java 8 tak by byl kód kratší z důvodu použití lambda výrazů při filtrování dat.

Algoritmus *ByName* vznikl spíše jako ukázka možností, které rozšířená definice metamodelu nabízí.

```
/**
 * Zpracuje uzel. Vlozi novy vertex do grafu, pokud jiz takovy
     neexistuje. <br>
 * Pokud nema predka, je novy vertex spojen hranou {@link
     EdgeLabel#HAS_RESOURCE} se svym resource. <br>
 * Pokud ma predka, je novy vertex spojen hranou {@link
     EdgeLabel#HAS_PARENT} se svym predkem. <br>
 * @param itemParameters data vkladaneho zaznamu
 * @param context Kontext procesoru slouzici k~udrzovani informace
     pro mapovani mezi stavem databaze a zpracovavanych objektu.
 * @return vysledek, dle zpracovani objektu
 */
protected ResultType processNode(String[] itemParameters,
    ProcessorContext context) {
if (itemParameters.length != ItemTypes.NODE.getMetamodelSize()) {
    LOGGER.warn("Incorrect node record: " +
        StringUtils.join(itemParameters, CsvHelper.DELIMITER));
    return ResultType.ERROR;
}
```

```
// nastaveni promenny z~itemParameters a promenne revizi

DbNode resourceVertexId =
    context.getMapLocalIdToDbIdAndDbMmId().get(resourceId);
if (resourceVertexId == null)
  //.. chyba

Vertex resourceVertex =
    context.getDbTransaction().getVertex(resourceVertexId.getDbId());
if (resourceVertex == null)
  //.. chyba

Vertex parentVertex;
MetamodelNode parentMetamodelNode;
EdgeLabel parentEdgeType;
if (parentLocalId == null || parentLocalId.isEmpty()) {
   // .. jako rodic se nastavi resource
} else {
    parentEdgeType = EdgeLabel.HAS_PARENT;
    DbNode parentVertexId =
        context.getMapLocalIdToDbIdAndDbMmId().get(parentLocalId);
    if (parentVertexId == null || parentVertexId.getDbId() == null)
    //.. chyba
    parentVertex =
        context.getDbTransaction().getVertex(parentVertexId.getDbId());
    if (parentVertex == null)
     //.. chyba
    parentMetamodelNode = parentVertexId.getMetamodelNode();
}

//overeni zda rodic reprezentuje nektery z~ekvivaletnich uzlu v~danem
    modelu ktery ma za dite uzel se stejnym nodeType
boolean metamodelValidationOk = false;
MetamodelNode metamodelNode = null;
// ... cyklus kde se overuje

//overeni zda typ vkladaneho uzlu neni validnim potomkem v~metamodelu
    sveho rodice
if (!metamodelValidationOk) {
  //.. chyba
}

//zjisteni zda predek ma hranu k~vkladanemu uzlu v~danem modelu
TitanVertexQuery query = ((TitanVertex) parentVertex).query()
        .has(EdgeProperty.CHILD_NAME.t(),
            GraphOperation.processValueForChildName(name))
        .has(EdgeProperty.TRAN_END.t(), Cmp.GREATER_THAN_EQUAL,
            lastCommitedRevision).direction(Direction.IN)
        .has(EdgeProperty.METAMODEL_NAME.t(),
```

```
            modelName).labels(parentEdgeType.t());

Iterator<Vertex> vertexIterator = query.vertices().iterator();
boolean notYetExist = true;
//overeni zda uzel jiz existuje nebo ne
while (vertexIterator.hasNext()) {
    Vertex checkedVertex = vertexIterator.next();
    // equals je jen pres jmeno a typ uzlu a jeho predky, jiz se
        nebere jeho primy resource
    // -> neumoznujume dvojici se stejnym jmenem a typem pod stejnym
        predkem a pouze s~jinymi resourcy
    if
        (checkedVertex.getProperty(NodeProperty.NODE_NAME.t()).equals(name)
        && context.getMapDbMmIdToMmNode()
            .get(checkedVertex.getProperty(NodeProperty.METAMODEL_NODE_ID.t()))
       .getModelEquivalentsIds().contains(metamodelNode.getDbId())) {
        notYetExist = false;
        DbNode node = new DbNode(checkedVertex.getId(), metamodelNode);
        context.getMapLocalIdToDbIdAndDbMmId().put(nodeId, node);
        // kontrola, zda nebyl node pridan v~ramci jednoho merge
        if (!RevisionUtils.isNodeInRevisionInterval(checkedVertex, new
            RevisionInterval(clientRevision)))
            RevisionUtils.setNodeTransactionEnd(checkedVertex,
                clientRevision);
        break;
    }
}

if (!notYetExist)
    return ResultType.ALREADY_EXIST;
Vertex resource = GraphOperation.getResource(parentVertex);

if (metamodelNode.getModelEquivalentsIds().size() == 1) {
    // ... nema zadne ekvivalenty - jen sebe -> vytvoreni noveho uzlu
    return ResultType.NEW_OBJECT;
```

Následně jsou dle metamodelu nalezeny všechny modely, které mohou být ekvivalentní. Pro každý možný model se vystoupá do prvního předka, který má taktéž možnou ekvivalenci do tohoto modelu. Tato funkcionalita je v metodě *getAncestor* viz níže.

```
/**
 * @param node uzel z~ktereho zaciname traverzovat
 * @param modelName model po kterem traversujeme nahoru
 * @param destinationModel model s~kterym ma mit uzel ekvivalenci
 * @param context
 * @param skipFirst hodnota zda se ma v~prvnim uzlu (node) hledat
     ekvivalence nebo rovnou traverzovat do rodice.
 *                true znamena, ze node je uzel z~ktereho zaciname a
```

```
     ne jiz jeho prvni predek
 *                a je tedy treba preskocit validaci
 * @return vrati null nebo prvniho rodice v~modelu modelName na
     ktereho narazi, ktery muze mit ekvivalenci do destinationModel
 */
private Vertex getAncestor(Vertex node, Integer modelName, Integer
    destinationModel, ProcessorContext context,
        boolean skipFirst) {
    if (!skipFirst) {
        MetamodelNode metamodelNode = context.getMapDbMmIdToMmNode()
                .get(node.getProperty(NodeProperty.METAMODEL_NODE_ID.t()));
        Set<MetamodelNode> ancestorEqualModelNodes =
            metamodelNode.getModelEquivalents();
        for (MetamodelNode ancestorEqualModelNode :
            ancestorEqualModelNodes) {
            if
                (ancestorEqualModelNode.getModelId().equals(destinationModel))
                {
                    return node;
            }
        }
    }

    //traverzujeme do rodice
    Iterator<Vertex> ancestorQuery =
        node.query().labels(EdgeLabel.HAS_PARENT.t()).direction(Direction.OUT)
            .has(EdgeProperty.METAMODEL_NAME.t(),
                modelName).vertices().iterator();
    if (!ancestorQuery.hasNext()) {
        ancestorQuery =
            node.query().labels(EdgeLabel.HAS_RESOURCE.t()).direction(Direction.OUT)
                .has(EdgeProperty.METAMODEL_NAME.t(),
                    modelName).vertices().iterator();
        if (!ancestorQuery.hasNext()) {
            return null;
        }
    }
    return getAncestor(ancestorQuery.next(), modelName,
        destinationModel, context, false);
}
```

Následně již máme principiálně jen dvě možnosti - 2 algoritmy. V algoritmu *UpDown* můžeme z tohoto uzlu traverzovat dolů po všech dětech do všech možných uzlů typu který hledáme. Traverzovací dotaz lze upravit tak, že rovnou nalezne uzel s daným jménem i typem (pokud existuje). Toto řešení má smysl jen pokud graf metamodelu bude strom, protože víme cestu po které se chceme vydat. Toto by se alternativně dalo vyřešit hranou vždy mezi dvojicí uzlů, které jsou ve stejném podstromě, oba mají ekvivalenci do stejné hierar-

chie a na cestě mezi nimi neexistuje žádný uzel, který by předchozí podmínky splňoval. Toto řešení by však mohlo vytvářet velké množství hran do jednoho uzlu. A to by byl případ na mnoha místech. To by se dále muselo řešit *Vertex-Centric* indexem a i s ním by byla rychlost algoritmu velice spekulativní. Tuto variantu proto zavrhneme a soustředíme se na prvně zmíněnou variantu. Jak již bylo zmíněno, tak se zde pracuje s předem danou cestou. Tato cesta odpovídá typům uzlů v metamodelu jak jdou až do hledaného uzlu kde by měla být možná ekvivalence. Všechny cesty jsou *lazy* a po prvním načtení jsou uloženy do paměti v příslušném *MetamodelNode*. Níže je algoritmus hledání všech shodných uzlů - měl by být maximálně jeden. Zajímavé je zde využití *GremlinPipeline*, které vytváří dotaz, který se provede nad databází. Bez tohoto zápisu by muselo dojít k načtení uzlů do paměti a provedení dotazů.

```
//cesta dolu k~uzlu ktery muze byt stejny na merge - poskladani dotazu
GremlinPipeline<Vertex, Vertex> pipe = new GremlinPipeline<Vertex,
    Vertex>();
pipe = pipe.start(ancestor);
String label = EdgeLabel.HAS_PARENT.t();
if (ancestor.getProperty(NodeProperty.RESOURCE_NAME.t()) != null) {
    label = EdgeLabel.HAS_RESOURCE.t();
}
for (int i = 0; i < reversedPath.size(); i++) {
    Object item = reversedPath.get(reversedPath.size() - 1 - i);
    if (i == reversedPath.size() - 1) {//posledniho se zeptame navic
        na jmeno
        pipe = pipe.inE(label).has(EdgeProperty.METAMODEL_NODE_ID.t(),
            item)
                .has(EdgeProperty.CHILD_NAME.t(),
                    GraphOperation.processValueForChildName(name)).outV();
    } else {//jinak jen na metamodel
        pipe = pipe.inE(label).has(EdgeProperty.METAMODEL_NODE_ID.t(),
            item).outV();
    }
    label = EdgeLabel.HAS_PARENT.t();
}
if (pipe.iterator().hasNext()) {
    return pipe.iterator().next();
}
return null;
```

Alternativním algoritmem je *ByName*. Ten se nikam z uzlu předka nespouští, ale funguje na opačném principu. Díky indexu jsou nalezeny všechny uzly stejného jména a typu a z nich je následně pomocí metody *getAncestor* nalezen předek. Pokud se předek shoduje s předkem vkládaného uzlu, tak jsme nalezli shodný uzel do kterého muže být vkládaný uzel sloučen. Výhodou tohoto algoritmu je jeho imunnost vůči nestromovým metamodelům. Kód tohoto algoritmu je níže.

```
//nalezeni vsech uzlu stejneho jmena a jednoho z~povolenych typu
    ekvivalenci
//s vyuzitim Indexu ktery by mel byt na uzlu
Iterator<Vertex> sameNameQuery =
    context.getDbTransaction().getVertices(NodeProperty.NODE_NAME.t(),
    name)
        .iterator();
List<Vertex> equalVertexesList = new LinkedList<Vertex>();
while (sameNameQuery.hasNext()) {
    //zkontrolujeme cestu nahoru. Node property uzlu nekontrolujeme
        protoze je obsazena v~hrane, stejne tak model
    Vertex next = sameNameQuery.next();
    Vertex possibleEqualAncestor = getAncestor(next,
        possibleModel.getModelId(), modelName, context, true);
    if (ancestor.equals(possibleEqualAncestor)) {
        return next;
    }
}
return null;
```

V obou případech je v případě úspěchu po provedení algoritmu provedeno vložení hrany k rodiči z uzlu, který byl nalezen jako shodný.

Bez příslušných výkonnostních testů nelze označit rychlejší z těchto dvou algoritmů. Osobně se domnívám, že pro každý algoritmus bude existovat složení metamodelu a dat takové, že bude rychlejší než ten druhý.

5.7 Možné směry dalšího vývoje

Jak již bylo řečeno, tak v současné době všechny hierarchie sdílí společnou revizi. Alternativa k tomuto řešení je mít různé revize pro každou hierarchii. Z toho by logicky mohl vznikat nesoulad v tom, že k jedné revizi v libovolném modelu by v jiném modelu mohlo být více revizí. Díky tomu vznikají nové výzvy na logiku traverzování, vkládání a zobrazování dat.

Algoritmicky je tato změna poměrně jasná. Kořen revizí RevisionRoot má na sobě hrany k uzlům, které mají na sobě datum vytvoření revize. Hrana k tomuto uzlu obsahuje číslo revize a label označující, že se jedná o hranu k reviznímu uzlu. Nyní by bylo navíc potřeba každé revizi ke každému modelu vytvořit nový uzel a na hraně k němu by navíc byl název modelu. Z časového hlediska by došlo k jistému zpomalení, protože by se muselo dohledávat v jaké revizi daný model zrovna je.

Mnoho otázek se nabízí v řešení uživatelského rozhraní. Při nabízení přechodu do jiné hierarchie by se nějak měly zobrazovat nejen všechny hierarchie, ale i všechny revize, které spadají do rozsahu. Představme si nějakou hierarchii, která se nebude prakticky měnit anebo jen ve velkých časových intervalech v řádu měsíců. Jiné hierarchie, se kterými by byl tento model v ekvivalenci

mohou mít novou revizi každý týden. V tom případě by uživatel dostal na výběr velké množství revizí, do kterých by mohl přejít, což by pro něj mohlo být nepřehledné. Na druhé straně se nabízí možnost mu nabídnout třeba jen poslední revizi, ve které jsou oba modely společné. V tom případě by na druhou stranu bylo potřeba mu nějak nabídnout i všechny ostatní, ale tak, aby ho to neomezovalo.

Kapitola 6

Výkonnostní testování

6.1 Cíle

Tato kapitola pojednává o testování rychlosti řešení. Porovnávat se budou dvě řešení vkládání a původní algoritmus. Vkládat se budou jen datové a zdrojové uzly. Snažíme se nalézt hodnotu zpomalení vkládání. Z povahy změněného algoritmu je pro nás důležité jen vkládání datových uzlů. Pokud by se vkládaly i jiné datové uzly, tak by docházelo ke zbytečnému šumu.

6.2 Testovací prostředí

Níže uvedené testy byly spuštěny na notebooku s parametry dle tabulky 6.1. Při testech byly spuštěny jen nutné programy pro běh operačního systému.

6.3 Testovací data

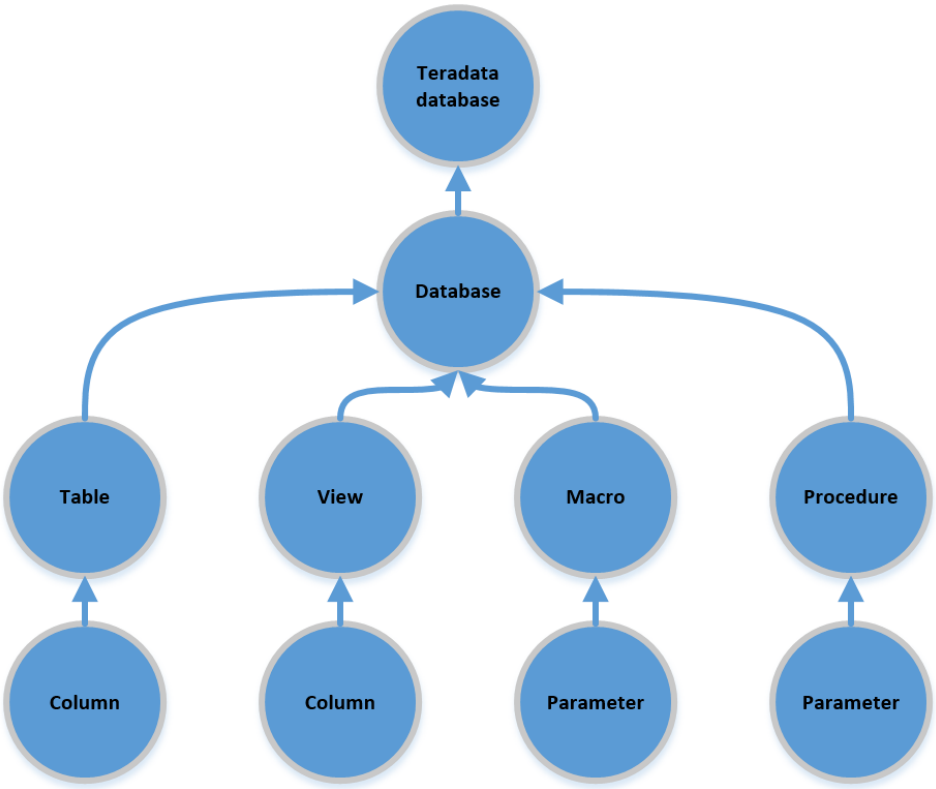
Pro testování byl použit dataset nejmenovaného telefonního operátora. Tento dataset obsahuje okolo milionu a půl datových uzlů. Velikost datasetu odpovídá limitům českých společností a nepředpokládá se, že by některá mohla mít řádově vetší množství databází.

Data obsahují devět hierarchií dle použitých technologií. Například se je jedná o Teradata, Teradata-DDL, Oracle, Oracle-external, MSSQL a File-

Notebook	Dell Latitude E6540
Operační systém	Windows 7 Profesional, Service Pack 1 (64-bit)
Procesor	Intel(R) Core(TM) i7-4810MQ CPU 2.80GHz
Paměť RAM	8 GB
SSD disk	ano

Tabulka 6.1: Parametry notebooku použitého k testování

system. Největší je Teradata s jedním milionem uzlů. Z datasetu se oddělily všechny hierarchie zvlášť do souborů a odfiltrovaly uzly, které mají zdrojové uzly v jiné hierarchii než svého rodiče. Dále se budeme zabývat jen hierarchií Teradata. Vyextraktovaný metamodel z dat je zobrazen na diagramu 6.1

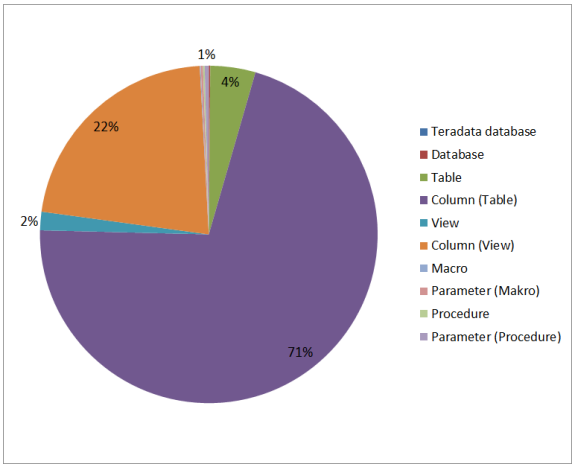


Obrázek 6.1: Metamodel teradata databáze

Pro lepší představu jsem ve své práci vytvořil několik statistik o stavu dat. Na diagramu 6.2 je vidět procentuální rozložení typů uzlů. V tabulce 6.2 jsou statistiky počtu uzlů a větvení jednotlivých typů uzlů tj. počet dětí. U duplicitních názvů jsou v závorce jména rodičovských uzlů. Na grafech 6.3 a 6.4 jsou počty uzlů dle počtu dětí pro typ uzlu table. Na grafu 6.5 jsou počty uzlů dle počtu dětí pro typ uzlu view.

Typ uzlu	Počet uzlů	Maximální # dětí	Průměrný # dětí
Teradata database	1	1 390	1 390
Database	1 390	5 507	45,2
Table	42 950	578	16,4
Column (Table)	705 295		
View	17 545	585	12,5
Column (View)	218 590		
Macro	879	62	2,5
Parameter (Makro)	2 205		
Procedure	1 566	50	2,5
Parameter (Procedure)	3 900		

Tabulka 6.2: Počty typů uzlů v hierarchii Teradata



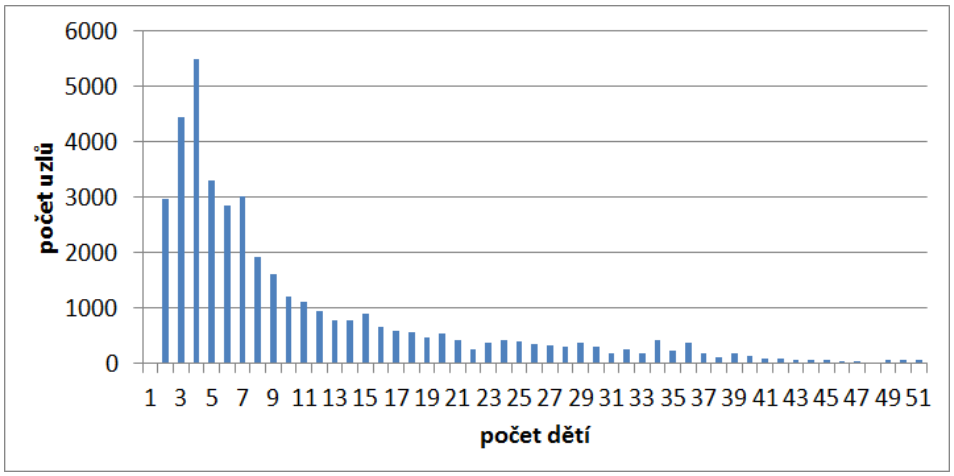
Obrázek 6.2: Rozdělení typů uzlů

Pro typ uzlu table platí tato fakta:

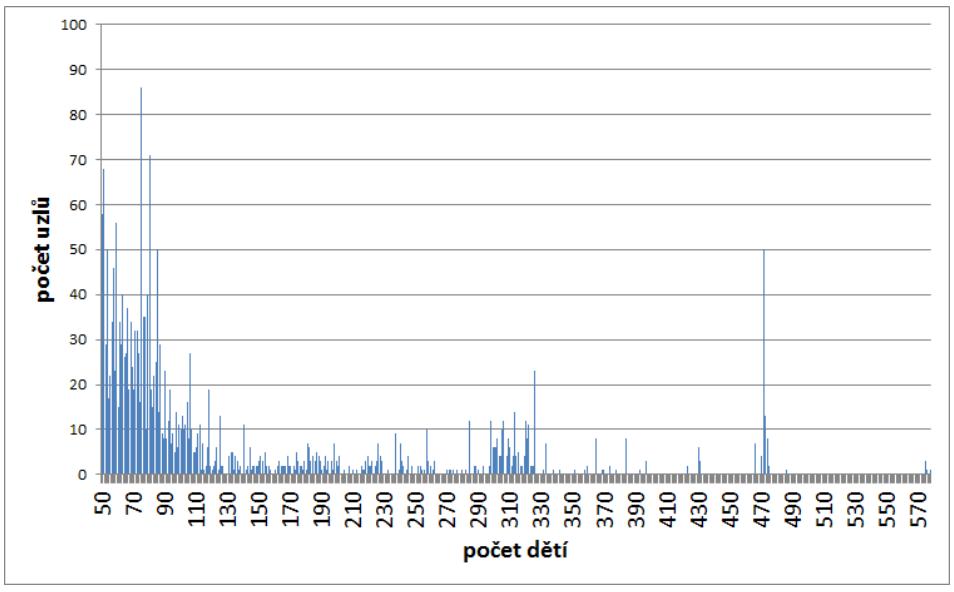
- více jak 100 dětí má 839 uzlů,
- více jak 200 dětí má 454 uzlů,
- maximum je 578,
- 472 dětí má 50 tabulek.

Pro typ uzlu view platí tato fakta:

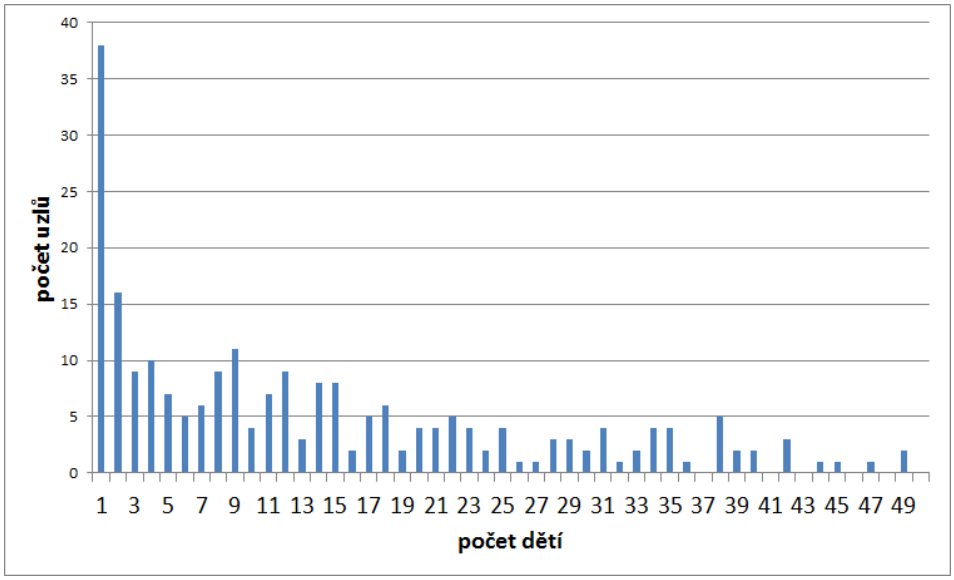
- více jak 100 dětí má 103 uzlů,
- více jak 500 dětí má 29 uzlů,



Obrázek 6.3: Počty uzlů dle počtu dětí pro typ uzlu table



Obrázek 6.4: Počty uzlů dle počtu dětí pro typ uzlu table



Obrázek 6.5: Počty uzlů dle počtu dětí pro typ uzlu view

- maximum je 5508,
- 1014 nemá děti.

Jak je vidět, existují tabulky i views s velmi velkým počtem sloupečků. Opravdu překvapující je view s 5508 sloupečky. Obecně pro grafovou databázi a vyhledávání v ní je příznivý malý počet dětí mezi uzly. Tj. malý počet hran vycházejících z uzlu. Ze dvou naprogramovaných algoritmů bude tímto velkým množstvím vycházejících hran ovlivněn algoritmus *UpDown*.

Naopak algoritmus *ByName* bude ovlivněn počtem stejně pojmenovaných uzlů. Bylo naměřeno, že maxilmální počet uzlů se stejným jménem a typem je 18 635. Což odpovídá 2% uzlů typu sloupeček v tabulce. Medián počtu stejných jmen v celé hierarchii byl naměřen 2.

U algoritmu *ByName* by se mohl použít Lucene index. V práci není otestován, protože Lucene indexuje až po databázovém commitu dat. Tento problém by se nejspíše dal vyřešit kombinací použití vyhledávání s Lucene a uložením dat, která ještě nejsou commitnutá do paměti programu.

6.4 Provedené měření

6.4.1 Porovnání s původním algoritmem

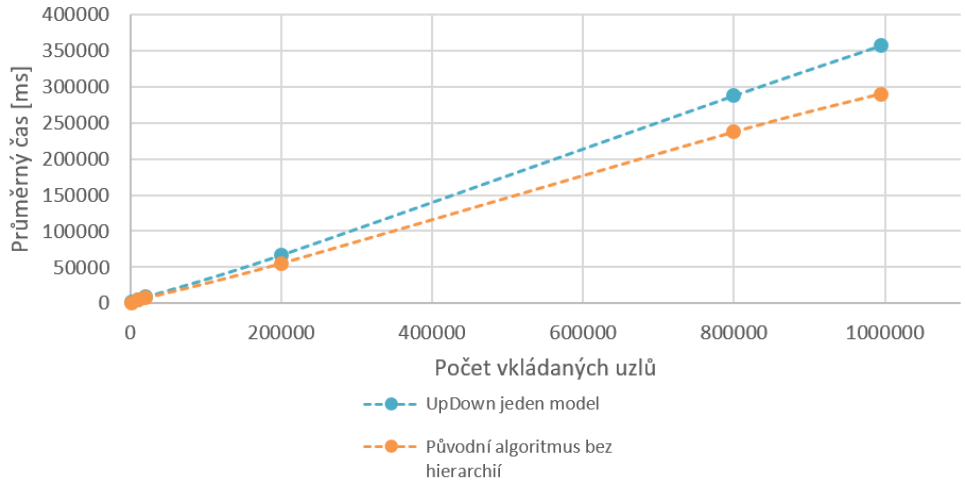
Rozdíl mezi původní algoritmem, který nerozeznává hierarchie a hierarchickým je v režii, kterou správa hierarchií vyžaduje. Tj. rozlišování typu modelu a typu uzlů dle metamodelu. Režie by principiálně neměla být velká. Toto

Počet uzlů	2 000	10 000	20 000	200 000	800 000	994 321
Původní algoritmus	1 295	4 684	7 371	55 457	237 069	289 754
UpDown bez indexů	1 633	5 497	8 991	67 157	287 476	356 518
Zpomalení [%]	21%	15%	18%	17%	18%	19%

Tabulka 6.3: Průměrné naměřené hodnoty pro vložení jedné hierarchie.

měření je jediné, které je v rámci funkcionality vlastně identické vůči původnímu algoritmu. V dalších měřeních už bude porovnávána stávající verze se sjednocováním modelů, což logicky musí trvat delší dobu. Vkládaná data se lišila jen o číslo modelu a vkládaný metamodel.

V tabulce 6.3 jsou vidět naměřené hodnoty pro původní algoritmus a *UpDown* bez indexů. Algoritmus *UpDown* nepotřebuje ke svému fungování nijak speciálně definované indexy pro jméno uzlu. Měření probíhalo nad datasetem Teradata - 994321 uzlů. Zpomalení oproti původnímu algoritmu je okolo 20% a nedá se říci, že by se výrazně měnilo. Část tohoto rozdílu bude způsobena zvětšením databáze o nové atributy. Zpomalení jako takové je v tomto případě rozhodně akceptovatelné. Výsledný graf pro tato data je zobrazen na obrázku 6.6.



Obrázek 6.6: Graf průměrných časů vkládání jedné hierarchie

6.5 Více hierarchií - algoritmus UpDown a ByName

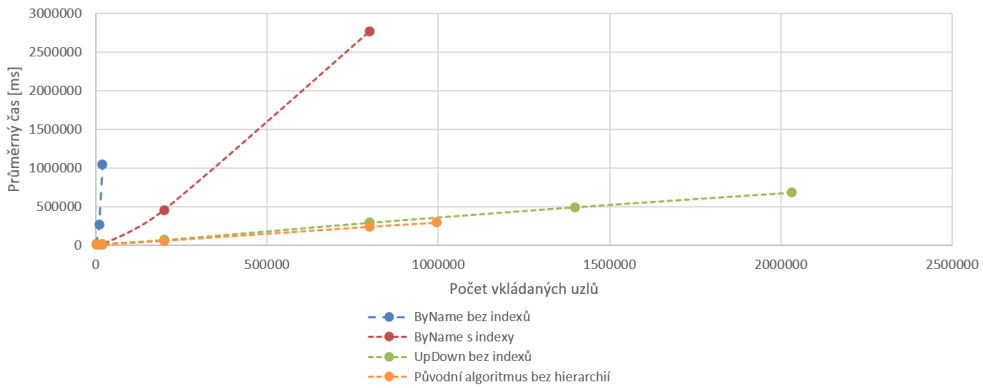
Pro měření více hierarchií byla použila data z diagramu 6.1 pro jednu hierarchii. Pro druhou hierarchii byla zvolena stejná data a mezi uzly *Table* a *Column*

Počet uzlů	10 000	20 000	200 000	800 000	2 031 592
ByName [1]	264 675	1 044 099			
ByName [2]	9 254	9 573	454 741	2 775 614	
UpDown	5 478	15 383	65 919	289 030	679 553
Původní	4 684	7 371	55 457	237 069	
Zpomalení [%]	14%	52%	16%	18%	

Tabulka 6.4: Průměrné naměřené hodnoty pro vložení dvou hierarchií.

byl přidán uzel. Všechny uzly, až na jeden nový, se označily za ekvivalentní. Reálná data pro logický nebo jiný model neexistují. Dá se však předpokládat, že by neobsahovala tak velký počet uzlů ve všech hierarchiích. Tento test, kde jsou všechny hierarchie stejně velké, je tedy více striktní a odráží nejhorší možný scénář.

Pro algoritmus *ByName* byly použity dvě konfigurace. Jedna bez indexu nad jménem uzlu a druhá s indexem shodnosti jména. Variantou k indexu shodnosti jména by byl tokenizovaný index, ale ten by byl zbytečný, protože se vyhledává ekvivalence. Výsledky jsou v tabulce 6.4. *ByName [1]* je bez použití indexu a *ByName [2]* s použitím indexu. Graf vývoje je zobrazen na obrázku 6.7. Pro lepší představu je do tabulky i grafu přidáno porovnání s původním algoritmem (bez hierarchií).



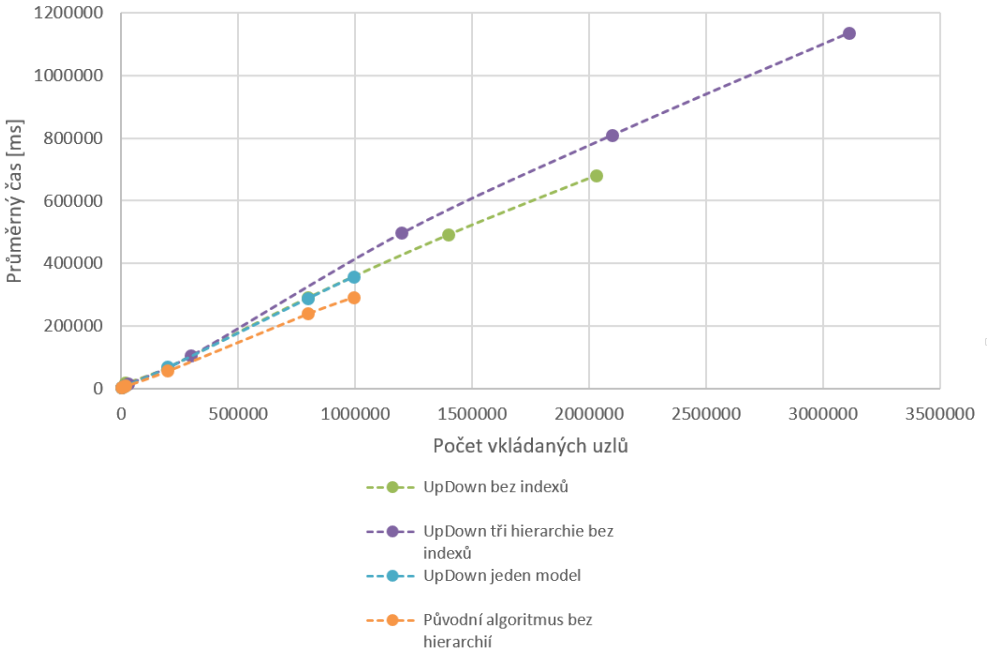
Obrázek 6.7: Graf průměrných časů vkládání dvou hierarchií a bez rozlišení hierarchie

Jak je vidět, tak algoritmus *UpDown* je rychlejší a některá měření pro algoritmus *ByName* nebyla ani měřena z důvodu jasně určeného růstu rychlosti.

Pro zhodnocení více hierarchií byl vytvořen i test se třemi hierarchiemi. Zde byl použit jen algoritmus *UpDown*. Oproti předchozímu testu byl přidán model s prodloužením vzdálenosti mezi *Column* a *Table* o další uzel. Pravidlo ekvivalencí zůstalo zachováno. Výhodou tohoto testu je, že obsahuje více jak tři

Počet uzlů	15 000	30 000	300 000	1 200 000	2 100 000	3 111 810
UpDown	6 230	13 119	103 879	496 122	809 800	1 136 520

Tabulka 6.5: Průměrné naměřené hodnoty pro vložení tří hierarchií algoritmem UpDown.



Obrázek 6.8: Graf průměrných časů vkládání tří hierarchií v porovnání s předešlým testem

miliony datových uzlů. Ty jsou sice v konečném důsledku sloučeny do jednoho milionu, ale zátěž z nových hran na uzlech zůstává. Naměřené hodnoty jsou v tabulce 6.5. Graf zobrazující test spolu s naměřenými hodnotami algoritmu *UpDown* z předchozího měření je na obrázku 6.8. Zpomalení při vkládání více hierarchií je způsobeno větším počtem hran, které jsou kvůli hierarchiím navázány na uzlech. V případě, že bychom měli řidší množství ekvivalence a velikosti modelů, tak se dá očekávat, že se bude tento rozdíl zmenšovat.

6.6 Závěry výsledků měření

Z naměřených dat je zjevné, že algoritmus *UpDown* obstál v požadavcích na rychlost. Zpomalení celkového vkládání uzlů v řádu desítek procent je přijatelné. Mohlo by se zdát, že rychlosti algoritmu pomáhá i množství sjednocených uzlů. Sjednocení uzlů se však nedá označit za čistě kladný jev. Uzlů je

sice méně, ale hran není méně.

Nevýhoda algoritmu *ByName* je z jeho základní definice, a to spoléhání se na nalezení uzlů v databázi dle jména. Tato operace je zjevně mnohem náročnější než traverzování mezi uzly. Dalším faktorem bylo i množství stejně pojmenovaných uzlů, které zpomaluje nalezení toho správného.

Výše uvedené výsledky nejsou ideální pro projekt. Jak již bylo zmíněno, tak sice oba algoritmy splňují definovanou požadovanou množinu funkcionality, ale je to právě algoritmus *ByName*, který má rozšířenou modelovací sílu. V případě, že by byla vyžadována větší modelovací síla než jakou nabízí *UpDown*, tak by bylo nutné algoritmus upravit. Jeden možný algoritmus je naznačen v sekci 5.6.3.

Další alternativou by bylo vynechání dotazu v *TinkerpopPipeline* a prohledávat sousední uzly prohledáváním do šířky. To by vedlo k jistému zpomalení, které by však nemuselo být tak významné jako při použití algoritmu *ByName*.

Tyto úvahy jsou silně závislé na reálných datech. Pokud by bylo množství uzlů v jiných hierarchiích řádově menší, než je v současných datech, tak by bylo možné využít i algoritmus *ByName*.

Řešením pro další rozvoj může být i hybridní algoritmus, kdy by se již v metamodelu mohlo určit, které uzly mají být sjednocovány algoritmem *ByName* a které algoritmem *UpDown*. Tím by se teoreticky mohla vyřešit jak výkonnost, tak modelovací síla algoritmů.

Závěr

Cílem této práce bylo zjistit možnosti změny databázového systému projektu Manta a následně navrhnout, implementovat a otestovat paralelní hierarchie pro zvolený databázový systém. Případná změna by mohla ovlivnit návrhová rozhodnutí, která mohou být specifická v každé grafové databázi. Z analýzy jiných databázových systémů vyplynulo, že v současné době neexistuje přesvědčivě lepší open-source grafová databáze než je Titan. Bylo však určeno, které databázové systémy by mohly být potencionálně vhodné v budoucnosti.

Paralelní hierarchie jsou různé modely dat, které mají uzly, o kterých se ví, že jsou sémanticky stejné, ale zatím nebyla možnost tuto shodnost určit. Pokud jsme schopni určit, které uzly jsou stejné, tak můžeme snadno traverzovat mezi hierarchiemi po ekvivalentních uzlech. Prvně tedy bylo definováno pravidlo shodnosti uzlů nutné k vytvoření heuristiky pro jejich vkládání. Heuristiku nazýváme metamodel. Ten obsahuje strom vztahů mezi typy datových uzlů, které jsou vkládány do databáze. Díky tomu je zaručen jednotný a efektivní přístup k nalezení ekvivalentních uzlů v jiných hierarchiích. Navíc byla definována rozšířená definice metamodelu, která podporuje i nestromové struktury dat. Protože není jisté, že lze efektivně hledat ekvivalence mezi dvěma a více grafy, které nemají stromovou strukturu, tak této problematice byl kladen okrajový zájem.

Byly popsány dvě varianty jak ukládat uzly hierarchií. Zvolena byla varianta, která je citlivější k množství dat, ale nemá takové možnosti rozdělení databáze do více strojů. To však vzhledem k současným velikostem dat není podstatný faktor při rozhodování.

V implementaci byly navrženy a naprogramovány dva algoritmy, algoritmus *UpDown* a *ByName*. *UpDown* je důležitější, efektivně využívá nástrojů grafové databáze při hledání v ekvivalentech uzlů. Naproti tomu *ByName* vznikl jako ukázka možností, které rozšířená definice metamodelu nabízí.

Dále byla provedena analýza reálných dat a vybrána největší hierarchie dat, která obsahuje okolo jednoho milionu datových uzlů. Z té byla vytvořena testovací data víceméně shodných hierarchií, na kterých byly provedeny vý-

konnostní testy. Testovací hierarchie byly zvoleny o stejné velikosti, protože horší scénář nemůže nastat. Velikost vkládaných dat byla až o velikosti tří milionů datových uzlů. Naměřené zpomalení v lepším z algoritmů (*UpDown*) pro vkládání grafu o dvou stejně velkých hierarchiích je dvacet procent. To se jistě dá považovat za dobrý výsledek.

Náměty pro další rozvoj

Řešení hierarchií respektovalo současné použití revizí. S příchodem hierarchií by se však mohlo začít uvažovat o různých revizích pro různé hierarchie. Pokud by byla různá čísla revizí v různých hierarchiích, tak se zde nabízí množství zajímavých otázek pro řešení přechodů mezi hierarchiemi a pro zachování efektivity vkládání uzlů.

Pro budoucí rozvoj metamodelu by bylo dobré směřovat budoucí práce na rozšířenou definici metamodelu a na potřeby k její efektivní implementaci. U algoritmu *UpDown* se nabízí například zrušení použití dotazovacího *TinkerPopPipeline* a použití ručního procházení do šířky. Je otázkou, jak by takové řešení bylo efektivní, ale po dalších úpravách by umožnilo algoritmu použít rozšířenou definici metamodelu. Zajímavý by byl i další rozvoj metamodelu směrem k částečně uživatelsky definovaným ekvivalentním uzlům.

Vzhledemk tomu, že obor grafových databází se velmi dynamicky rozvíjí a vývoj současného databázového systému byl zastaven, tak se ve střednědobém horizontu asi znova začne nabízet otázka, zda by nebylo lepší přejít na jinou databázi. S touto změnou by musela proběhnout nová validace všech databázových optimalizací, které v projekt obsahuje.

Literatura

[1] Apache Software Foundation : The Apache Cassandra Project. [online], [cit. 18. 7. 2015]. Dostupné z: http://cassandra.apache.org

[2] Stephen Pimentel: The rise of the multimodel database. [online], [cit. 20. 7. 2015]. Dostupné z: http://www.infoworld.com/article/2861579/database/the-rise-of-the-multimodel-database.html

[3] Marko A. Rodriguez: Aurelius Acquired by DataStax. [online], [cit. 20. 7. 2015]. Dostupné z: http://thinkaurelius.com/2015/02/03/aurelius-acquired-by-datastax

[4] Robin Schumacher: DataStax, Graph, and the Move to a Multi-Model Database Platform. [online], [cit. 20. 7. 2015]. Dostupné z: http://www.datastax.com/2015/02/datastax-graph-and-the-move-to-a-multi-model-database-platform

[5] solid IT: DB-Engines Ranking - popularity ranking of graph DBMS. [online], [cit. 20. 7. 2015]. Dostupné z: http://db-engines.com/en/ranking/graph+dbms

[6] IBM: IBM Cognos. [online], [cit. 14. 8. 2015]. Dostupné z: https://www.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.whse.doc/ids_ddi_356.html

[7] thinkaurelius/titan Wiki. [online], [cit. 14. 8. 2015]. Dostupné z: https://github.com/thinkaurelius/titan/wiki/Using-Persistit

[8] Aurelius: Titan and the CAP Theorem. [online], [cit. 14. 8. 2015]. Dostupné z: http://s3.thinkaurelius.com/docs/titan/0.5.0/benefits.html#_titan_and_the_cap_theorem

[9] Marko A. Rodriguez: Educating the Planet with Pearson. [online], [cit. 14. 8. 2015]. Dostupné z: http://thinkaurelius.com/2013/05/13/educating-the-planet-with-pearson

[10] tinkerpop/blueprints Wiki. [online], [cit. 14. 8. 2015]. Dostupné z: https://github.com/tinkerpop/blueprints/wiki

[11] Philip Rathle: Neo4j sharding aspect. [online], [cit. 14. 8. 2015]. Dostupné z: http://stackoverflow.com/a/21566766

[12] John R. Jenson: Neo, Titan & Cassandra. [online], [cit. 14. 8. 2015]. Dostupné z: http://www.slideshare.net/johnrjenson/no-sql-night-2

[13] neo4j-contrib/gremlin-plugin. [online], [cit. 14. 8. 2015]. Dostupné z: https://github.com/neo4j-contrib/gremlin-plugin

[14] Neo4j - Cypher vs Gremlin query language. [online], [cit. 14. 8. 2015]. Dostupné z: http://stackoverflow.com/a/13836585

[15] Neo4j - Cypher vs Gremlin query language. [online], [cit. 14. 8. 2015]. Dostupné z: http://stackoverflow.com/a/15573154

[16] Romiko Derbynew: Gremlin vs Cypher Initial Thoughts Neo4j. [online], [cit. 14. 8. 2015]. Dostupné z: http://romikoderbynew.com/2012/02/22/gremlin-vs-cypher-initial-thoughts-neo4j

[17] OrientDB LTD.: OrientDB vs Neo4j. [online], [cit. 14. 8. 2015]. Dostupné z: http://orientdb.com/orientdb-vs-neo4j

[18] OrientDB LTD.: OrientDB vs MongoDB. [online], [cit. 14. 8. 2015]. Dostupné z: http://orientdb.com/orientdb-vs-mongodb

[19] solid IT: DB-Engines Ranking of Graph DBMS. [online], [cit. 14. 8. 2015]. Dostupné z: http://db-engines.com/en/ranking/graph+dbms

[20] Mary Jo Foley: Microsoft quietly delivers first preview of Graph Engine. [online], [cit. 3. 9. 2015]. Dostupné z: http://www.zdnet.com/article/microsoft-quietly-delivers-first-preview-of-graph-engine

[21] Microsoft Corporation: Microsoft quietly delivers first preview of Graph Engine. [online], [cit. 3. 9. 2015]. Dostupné z: http://www.graphengine.io

[22] Mary Jo Foley: Microsoft drops Dryad; puts its big-data bets on Hadoop. [online], [cit. 3. 9. 2015]. Dostupné z: http://www.zdnet.com/article/microsoft-drops-dryad-puts-its-big-data-bets-on-hadoop

[23] Robin Schumacher: DataStax, Graph, and the Move to a Multi-Model Database Platform. [online], [cit. 3. 9. 2015]. Dostupné z: http://www.datastax.com/2015/02/datastax-graph-and-the-move-to-a-multi-model-database-platform

[24] TIOBE software BV: TIOBE Index for April 2016. [online], [cit. 20. 4. 2015]. Dostupné z: http://www.tiobe.com/tiobe_index

[25] Stack Exchange, Inc.: Developer Survey Results 2016. [online]. Dostupné z: http://stackoverflow.com/research/developer-survey-2016

[26] Pierre Carbonnelle: PYPL PopularitY of Programming Language. [online], [cit. 20. 4. 2015]. Dostupné z: http://pypl.github.io/PYPL.html

[27] Node.js Foundation: Node.js. [online], [cit. 20. 4. 2015]. Dostupné z: https://nodejs.org/en

[28] Google Inc.: AngularJS. [online], [cit. 20. 4. 2015]. Dostupné z: https://angularjs.org

[29] Facebook Inc.: React. [online], [cit. 20. 4. 2015]. Dostupné z: https://facebook.github.io/react

[30] OrientDB LTD: OrientDB. [online], [cit. 20. 4. 2015]. Dostupné z: http://orientdb.com

[31] ArangoDB GmbH: ArangoDB. [online], [cit. 20. 4. 2015]. Dostupné z: https://www.arangodb.com

[32] OpenLink Software: Virtuoso. [online], [cit. 20. 4. 2015]. Dostupné z: http://virtuoso.openlinksw.com

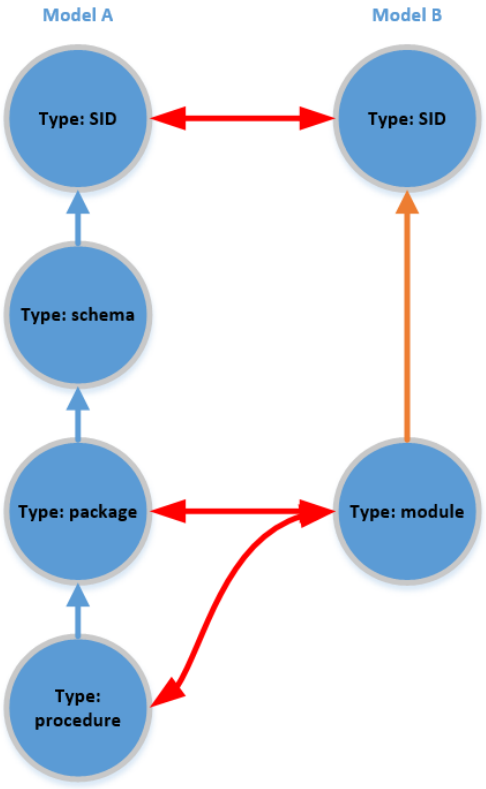
[33] Pivotal Software, Inc: Spring. [online], [cit. 20. 4. 2015]. Dostupné z: https://spring.io

Příloha A

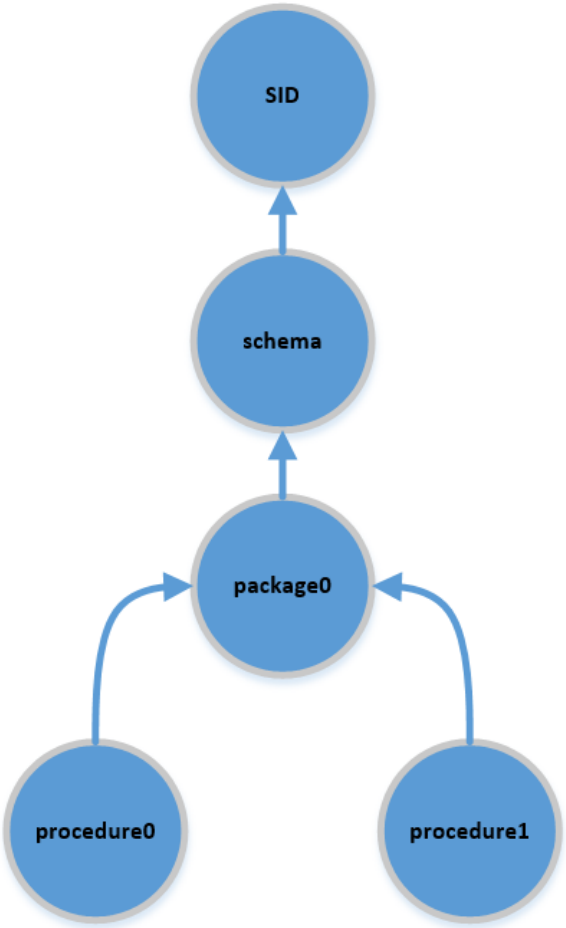
Testovací data

V této příloze je možné nalézt diagramy některých testovacích dat a výsledek po jejich vložení.

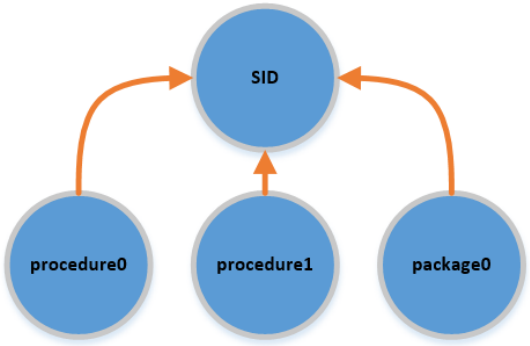
A.1 metamodel12



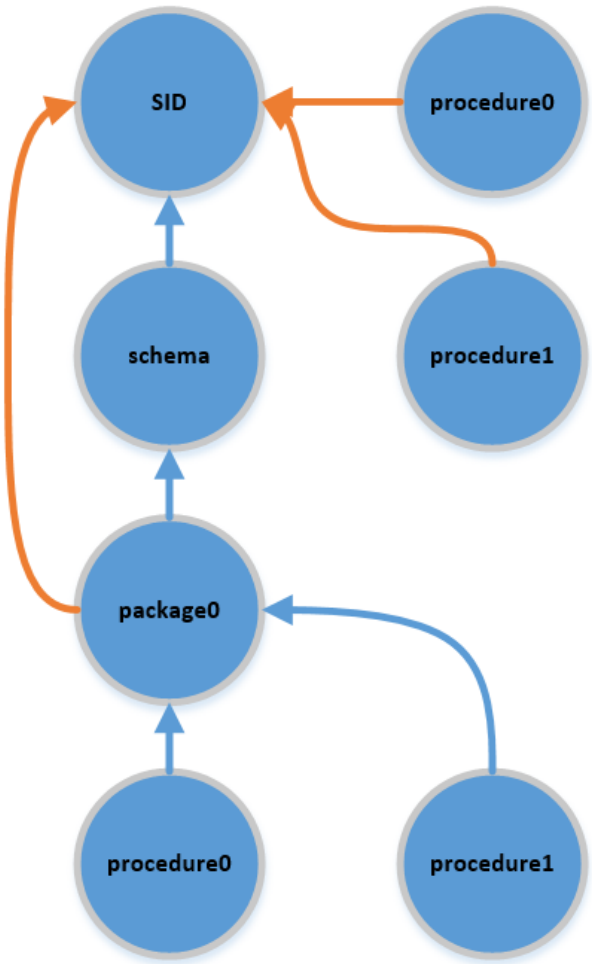
Obrázek A.1: Metamodel ze scénáře metamodel_12



Obrázek A.2: Data modelu A ze scénáře metamodel_12

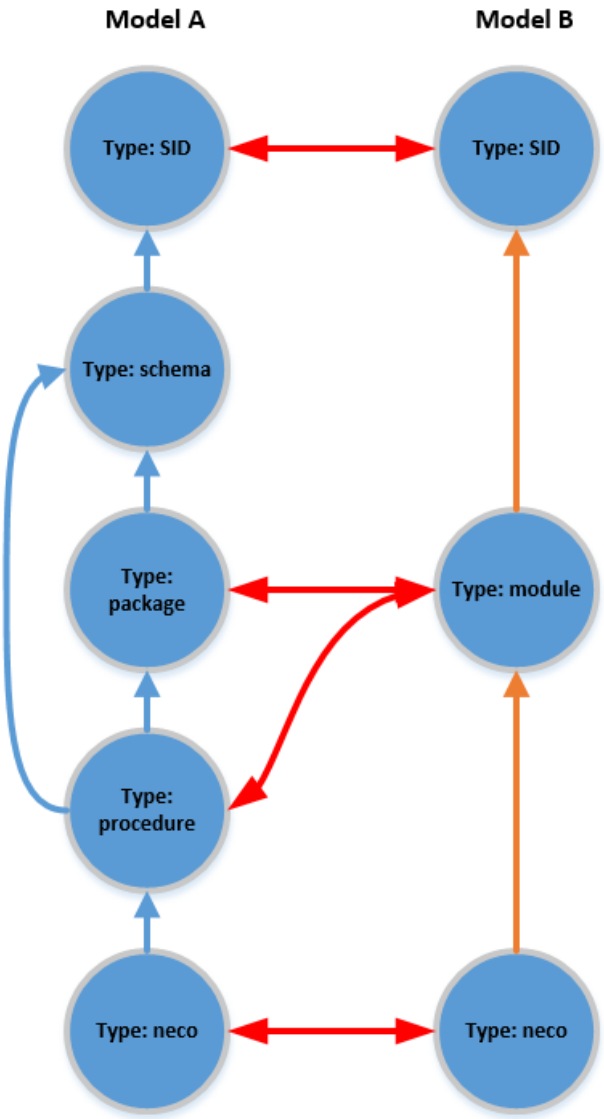


Obrázek A.3: Data modelu B ze scénáře metamodel_12

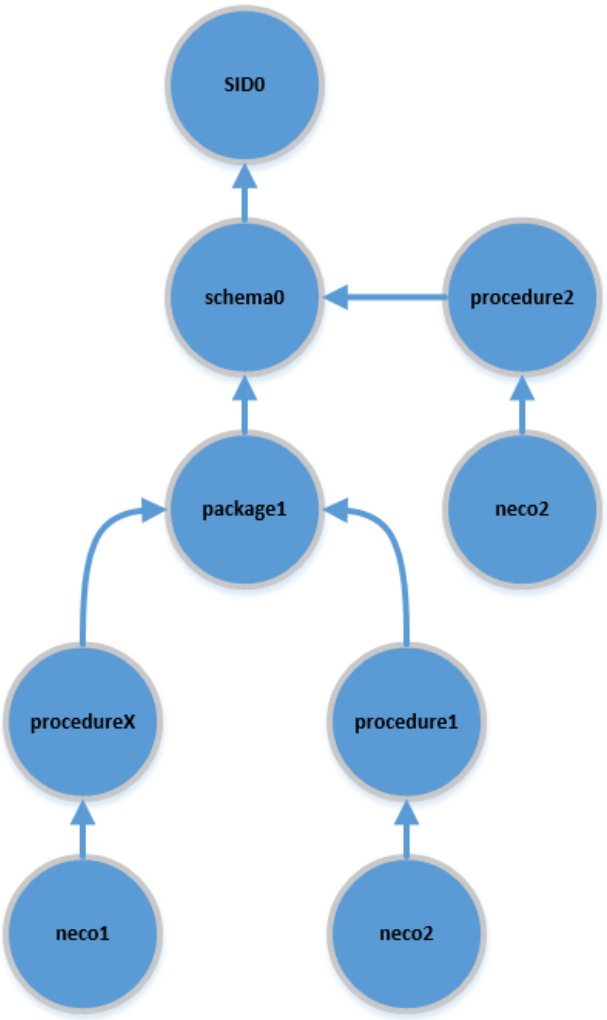


Obrázek A.4: Výsledek po vložení dat ze scénáře metamodel_12

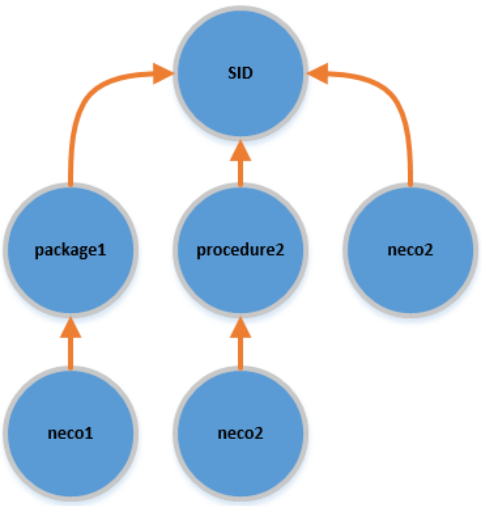
A.2 metamodel13



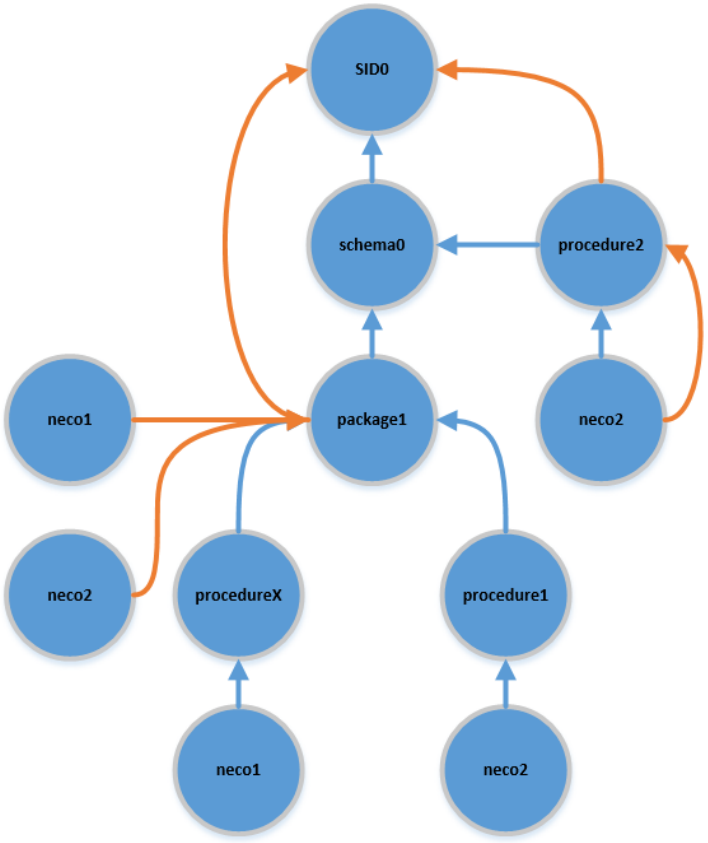
Obrázek A.5: Metamodel ze scénáře metamodel_13



Obrázek A.6: Data modelu A ze scénáře metamodel_13

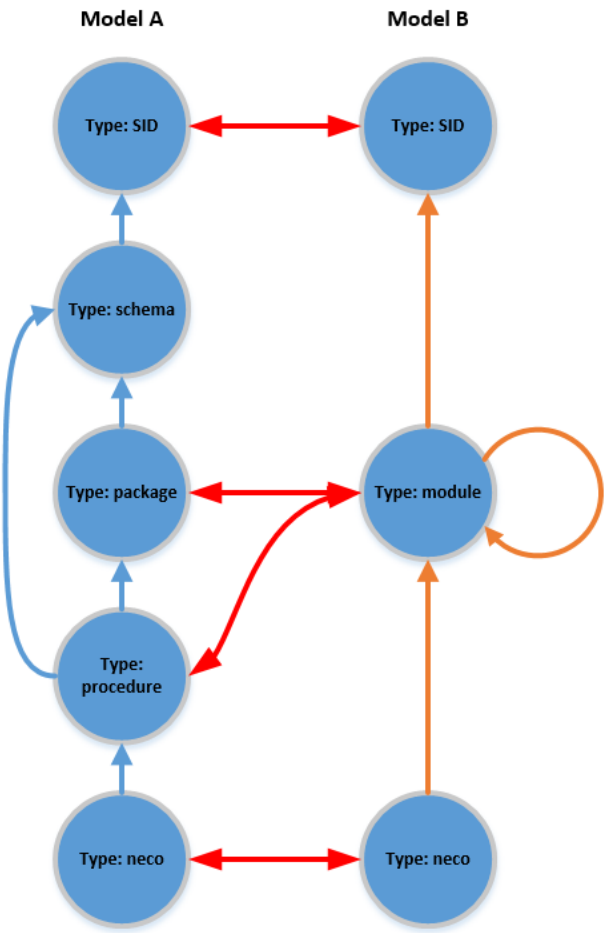


Obrázek A.7: Data modelu B ze scénáře metamodel_13

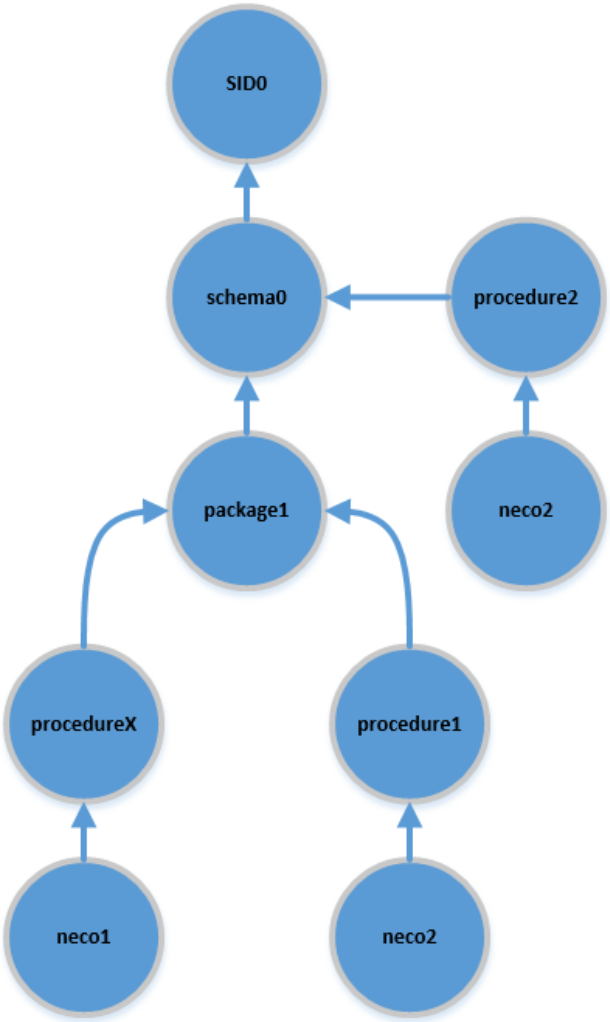


Obrázek A.8: Výsledek po vložení dat ze scénáře metamodel_13

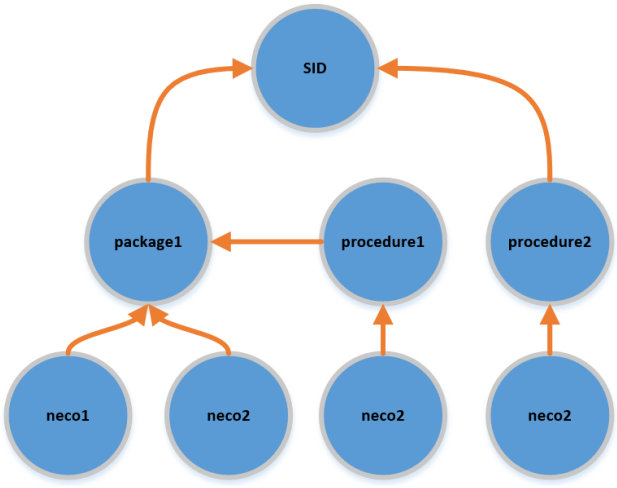
A.3 metamodel14



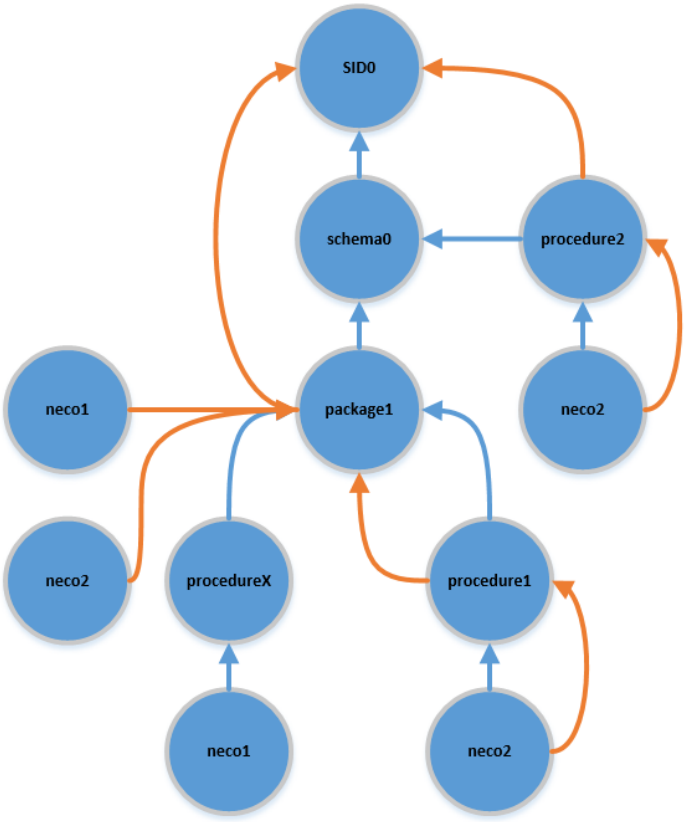
Obrázek A.9: Metamodel ze scénáře metamodel_14



Obrázek A.10: Data modelu A ze scénáře metamodel_14

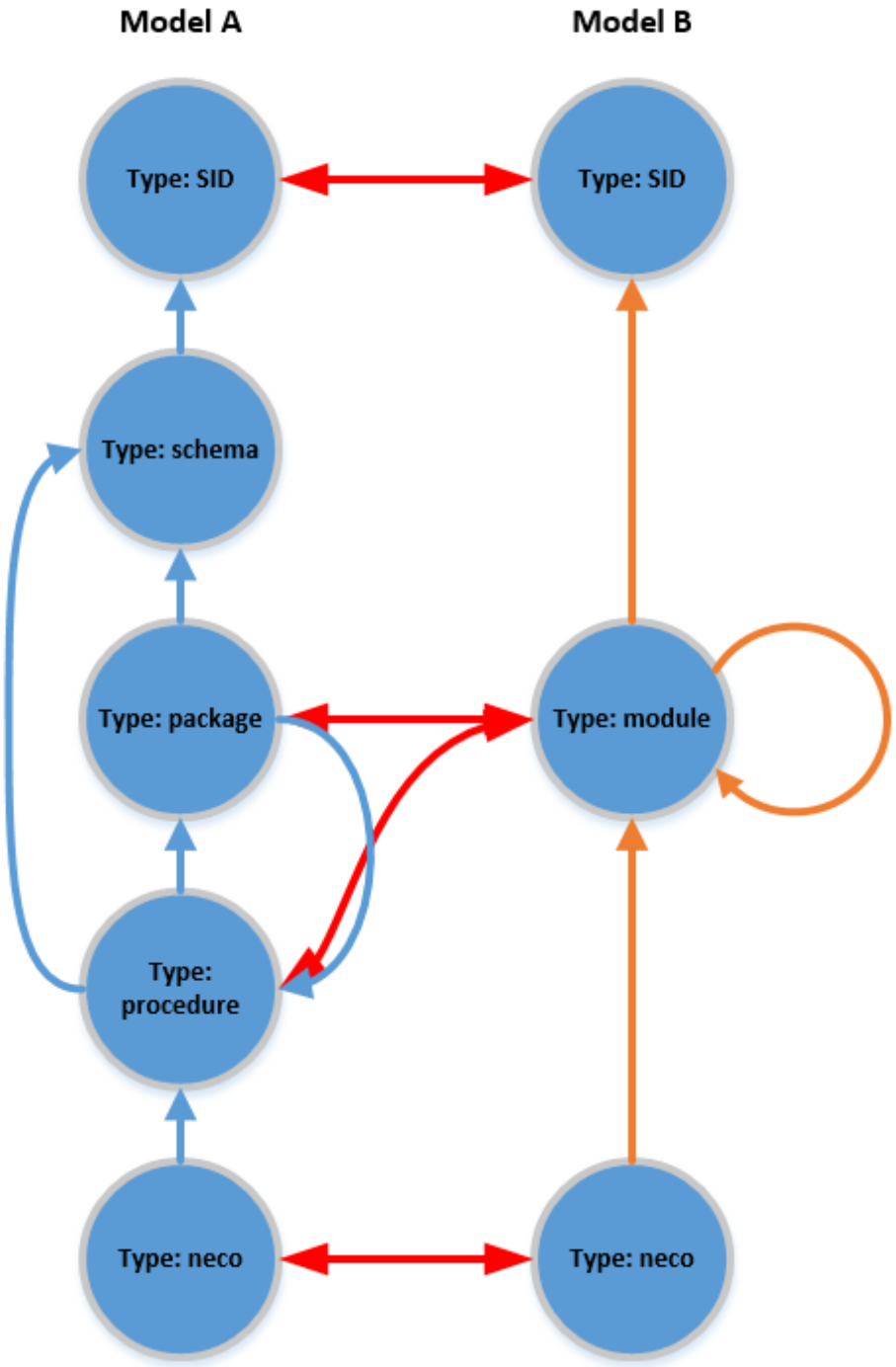


Obrázek A.11: Data modelu B ze scénáře metamodel_14

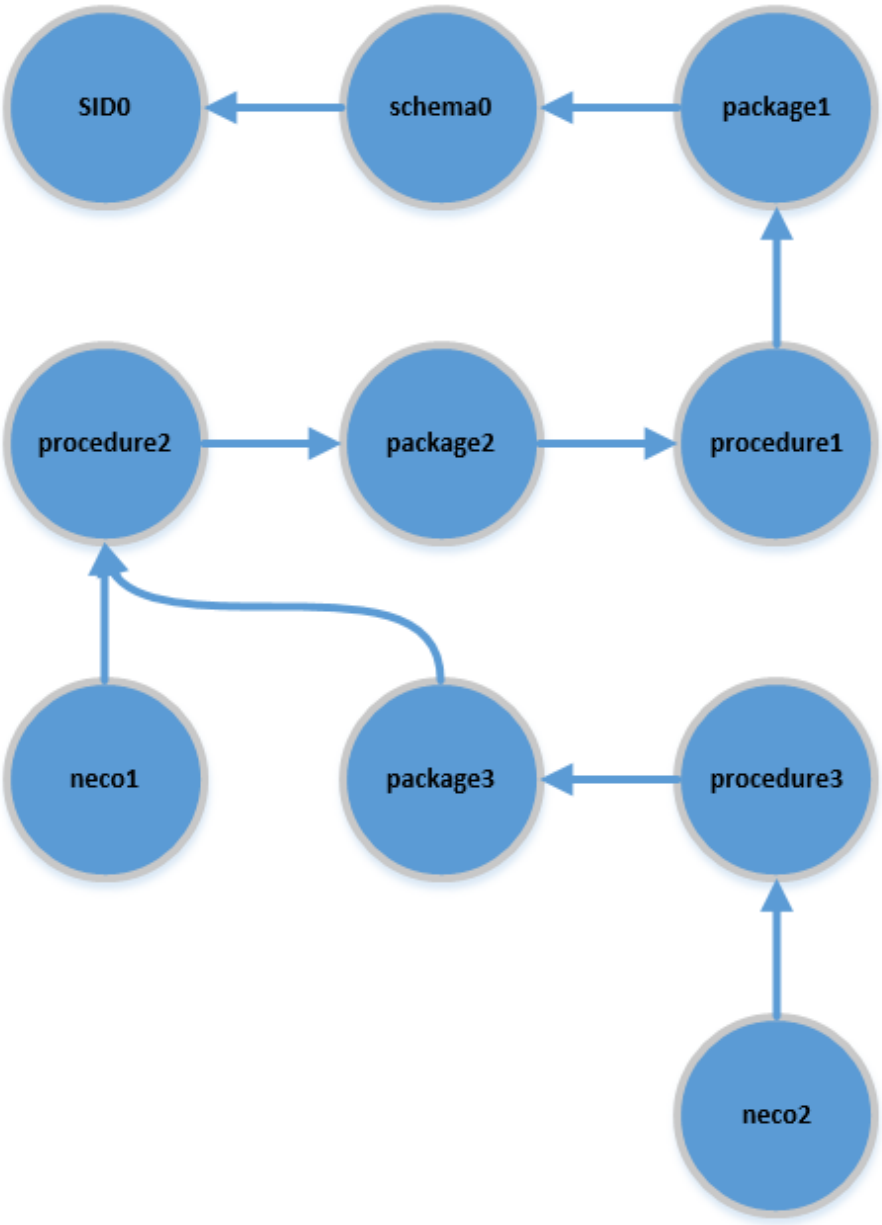


Obrázek A.12: Výsledek po vložení dat ze scénáře metamodel_14

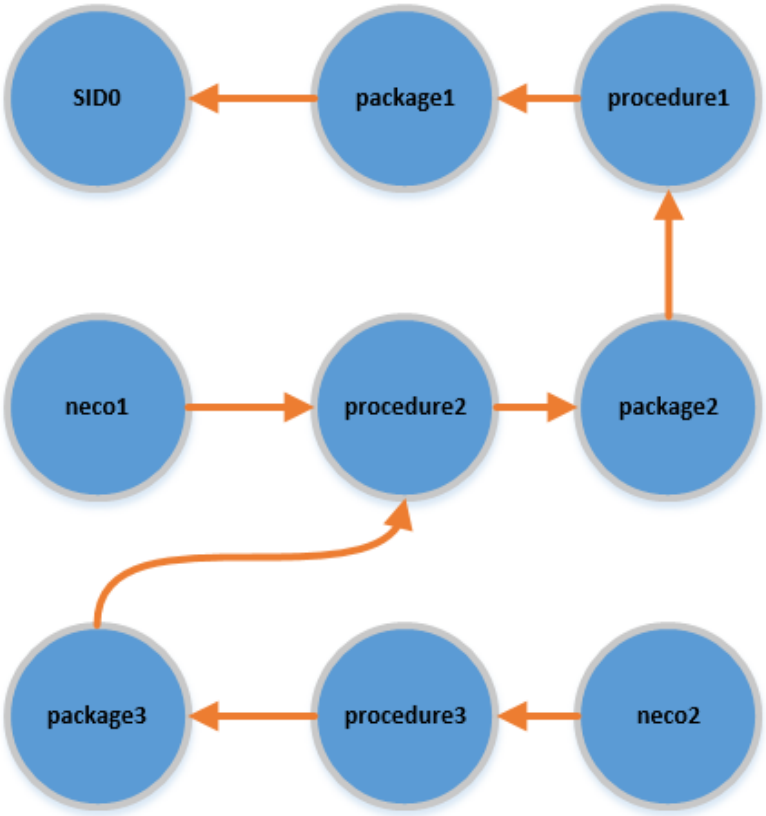
A.4 metamodel15



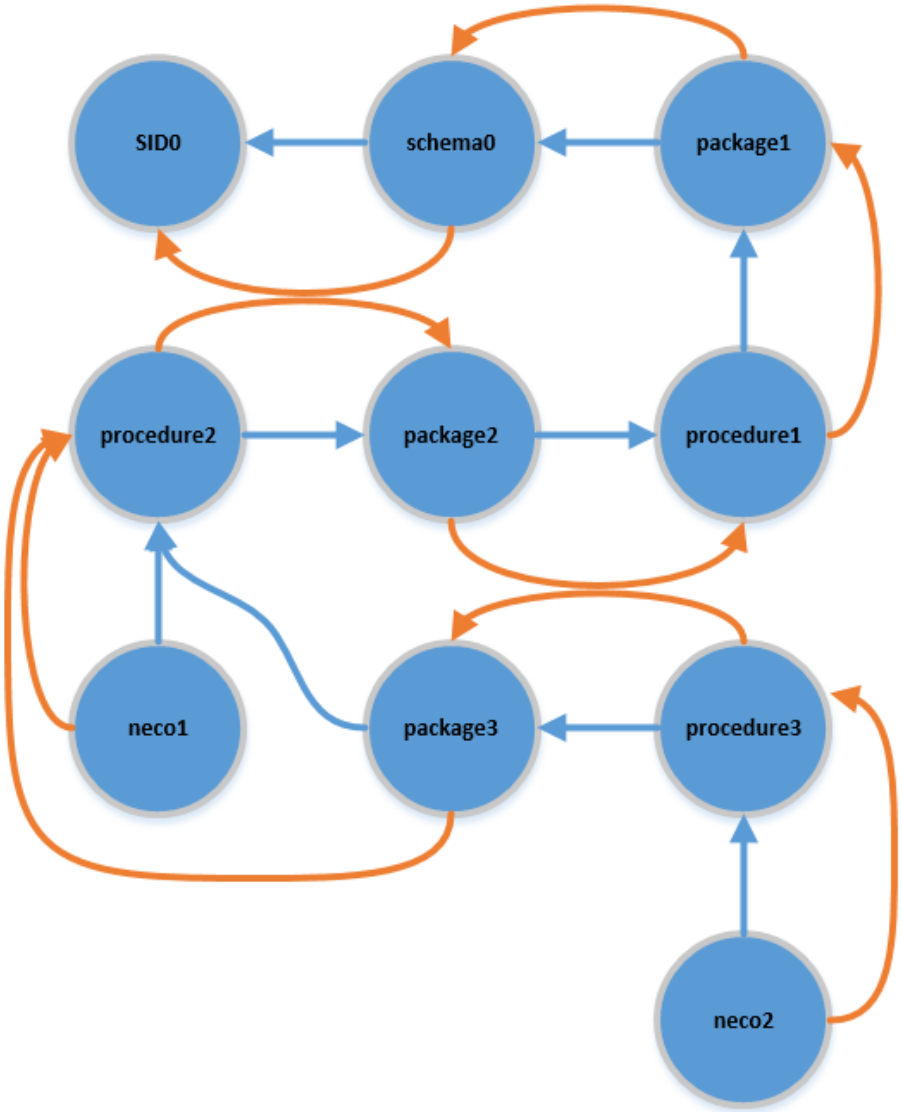
Obrázek A.13: Metamodel ze scénáře metamodel_15



Obrázek A.14: Data modelu A ze scénáře metamodel_15



Obrázek A.15: Data modelu B ze scénáře metamodel_15



Obrázek A.16: Výsledek po vložení dat ze scénáře metamodel_15

Příloha B

Naměřená data

B.1 Původní algoritmus bez hierarchií

Počet uzlů	Čas [ms]
2 000	1 257
2 000	1 283
2 000	1 346

Tabulka B.1: Naměřené hodnoty původního algoritmu pro 2 000 uzlů bez použití indexu na jménu uzlu.

Počet uzlů	Čas [ms]
10 000	4 692
10 000	4 641
10 000	4 718

Tabulka B.2: Naměřené hodnoty původního algoritmu pro 10 000 uzlů bez použití indexu na jménu uzlu.

Počet uzlů	Čas [ms]
20 000	7 193
20 000	7 722
20 000	7 197

Tabulka B.3: Naměřené hodnoty původního algoritmu pro 20 000 uzlů bez použití indexu na jménu uzlu.

Počet uzlů	Čas [ms]
200 000	59 397
200 000	54 137
200 000	52 838

Tabulka B.4: Naměřené hodnoty původního algoritmu pro 200 000 uzlů bez použití indexu na jménu uzlu.

Počet uzlů	Čas [ms]
800 000	243 569
800 000	233 350
800 000	234 287

Tabulka B.5: Naměřené hodnoty původního algoritmu pro 800 000 uzlů bez použití indexu na jménu uzlu.

Počet uzlů	Čas [ms]
994 321	283 238
994 321	303 193
994 321	282 830

Tabulka B.6: Naměřené hodnoty původního algoritmu pro 994 321 uzlů bez použití indexu na jménu uzlu.

B.2 Vkládání jedné hierarchie

Počet uzlů	Čas [ms]
2 000	1 565
2 000	1 566
2 000	1 767

Tabulka B.7: Naměřené hodnoty algoritmu UpDown pro 2 000 uzlů bez použití indexu na jménu uzlu. Použita jedna hierarchie.

Počet uzlů	Čas [ms]
10 000	5 626
10 000	5 683
10 000	5 183

Tabulka B.8: Naměřené hodnoty algoritmu UpDown pro 10000 uzlů bez použití indexu na jménu uzlu. Použita jedna hierarchie.

Počet uzlů	Čas [ms]
20 000	8 764
20 000	8 586
20 000	9 622

Tabulka B.9: Naměřené hodnoty algoritmu UpDown pro 20 000 uzlů bez použití indexu na jménu uzlu. Použita jedna hierarchie.

Počet uzlů	Čas [ms]
200 000	65 962
200 000	67 962
200 000	67 548

Tabulka B.10: Naměřené hodnoty algoritmu UpDown pro 200000 uzlů bez použití indexu na jménu uzlu. Použita jedna hierarchie.

Počet uzlů	Čas [ms]
800 000	284 675
800 000	284 850
800 000	292 902

Tabulka B.11: Naměřené hodnoty algoritmu UpDown pro 800 000 uzlů bez použití indexu na jménu uzlu. Použita jedna hierarchie.

Počet uzlů	Čas [ms]
994 321	347 372
994 321	358 419
994 321	363 763

Tabulka B.12: Naměřené hodnoty algoritmu UpDown pro 994 321 uzlů bez použití indexu na jménu uzlu. Použita jedna hierarchie.

B.3 ByName bez použití indexu na jménu uzlu pro dvě hierarchie

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
1 000	1 000	2 000	11 895
1 000	1 000	2 000	11 020
1 000	1 000	2 000	11 627

Tabulka B.13: Naměřené hodnoty k algoritmu ByName pro 2 000 uzlů bez použití indexu na jménu uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
5 000	5 000	10 000	267 681
5 000	5 000	10 000	259 799
5 000	5 000	10 000	266 547

Tabulka B.14: Naměřené hodnoty k algoritmu ByName pro 10 000 uzlů bez použití indexu na jménu uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
10 000	10 000	20 000	1026 550
10 000	10 000	20 000	1054 312
10 000	10 000	20 000	1051 437

Tabulka B.15: Naměřené hodnoty k algoritmu ByName pro 20 000 uzlů bez použití indexu na jménu uzlu

B.4 ByName s použitím indexu identity řetězce jména uzlu pro dvě hierarchie

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
1 000	1 000	2 000	2 644
1 000	1 000	2 000	2 779
1 000	1 000	2 000	2 612

Tabulka B.16: Naměřené hodnoty k algoritmu ByName pro 2 000 uzlů s použitím indexu identity řetězce jména uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
1 000	1 000	2 000	2 644
1 000	1 000	2 000	2 779
1 000	1 000	2 000	2 612

Tabulka B.17: Naměřené hodnoty k algoritmu ByName pro 2 000 uzlů s použitím indexu identity řetězce jména uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
5 000	5 000	10 000	8 950
5 000	5 000	10 000	9 599
5 000	5 000	10 000	9 213

Tabulka B.18: Naměřené hodnoty k algoritmu ByName pro 10 000 uzlů s použitím indexu identity řetězce jména uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
10 000	10 000	20 000	15 219
10 000	10 000	20 000	15 280
10 000	10 000	20 000	15 650

Tabulka B.19: Naměřené hodnoty k algoritmu ByName pro 20000 uzlů s použitím indexu identity řetězce jména uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
100 000	100 000	200 000	440 099
100 000	100 000	200 000	466 525
100 000	100 000	200 000	457 600

Tabulka B.20: Naměřené hodnoty k algoritmu ByName pro 200 000 uzlů s použitím indexu identity řetězce jména uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
200 000	200 000	400 000	2 740 534
200 000	200 000	400 000	2 736 035
200 000	200 000	400 000	2 850 274

Tabulka B.21: Naměřené hodnoty k algoritmu ByName pro 400 000 uzlů s použitím indexu identity řetězce jména uzlu

B.5 UpDown bez použití indexu na jménu uzlu pro dvě hierarchie

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
1 000	1 000	2 000	1 776
1 000	1 000	2 000	1 715
1 000	1 000	2 000	1 857

Tabulka B.22: Naměřené hodnoty k algoritmu UpDown pro 2 000 uzlů bez použití indexu na jménu uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
5 000	5 000	10 000	5 403
5 000	5 000	10 000	5 574
5 000	5 000	10 000	5 458

Tabulka B.23: Naměřené hodnoty k algoritmu UpDown pro 10 000 uzlů bez použití indexu na jménu uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
10 000	10 000	20 000	8 766
10 000	10 000	20 000	10 321
10 000	10 000	20 000	9 633

Tabulka B.24: Naměřené hodnoty k algoritmu UpDown pro 20 000 uzlů bez použití indexu na jménu uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
100 000	100 000	200 000	66 754
100 000	100 000	200 000	66 083
100 000	100 000	200 000	64 919

Tabulka B.25: Naměřené hodnoty k algoritmu UpDown pro 200000 uzlů bez použití indexu na jménu uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
400 000	400 000	800 000	297 744
400 000	400 000	800 000	284 869
400 000	400 000	800 000	284 476

Tabulka B.26: Naměřené hodnoty k algoritmu UpDown pro 800 000 uzlů bez použití indexu na jménu uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
400 000	400 000	1 400 000	492 083
400 000	400 000	1 400 000	491 771
400 000	400 000	1 400 000	486 203

Tabulka B.27: Naměřené hodnoty k algoritmu UpDown pro 1 400 000 uzlů bez použití indexu na jménu uzlu

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
1 015 796	1 015 796	2 031 592	680 245
1 015 796	1 015 796	2 031 592	680 828
1 015 796	1 015 796	2 031 592	677 586

Tabulka B.28: Naměřené hodnoty k algoritmu UpDown pro 2 031 592 uzlů bez použití indexu na jménu uzlu

B.6 UpDown bez použití indexu na jménu uzlu pro tři hierarchie

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
1 000	2 000	3 000	2 317
1 000	2 000	3 000	2 113
1 000	2 000	3 000	2 139

Tabulka B.29: Naměřené hodnoty k algoritmu UpDown pro 3000 uzlů bez použití indexu na jménu uzlu. Použity 3 hierarchie o stejném počtu uzlů.

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
5 000	10 000	15 000	6 015
5 000	10 000	15 000	5 634
5 000	10 000	15 000	7 040

Tabulka B.30: Naměřené hodnoty k algoritmu UpDown pro 15 000 uzlů bez použití indexu na jménu uzlu. Použity 3 hierarchie o stejném počtu uzlů.

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
10 000	20 000	30 000	13 219
10 000	20 000	30 000	13 556
10 000	20 000	30 000	12 581

Tabulka B.31: Naměřené hodnoty k algoritmu UpDown pro 30 000 uzlů bez použití indexu na jménu uzlu. Použity 3 hierarchie o stejném počtu uzlů.

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
100 000	200 000	300 000	110 946
100 000	200 000	300 000	100 222
100 000	200 000	300 000	100 468

Tabulka B.32: Naměřené hodnoty k algoritmu UpDown pro 300 000 uzlů bez použití indexu na jménu uzlu. Použity 3 hierarchie o stejném počtu uzlů.

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
400 000	800 000	1 200 000	464 431
400 000	800 000	1 200 000	533 334
400 000	800 000	1 200 000	490 601

Tabulka B.33: Naměřené hodnoty k algoritmu UpDown pro 1 200 000 uzlů bez použití indexu na jménu uzlu. Použity 3 hierarchie o stejném počtu uzlů.

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
700 000	1 400 000	2 100 000	826 855
700 000	1 400 000	2 100 000	769 979
700 000	1 400 000	2 100 000	832 566

Tabulka B.34: Naměřené hodnoty k algoritmu UpDown pro 2 100 000 uzlů bez použití indexu na jménu uzlu. Použity 3 hierarchie o stejném počtu uzlů.

Počet uzlů v první hierarchii	Počet slučovaných uzlů	Celkem uzlů	Čas [ms]
1 037 270	2 074 540	3 111 810	1 175 117
1 037 270	2 074 540	3 111 810	1 083 252
1 037 270	2 074 540	3 111 810	1 151 192

Tabulka B.35: Naměřené hodnoty k algoritmu UpDown pro 3 111 810 uzlů bez použití indexu na jménu uzlu. Použity 3 hierarchie o stejném počtu uzlů.

B.7 Průměrné hodnoty

Celkový počet uzlů	Průměrný čas [ms]
2 000	1 295
10 000	4 684
20 000	7 371
200 000	55 457
800 000	237 069
994 321	289 754

Tabulka B.36: Průměrné naměřené hodnoty původního algoritmu bez použití hierarchií a indexu na jméno uzlu.

Celkový počet uzlů	Průměrný čas [ms]
2 000	1 633
10 000	5 497
20 000	8 991
200 000	67 157
800 000	287 476
994 321	356 518

Tabulka B.37: Průměrné naměřené hodnoty algoritmu UpDown pro jeden model bez použití indexu na jméno uzlu.

Celkový počet uzlů	Průměrný čas [ms]
2 000	11 514
10 000	264 676
20 000	1 044 100

Tabulka B.38: Průměrné naměřené hodnoty algoritmu ByName pro dvě hierarchie bez použití indexu na jméno uzlu.

Celkový počet uzlů	Průměrný čas [ms]
2 000	2 678
10 000	9 254
20 000	15 383
200 000	454 741
800 000	2 775 614

Tabulka B.39: Průměrné naměřené hodnoty algoritmu ByName s použitím indexu identity řetězce na jménu uzlu pro dvě hierarchie

Celkový počet uzlů	Průměrný čas [ms]
2 000	1 783
10 000	5 478
20 000	9 573
200 000	65 919
800 000	289 030
1 400 000	490 019
2031592	679553

Tabulka B.40: Průměrné naměřené hodnoty algoritmu UpDown bez použití indexu na jménu uzlu pro dvě hierarchie

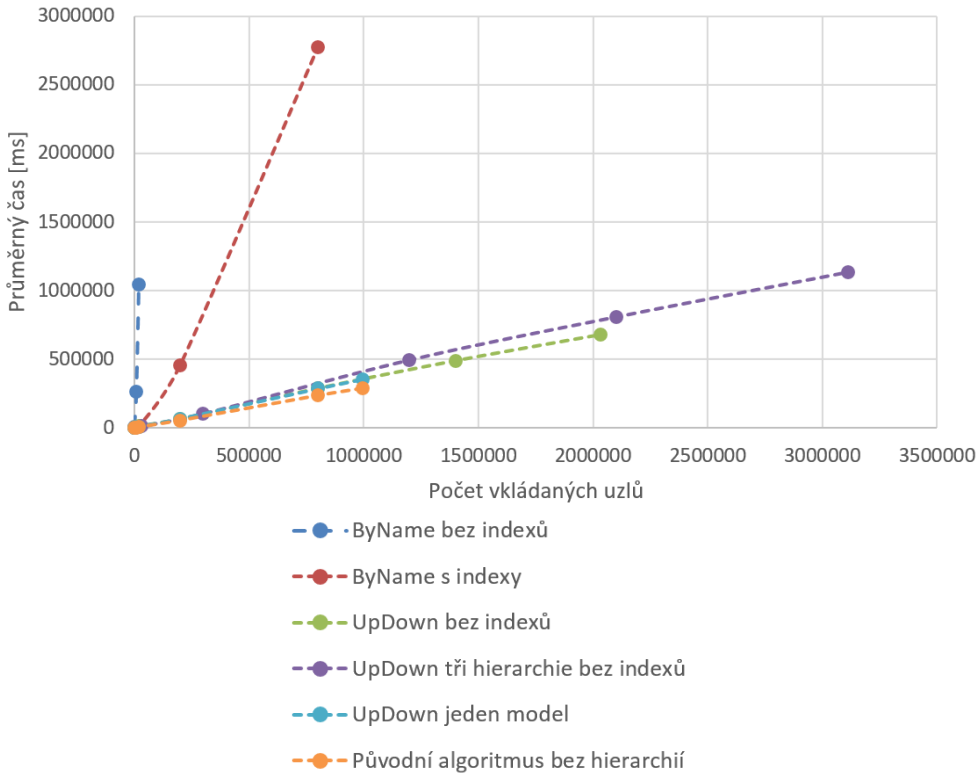
Celkový počet uzlů	Průměrný čas [ms]
3 000	2 190
15 000	6 230
30 000	13 119
300 000	103 879
1 200 000	496 122
2 100 000	809 800
3 111 810	1 136 520

Tabulka B.41: Průměrné naměřené hodnoty algoritmu UpDown bez použití indexu na jménu uzlu pro tři hierarchie

Příloha C

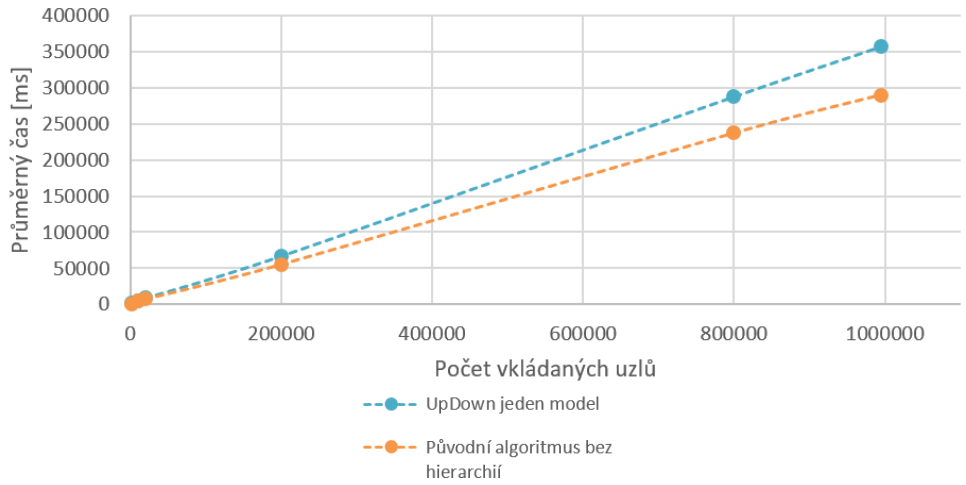
Grafy naměřených hodnot

C.1 Všechna měření



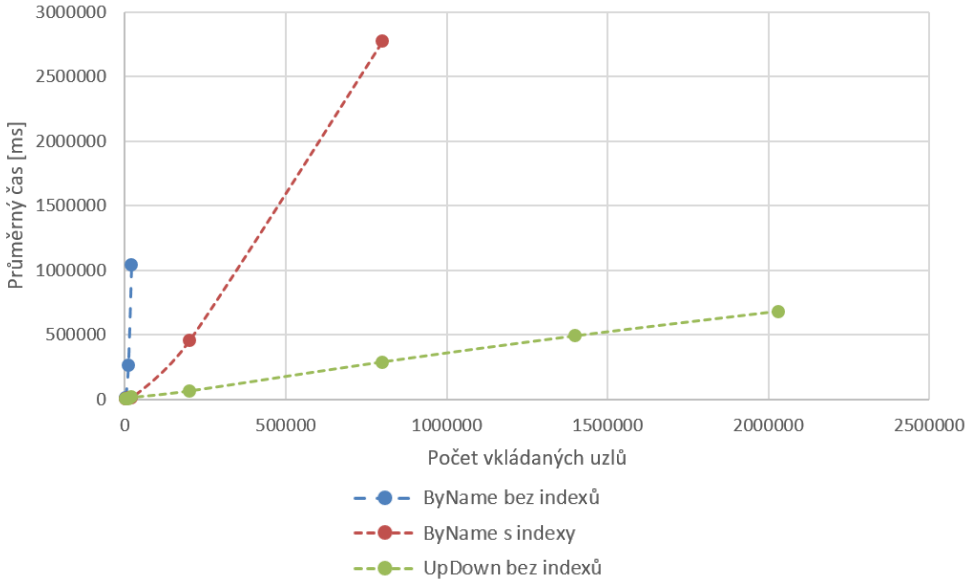
Obrázek C.1: Graf průměrných časů všech algoritmů

C.2 Jedna hierarchie



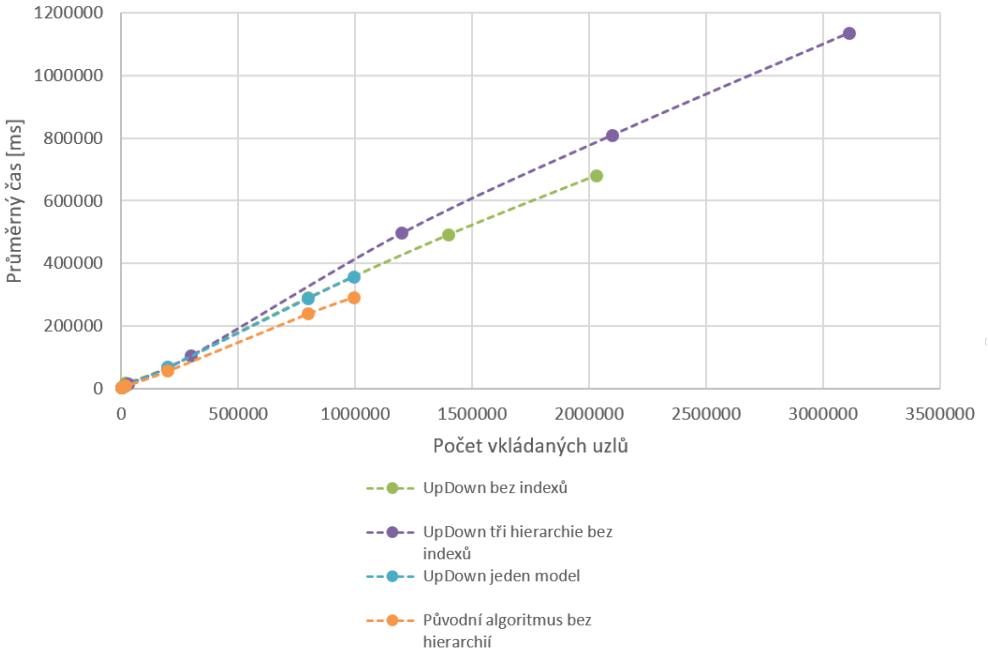
Obrázek C.2: Graf průměrných časů vkládání jedné hierarchie

C.3 Dvě hierarchie



Obrázek C.3: Graf průměrných časů vkládání dvou hierarchií

C.4 Porovnání žádné, jedné, dvou a tří heirarchií



Obrázek C.4: Graf průměrných časů vkládání žádné, jedné, dvou a tří heirarchií

Příloha D

Seznam použitých zkratek

CAP Consistency, Availability, Partition tolerance

DDL Data definition language

ETL Exctract, Transform, Load

JVM Java virtual machine

LINQ Language Integrated Query

NoSQL Non Structured Query Language

SQL Structured Query Language

XML Extensible markup language

Příloha **E**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF