Sem vložte zadání Vaší práce.

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

Magisterská práce

Temporální data v grafové databázi v projektu Manta

Bc. Petr Holeček

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

5. května 2015

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu, Ing. Michalovi Valentovi, Ph.D., za cenné rady, pomoc a diskuzi k řešení a také za seznámení s projektem Manta. Dále bych chtěl poděkovat Mgr. Tomášovi Fechtnerovi za hodnotné rady k analýze, návrhu a implementaci. V neposlední řadě bych rád poděkoval rodině za podporu a poskytnutí zázemí.

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 včech 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.

Praha dne 5. května 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Petr Holeček. *Temporální data v grafové databázi v projektu Manta: Magisterská práce.* Praha: ČVUT v Praze, Fakulta informačních technologií, 2015.

Abstract

The aim of this work is to analyze the ways of implementing temporality into the project Manta, which analyzes data flows. The object of this work is research of existing solutions of introducing time components in relational databases, design possible solutions in graph databases, evaluate effectiveness, the selection of an optimal solution for the project Manta and subsequent implementation and testing.

Keywords Manta, data flows, temporality, graph databases, metadata, TitanDB, Java

Abstrakt

Cílem této práce je analýza možností zavedení temporality do projektu Manta, který se zabývá analýzou datových toků. Předmětem práce je rešerše existujících řešení zavedení časových složek v relačních databázích, návrh možných řešení v grafových databázích, zhodnocení efektivity, výběr nejoptimálnějšího řešení pro projekt Manta a následná implementace a testování.

Klíčová slova Manta, datové toky, temporalita, grafové databáze, metadata, TitanDB, Java

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Stále se zvyšující počet aplikací a složitost softwarových systémů vede k nárůstu dat a složitosti datových toků. Jelikož systémy už většinou nepracují s vlastními daty, ale s daty dalších aplikací, vzniká mezi těmito systémy velká provázanost. Tato data se navzájem ovlivňují. V případě složitých systémů pak zavlečení jakékoliv chyby přináší do softwarových systémů mnoho problémů. Mezi ně patří špatná dohledatelnost zdroje chyby, možnost dohledání všech ovlivněných systémů a dat a v neposlední řadě také možnost data opravit.

Touto problematikou se zabývá nástroj Manta. Manta je softwarový systém sloužící k strukturální a částečně sémantické analýze redundance datových transformací, skriptů pro datové transformace (ETL procesy) v příslušné organizaci, zobrazení (vizualizace) datových toků, kontrole a analýze efektivity modelu práv apod. Nástroj Manta je koncipován tak, aby uměl data zpracovávat nehledě na aplikační doménu, proto ho lze využít pro různé systémy V současnosti však mnoho softwarových systémů zpracovává data s ohledem na čas. Mezi tyto aplikace patří finanční aplikace (např. systémy pro burzy cenných papírů), rezervační systémy (např. systémy pro objednávání letenek), zdravotní systémy (sledování zdravotní stavu pacientů a proběh nemoci v čase), počítačové systémy (historie filesystémů) a další ([9]). Nástroj Manta v současné chvíli nenabízí možnosti sledování datových toků s ohledem na časová kritéria.

Cílen této práce je

- Seznámení s problematikou temporality - co se temporalitou myslí, jaké všechny časové údaje se ukládají, co jsou temporální dimenze
- Seznámení s řešením temporality v relačních databázích - způsob uložení dat v RDBMS, výhody a nevýhody, typy temporálních databází
- Seznámení s řešením temporality v grafových databázích - zhodnocení stávajících řešení, rozebrání způsobu reprezentace dat v grafu, způsob realizace dvou temporálních dimenzí - historických metadat, procesní metadat
- Seznámení s projektem Manta - popis jednotlivých modulů a stávající funkcionality, reprezentace dat

- Analýza temporality v projektu Manta - zjištění, které temporální dimenze jsou pro projekt Manta použitelné, analýza možných způsobů implementace a výběr nejvhodnějšího řešení
- Implementace temporality - implementace řešení do stávajícího projektu
- Testování - ověření, zda bylo dosaženo stavu navrženého v analýze, ověření korektnosti implementace
- Měření - benchmarking pro porovnání výpočetní a paměťové náročnosti verzí s temporalitou a bez temporality

KAPITOLA 1

Temporalita obecně

1.1 Pohled na data z hlediska času

- Čas jako spojitá veličina - reprezentace pomocí reálných čísel, mezi dvěma časovými body lze vždy vytvořit nový bod času -> k dispozici nekonečně mnoho časových bodů
- Čas jako diskrétní veličina - reprezentace pomocí přirozených čísel, jedná se o množinu bodů času se stejnou vzájemnou vzdáleností a stejným pořadím

Chronon - nejkratší možný časový interval, tato jednotka je dále nedělitelná. Slouží k sestavení všech časových jednotek pro diskrétní čas.

K vymodelování času se používá model diskrétního času. Důvodem je jeho jednoduchost, snadná implementace a také na něj je více zaměření z hlediska výzkumu. Model spojitého času by kvůli své nekonečné přesnosti představoval problém z hlediska aritmetických operací.

1.2 Temporální dimenze

Temporální dimenze představují typy časových složek jednotlivých objektů ([4])

- Transaction time - databázové objekty jsou ukládány v určitém časovém okamžiku. Transakční čas reprezentuje čas, kdy byl objekt v databázi vytvořen.
- Valid time - čas platnosti databázového objektu reprezentuje interval, kdy objekt nabýval daného stavu.

1.3 Druhy temporálních databází

Temporální databáze se z hlediska druhu temporálních informací dělí na následující typy ([8]).

1.3.1 Historical databases

Historické databáze uchovávají data s ohledem na dobu, kdy byla data platná (validní) vzhledem k reálnému světu. Příkladem může být informace o zaměstnanci, který pracoval od roku 1965 do roku 1975 ve firmě. Interval validity dat (záznamu zaměstnance) je zde 1965-1975.

1.3.2 Rollback databases

Rollback databáze uchovávají data s ohledem na čas transakce. Na rozdíl od historických databází, které uchovávají čas validity vzhledem k reálnému světu, zachycují rollback databáze čas zápisu a platnosti v samotné databázi. Příkladem může být například zaměstnanec, který sice pracoval v letech 1965-1975, ale do databáze byl záznam uložen až v roce 1985. Rollback databáze nám tedy umožňují jakési snapshoty - tedy dotazovat se na obraz databáze v určitém časovém okamžiku. Pro počáteční transakční čas platí, že nesmí být větší než současný čas (tedy i v případě že víme, že data budou například pomocí dávky uložena až zítra, musí být transakční čas nastaven až v momentě spuštění dávky)

1.3.3 Bitemporal databases

Bitemporální databáze uchovávají informace o platnosti i o časech transakcí, jedná se o historické a současně rollback databáze.

1.4 Řešení temporality v RDBMS

K zavedení temporality do relačních databází je nejprve potřeba vyřešit samotnou reprezentaci času. Většina databázových strojů podporuje datové typy pro práci s časem a datumy - klasicky bývá datum reprezentováno datovým type DATE, čas datovým typem TIME a datový typ pro spojení obou těchto informací je označován jako TIMESTAMP ([7]). Zde však narážíme na zásadní problém - data týkající se temporality nejsou obecně definována SQL standardem. To vede k tomu, že databázové stroje mají sice stejně pojmenované datové typy pro čas, ale obecně mohou být tyto typy odlišné. Příkladem může být datový typ DATE na databázovém stroji Oracle, který se ve skutečnosti chová jako TIMESTAMP, jelikož si drží i údaje o čase. Takovéto odlišnosti vedou ke špatné 'přenositelnosti' kódu mezi jednotlivými typy databázových strojů. Pozn.: Všechny příklady v této kapitole jsou uváděny v Microsoft TSQL dialektu.

1.4.1 Koncepty pro modelování temporálních dat v RDBMS

Obecně se pro modelování a práci nad temporálními daty používá temporální algebra ([5]), která je dále popsána v ([3]). Pro modelování dat v relačních databázích budou v rámci držení temporálních údajů dodržována následující pravidla, která jsou využívána například i v temporálních databázích firmy Teradata. ([6]).

- Každý databázový objekt (v RDBMS n-tice) bude držet časovou informaci (použit model pro diskrétní čas)
- Obor hodnot pro časový interval bude uzavřený a konečný, tj. všechny časové hodnoty budou elementem tohoto intervalu
- Bude použit model lineárního času, tj. pro jakýkoliv časový bod bude dostupná pouze jedna verze dat (na rozdíl od nelineárního modelu, který umožňuje větvení)

1.4.2 Historical Databases

Zde je potřeba držet informaci o časovém intervalu reprezentujícím platnost záznamu - tento interval lze klasicky reprezentovat zavedením počátku platnosti a konce platnosti - tedy přidáním dvou sloupců. V rámci dotazování se pak lze doptávat, zda je datum, ke kterému nás platnost zajímá, v požadovaném intervalu.

Příkladem může být následující dotaz, který vytáhne všechny řádky platné k aktuálnímu datu

- Čtení záznamu - v rámci SELECTU pravděpodobně budeme chtít zohlednit datum platnosti, tj. aby počátek platnosti byl menší než zadané datum a současně aby konec platnosti byl větší než zadané datum (nebo bylo datum konce platnosti prázdné)- tím docílíme platnosti dat k určitému datu.

```
DECLARE @current_date DATE
SET @current_date = getDate()

SELECT * FROM data AS d
WHERE (@current_date BETWEEN d.valid_from AND d.valid_to)
       OR d.end_date = NULL;
```

- Přidání nového záznamu - postačí obyčejný INSERT s nastavením počátku platnosti záznamu.

```
DECLARE @current_date DATE
SET @current_date = getDate()

INSERT INTO data (column, valid_from)
VALUES (value1, @current_date);
```

- Úprava existujícího záznamu - v rámci historických databází při změně záznamu buď měníme netemporální data, která nemají na validitu vliv, nebo měníme přímo temporální data pomocí jednoduchého UPDATE příkazu.

- Mazání existujícího záznamu - zde je možnost smazat celý záznam, ale tím ztratíme informace o temporálních datech. Pokud tedy nějaký záznam platící od minulosti skončí, řešením je mu pouze ukončit platnost

```
DECLARE @current_date DATE
SET @current_date = getDate()

UPDATE data
SET valid_to = @current_date
WHERE id = 1;
```

1.4.3 Rollback databases

V rollback databázích je nutné si pamatovat čas transakce - tedy čas zápisu. Konec transakčního času by znamenal, že záznam byl smazán a tím pádem byla smazány temporální údaje, proto je v případě nutnosti pamatování si tohoto času nutné rozdělit časové údaje do dalších tabulek.

- Čtení záznamu - SELECT je podobný jako v historických databázích, avšak zde zjišťujeme stav databázových dat k určitému časovému bodu.

```
DECLARE @current_date DATE
SET @current_date = getDate()

SELECT * FROM data AS d
WHERE
  (@current_date BETWEEN d.tran_start AND d.tran_end)
    OR
  (@current_date < d.tran_start AND d.tran_end = NULL)
```

- Přidání nového záznamu - novému záznamu musí být v rollback databázích přidělen počáteční transakční čas. Záleží na databázovém administrátorovi, jestli je vhodné nastavovat transakční počátek ručně, nebo pomocí triggeru. Datum konce, tedy transakční konec, lze vyjádřit nenastaveným atributem pro transakční konec, nebo nastaveným na poslední možný chronon. Standardně se však používá první možnost.

- Úprava existujícího záznamu - UPDATE existujících dat provede z hlediska databázového obrazu změnu, proto je nutné ukončit transakční platnost a vytvořit nový záznam. Příkladem může být následující SQL kód, který je sice podobný kódu pro úpravu záznamu v historických databázích, ale významově jde o odlišný kód.

```
DECLARE @current_date DATE
SET @current_date = getDate()

UPDATE data
SET tran_end = @current_date
WHERE id = 1;

INSERT INTO data (id, column, tran_start)
VALUES (1, "new_value", @current_date);
```

- Mazání existujícího záznamu - mazání reprezentuje skutečnost, že záznam má přestat existovat databázi. Jelikož ale chceme mít obrazy databáze k určitému datu, nesmíme záznam mazat, ale pouze mu nastavit datum ukončení transakce, příkladem může být následující SQL kód.

```
DECLARE @current_date DATE
SET @current_date = getDate()

UPDATE data
SET tran_end = @current_date
WHERE id = 1;
```

1.4.4 Bitemporal databases

Jak již bylo řečeno výše, bitemporální databáze musí uchovávat obě informace - jak čas validity dat vůči reálnému světu, tak informace o existenci

dat v databázi, bitemporální databáze jsou tedy typově historické a současně rollback databáze.

- Čtení záznamu - čtení je v podstatě identické jako v předešlých příkladech. Buď nás zajímá existence platnost dat a proto se dotazujeme na `valid_from` a `valid_to`, nebo se dotazujeme nad databázovým obrazem pomocí `TRAN_START` a `TRAN_END`. Případně můžeme využít obou temporálních dimenzí - příkladem může být následující T-SQL kód. Ten zjistí, od kdy do kdy byl zaměstnán zaměstnanec s id 1 ke dni 23.10.2004.

```
-- deklarace promennych pro datum
DECLARE @Day TinyInt SET @DayOfMonth = 23
DECLARE @Month TinyInt SET @Month = 10
DECLARE @Year Integer SET @Year = 2004
DECLARE @date DATE

-- ulozi datum do promenne
SET @date = SELECT DateAdd(day, @Day - 1,
         DateAdd(month, @Month - 1,
             DateAdd(Year, @Year-1900, 0)))


SELECT e.valid_from, e.valid_to FROM employee AS e
WHERE
  (e.id = 1) AND
  (
    (@date BETWEEN e.tran_start AND e.tran_end)
      OR
    (@date < d.tran_start AND d.tran_end = NULL)
  )
```

- Přidání nového záznamu - v INSERT příkazu je potřeba pouze specifikovat hodnoty pro platnost záznamu a ke kterému okamžiku databázová data náleží.
- Úprava existujícího záznamu - pro úpravu platí to co pro historické i rollback databáze, při úpravě času validity jsou ovlivněny obě dimenze.
- Mazání existujícího záznamu - jelikož je nutné držet obraz databáze, pouze se přepíše hodnota `TRAN_END` stejně jako v případě rollback databází.

1.5 Nástroj Manta

Jak již bylo řečeno v úvodu, Manta je analytický nástroj sloužící k strukturální a částečně sémantické analýze a celkově zpřístupněním informací o datových tocích. Datové toky jsou definovány analýzou transoformačních skriptů

pracujícími nad daty dané organizace. Tato data jsou zpracována a analyzována a výsledek je uložen ve formě metadat (tedy dat o datech).

1.5.1 Architektura

Architektura systému je rozdělena na serverovou a klientskou část, které mezi sebou komunikují pomocí specifikovaného komunikačního rozhraní nad TCP/IP protokolem. Serverová část, která obsahuje grafovou databázi, je složena z modulů, z nichž nejdůležitější jsou moduly pro import, export a dotazování se nad grafovou databází. Architekturu systému popisuje obrázek 1.1.

1.5.2 Použité technologie

Systém Manta běží na platformě Java 1.7 s použitím Spring frameworku. O databázi se rozhodovalo v rámci analýzy a testů viz [11], kde byla grafová databáze Neo4j verze 2 vyhodnocena jako vhodnější, než relační databáze ProstgreSQL verze 9. Z licenčních důvodů ale bylo od databáze Neo4j upuštěno (Neo4j lze používat pod GPL licencí), jejím náhradníkem je v současné chvíli databáze TitanDB (Apache 2 licence), která byla vybrána na také na základě analýzy a testů, viz [10] a která je pro projekt Manta výkonnostně srovnatelná s databází Neo4j.

1.5.3 Hlavní moduly systému

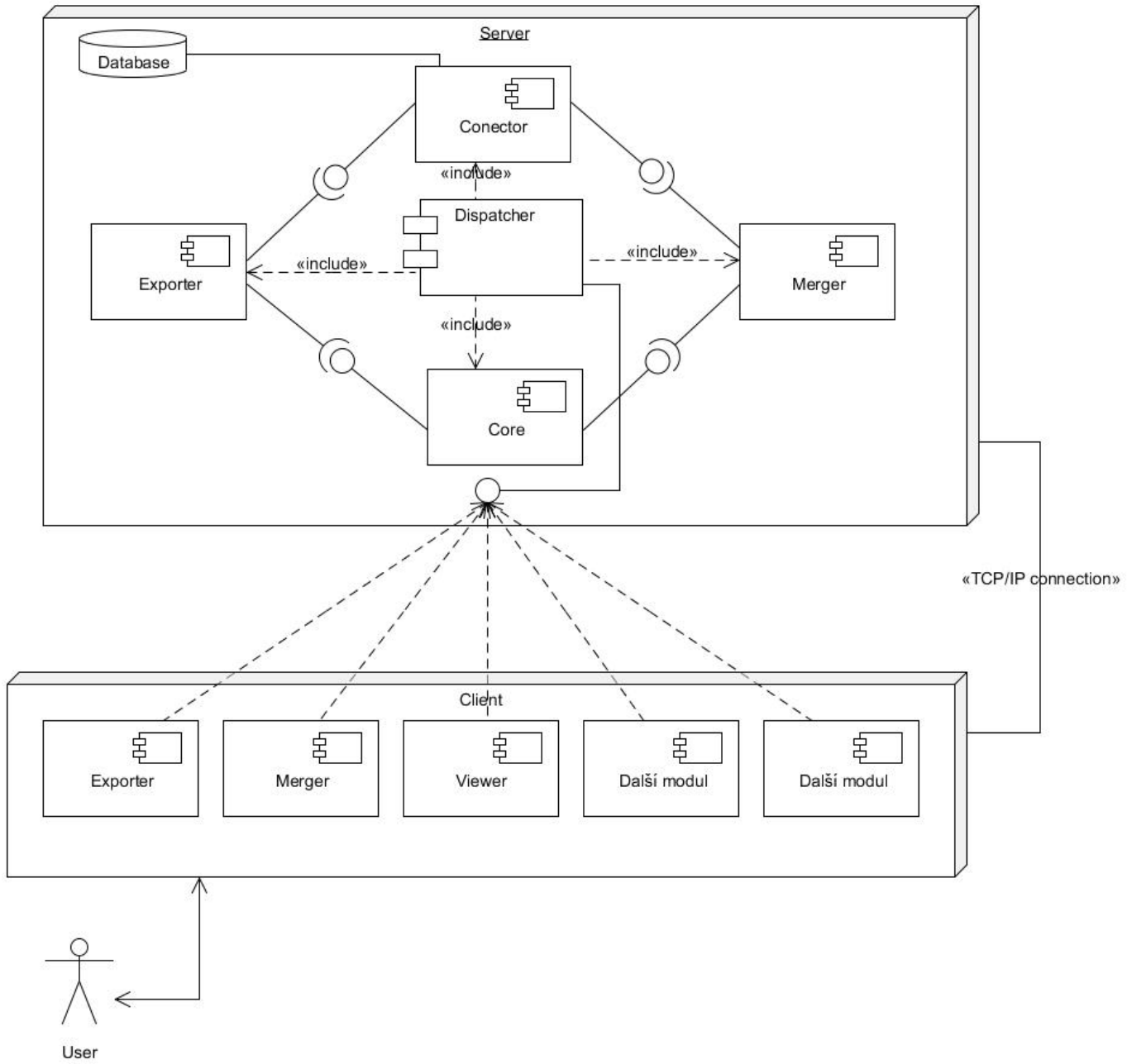
- Merger – jedná se o modul slouží ke spojování grafů. Na svém vstupu bere grafové toky (ze skriptu/ ETL transformace) reprezentované grafem, které následně přidá do existujícího grafu datových toků. Spojování provádí na základě identifikátorů, tj. nedochází k vytváření duplicitních dat, ale k namapování vstupu na existující zdroje.

- Query – modul sloužící k dotazování nad metadaty, tj. nad data reprezentována grafem. Příkladem mohou být dotazy na získání všech datových toků a ovlivněných objektů daným objektem (tedy například sloupcem tabulky).

- Dispatcher – modul zajišťující navázání spojení, komunikaci mezi serverem a klientem, získání vstupů od klienta, inicializaci a volání příslušného serverového modulu a pro předání výstupu klientovi.

- Connector - modul zapouzdřující databázi a poskytující rozhraní pro transakční zpracování

- Exporter - modul zapouzdřující export dat



Obrázek 1.1: Architektura systému Manta dle zadané dokumentace

1.5.4 Reprezentace dat a metadat

Temprální data jsou získána z datových zdrojů, v systému Manta označovaných jako resource. Tyto datové zdroje mohou být různých typů - např. produkční databáze, transformační skripty, filesystém atd. Každý z těchto resourců má vlastní loader, který slouží k transformaci dat do podoby metadat a datových toků pro uložení do databáze. Metadata jsou ukládána v grafové databázi TitanDB. Struktura metadat je reprezentována stromově s následující reprezentací:

1.5.4.1 Uzly

- Super root/root (kořenový uzel) - speciální uzel reprezentující vrchol grafu, jediný v celé grafové databázi, obsahuje vazby na všechny resource vrcholy databáze
- Resource (zdrojový uzel) - typ uzlu reprezentující jednu konkrétní technologii - například Oracle, Teradata atd. Resource uzel má následující atributy:
  - resourceName - jméno daného datového zdroje
  - resourceType - typ dané technologie, příkladem může být technologie Oracle PL/SQL
  - resourceDescription - popis daného datového zdroje
- Node (datový uzel)- slouží popisu jednoho konkrétního uzlu v datových tocích. Má následující atributy:
  - nodeName - jméno uzlu
  - nodeType - typy těchto uzlů jsou například databáze, schéma, tabulka, sloupec, soubor, parametr atd.
- Atribute (atribut) - jde o přídavné atributy pro vrcholy typu Node.
  - atributeName - jméno atributu
  - atributeValue - hodnota atributu

1.5.4.2 Hrany

Vazby jsou děleny podobně jako uzly do určitých typů. Hrany typu DIRECT a FILTER slouží k zachycení datových toků. Hrany hasParent, hasAttribute a hasResource jsou "vnitřní"hrany sloužící pro udržení struktury úložiště.

- hasResource - hrana, nachází se tam, kde zdrojový uzel typu node má jako cílový uzel resource. Dále je tato hrana použita pro všechna spojení

mezi resource vrcholem a root vrcholem (orientace je ve směru od resource vrcholu do root vrcholu). Platí, že každý vrchol má nejvýše jeden resource, proto může z uzlů typu node vést maximálně jedna tato hrana.

- hasParent - hrana sloužící k hierarchické reprezentaci uzlů. Tato hrana má jako zdroj uzel typu node, cíl je reprezentace předka v hierarchii objektů. Příkladem může být tato hrana mezi sloupcem tabulky a samotnou tabulkou.
- hasAttribute - slouží pro propojení uzlu typu node s jeho atributem (uzel typu atribute)
- directFlow (DIRECT)- reprezentace datového toku. Tato hrana už neslouží k hierarchickému popisu dat, ale k popisu předmětu a objektu datového toku, tj. v pomyslné grafové reprezentaci nevede hrana vertikálně, ale horizontálně. Příkladem může být hrana mezi dvěma uzly node, symbolizující ovlivnění cílového uzlu zdrojovým uzlem.
- filterFlow (FILTER)- tato hrana je použita tam, kde hodnota jednoho listového datového uzlu je řízena (filtrována) hodnotou jiného listového datového uzlu

1.5.4.3 Indexy

Další nedílnou součástí struktury datového úložitě Manta jsou indexy sloužící k rychlému vyhledávání. V současné chvíli jsou používány 3 indexy.

- Fulltextový index nad uzly vyhledávající přes atribut nodeName (jméno uzlu) řešený pomocí externího Lucene indexu
- Index nad uzly vyhledávající přes atribut superRoot sloužící k rychlému vyhledání kořenového uzlu
- Index nad hranami vyhledávající přes atribut childName pro získání potomka s daným jménem od daného předka

KAPITOLA 2

Analýza a design

2.1 Reprezentace temporálních dat v grafu

Tato kapitola je věnována způsobu reprezentace temporálních dat. Jsou v ní rozebrány jednak možnosti uložení těchto dat, ale také výhody a nevýhody jednotlivých možností vzhledem k operacím nad grafem.

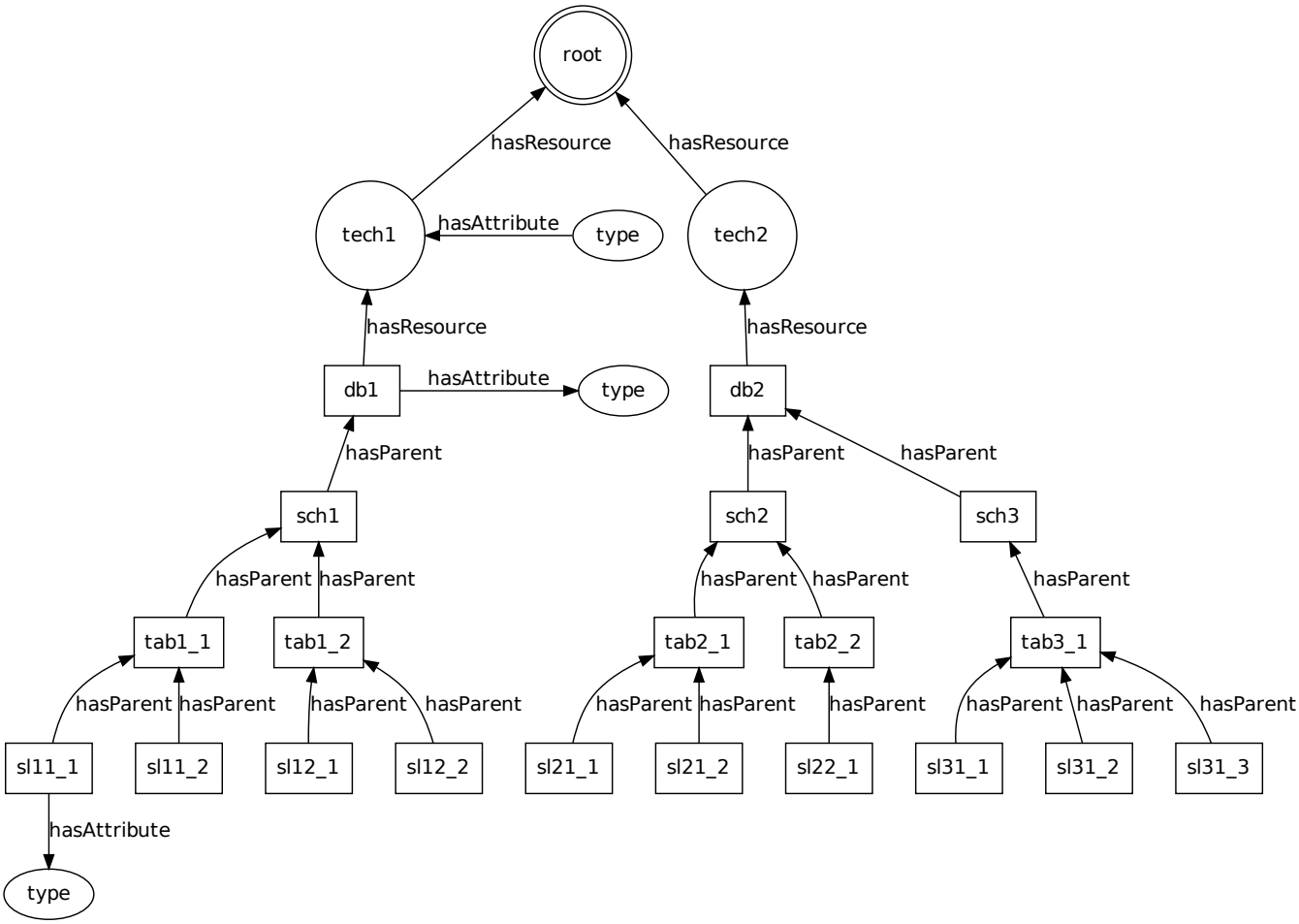
Jelikož tato analýza zohledňuje projekt Manta, v němž v současné době nejsou data, u nichž je potřeba držet informace o validitě, ale naopak je žádoucí možnost dohledat stav metadat k určitému datu, či v jaký den daný skript ovlivnil databázové objekty, budou zde temporální data uváděna pro případ rollback databází, tj. složky času budou reprezentovat hodnoty `TRAN_START` (počáteční transakční čas) a `TRAN_END` (konečný transakční čas) tak, jak je to popsáno v kapitole 1.3.2. Temporalita z hlediska času validity nebude z tohoto důvodu v této práci řešena.

2.1.1 Testovací podgraf

V rámci analýzy byl z důvodu simulace možných operací na grafem vytvořen skript sloužící k vytvoření testovacího grafu v databázi Gremlin a v programu GraphViz pro vizualizaci. Tento graf vypadá následovně (2.1).

Pro graf platí následující:

- Uzel Root je reprezentován dvojitým kruhem
- Zdrojové uzly jsou reprezentovány jednoduchým kruhem
- Datové uzly jsou reprezentovány obdelníkem
- Atributové uzly jsou reprezentovány elipsou



Obrázek 2.1: Testovací podgraf

Pojmenování uzlů a hran bylo pro účely demonstrace akcí voleno generic-ky, tj. nejedná se o žádné konkrétní názvy. V testovacím grafu se kořenový uzel nazývá `root`, zdrojové uzly reprezentující technologie mají názvy `tech[pořadové číslo]`, databáze mají název `db[číslo databáze]`, databázové schéma je `sch[číslo schématu]`, tabulka je pojmenována jako `tab[číslo schématu]_[číslo tabulky]`, tabulkový sloupec je `sl[číslo schématu][číslo tabulky]_[číslo sloupce]`.

Data mohou být v grafu reprezentována následujícími způsoby - jako hodnota na uzlu, hodnota na hraně, nebo připojením uzlu s časovým údajem na daný uzel.

2.1.2 Dotazování na stav grafu vzhledem k validnímu času

Primárním účelem zavedení temporálních dat do databáze metadat reprezentované grafem je ten, že se budeme moci dotazovat na stav databáze k určitému časovému okamžiku. Vzhledem k časovému okamžiku jsme pak schopní zjistit, jak vypadala například určitá tabulka v čase t_x apod.

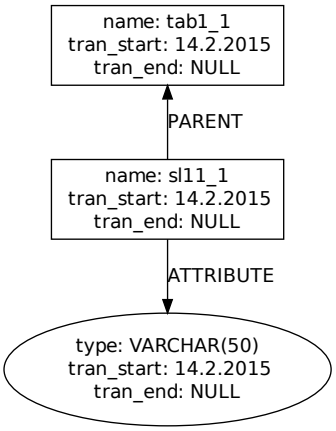
V následujících podkapitolách jsou probrány jednotlivé způsoby držení temporálních metadat vzhledem k změnám v grafu a případném dotazování se nad grafem.

2.1.2.1 Údaje na uzlu

První možností je data reprezentovat přímo na uzlu, jak je uvedeno v následujícím obrázku. V případě změny uzlu by bylo nutné nastavit transakční konec (`TRAN_END`), vytvořit nový, modifikovaný uzel, nastavit mu datum počátku transakce (`TRAN_START`) a vytvořit hrany (na propojení nového uzlu s rodičovským uzlem a propojení s případnými potomky). Data na uzlu znázorňuje obrázek (2.2.

S dalším postupem máme dvě možnosti. Pro obě platí, že z důvodu nutnosti držení stavů nepřipadá v úvahu odmazávání uzlů a hran.
První možností je propagovat informaci o změně i na rodiče podle následující úvahy - tím, že je ale změněn podstrom rodiče, je potřeba změnit i temporální data rodiče - změnu lze provést pouze zkopírováním uzlů, nastavením transakčních dat a vytvořením nových hran. Takto se rekurzivně dostaneme k tomu, že je potřeba měnit všechny předky až k kořenovému uzlu. To však vede k nepřípustné paměťové a výpočetní náročnosti jakékoliv operace v případě větších grafů.

Druhou možností je neměnit transakční konce rodičů. Tím docílíme toho, že není potřeba procházet předky až ke kořenu, ale pouze přidružené uzly. Výpočetní i paměťová náročnost tedy klesla, ovšem v případě implementace algoritmu pro dotazy na temporální složky by bylo nutné z každého uzlu kontrolovat všechny přidružené uzly z důvodu kontroly transakční platnosti.

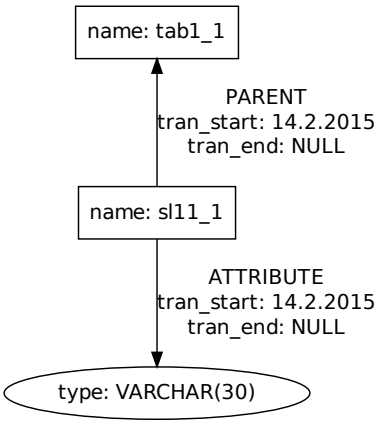


Obrázek 2.2: Transakční údaje na uzlu

Samotná operace změny je tedy rychlejší, ale následné dotazování nad grafem by pak bylo pomalejší. Toto řešení je přijatelnější než první možnost.

2.1.2.2 Údaje na hraně

Další možností je temporální data držet na hranách. Reprezentace temporálních dat na hraně by vypadala takto (2.3).



Obrázek 2.3: Transakční údaje na hraně

Hrana by tak určila stav vzhledem k času. V případě, že by došlo k změně údajů na datovém uzlu, hraně by se nastavil čas konce transakce, podstrom by se zkopíroval, aplikovala by se na něm požadovaná změna a hraně spojující tento podstrom a rodičovský uzel by se nastavil čas počátku transakce, který by byl roven datu konci transakce na původním podstromu.

Reprezentace časových údajů na hraně tedy obnáší podobné problémy jako reprezentace přímo na uzlu. Jelikož nemůžeme mazat, musíme situaci řešit přepojováním hran. Zde máme ale oproti držení temporálních údajů na uzlu výhodu - můžeme využít možnosti multihran, tedy možnosti existence více hran mezi dvěma uzly. Tato operace projde podstrom, nastaví konec současným a hranám a vytvoří hrany nové (se stejným zdrojovým a koncovým uzlem) s transakčním počátkem rovným aktuálnímu času. Změněný uzel se zkopíruje a upraví.

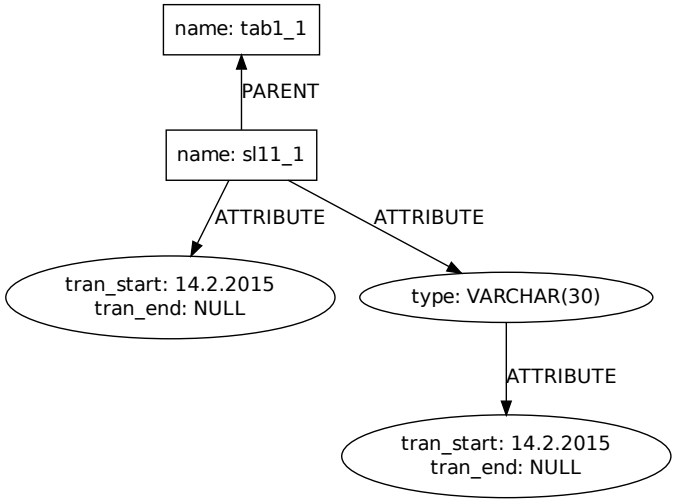
Operace bude také výpočetně náročná, jelikož se musí projít všichni předci, všem existujícím hranám se musí nastavit `TRAN_END` a musí se vytvořit nové hrany s nastaveným `TRAN_START` rovnému `TRAN_END` původních hran. Paměťová náročnost bude nižší, jelikož není třeba vytvářet nové uzly.

Třetí možnost je analogická k možnosti u držení údajů na uzlu. Provedu kopii uzlu, aplikuji na něm změny, staré hraně nastavím transakční konec. Vytvořím nové hrany, nastavím jim transakční počátek, ale rekurzivně už neprocházím předky ani potomky. Tím docílím rychlé operace změny grafu, ale pomalé operace dotazů vzhledem k temporálním datům, jelikož při průchodu musím kontrolovat čas na všech hranách.

2.1.2.3 Údaje reprezentovány v samostatných uzlech

Poslední možností, jak udržovat temporální údaje, je vytvořit uzly pro každý časový údaj. Reprezentace by pak mohla vypadat následovně (2.4).

Změna uzlu by pak vypadalo podobně jako v předchozím možnostech. Při změně se zkopíruje uzel, provede se na něm změna, původnímu uzlu se atribut s časovým údajem upraví, tj. nastaví se konec transakčního času. Novému uzlu se nastaví počátek transakce. Dále máme stejné možnosti jako v předchozích příkladech - buď vytvářet nové uzly pro temporální data u všech předků s tím, že dotazování nad grafem v podobě selectů bude rychlejší, ale vytváření nových uzlů bude příliš výpočetně a paměťově náročné, nebo předky ovlivňovat nebudu. Tím dosáhnu rychlejších úprav grafu, ale dotazování bude pomalejší - dokonce nejpomalejší ze všech možností, jelikož musím z každého uzlu přejít přes hranu do všech okolních uzlů, poté se přes hranu ATTRIBUTE dostat na uzel s temporálními daty, ty zkontrolovat a podle toho se rozhodnout, kudy pokračovat v grafu dále.



Obrázek 2.4: Transakční údaje v samostatných uzlech

2.1.3 Atributová reprezentace

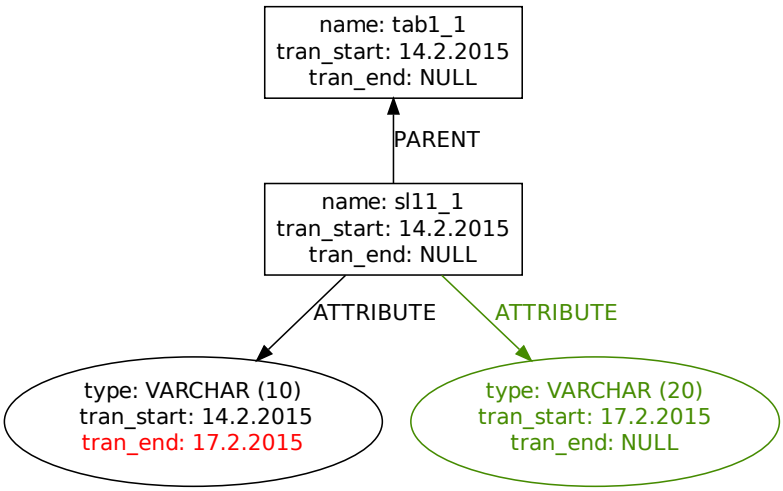
Ve všech třech řešeních se potýkáme s problémem výpočetní náročnosti operací. Důvodem je jednak výpočetní náročnost propagace změn, při které dochází k rekurzivnímu kopírování předků ale také fakt, že při změně se musí samotné měněné uzly kopírovat a musí se vytvářet kopie hran. Tomu se však dá poměrně jednoduše zamezit - veškerá metadata o datových uzlech budu reprezentovat novým uzlem. Tím se vyhnu potřebám uzly kopírovat, stačí vytvořit pouze nový uzel a vazbu. Jelikož budou veškerá metadata reprezentována uzlem (ne již uzlem s hodnotami), bude sice velikost grafu větší, ale v tomto případě hraje klíčovou roli rychlost dotazování.
V následujících podkapitolách bude rozebrána možnost reprezentace temporálních dat za předpokladu, že veškerá data o uzlech jsou reprezentována pomocí uzlů.

Obrázky v následujících kapitolách budou sloužit k reprezentaci operace. Zelené objekty reprezentují nové objekty, červené objekty reprezentujíc změnu.

2.1.3.1 Údaje na uzlu

Jelikož jsou údaje jakéhokoliv uzlu uloženy jako uzel, stačí pouze vytvořit nový uzel a připojit k němu hranu. Následující obrázek popisuje změnu datového typu sloupečku. Uzlu obsahující datový typ uzlu se nastaví transakční konec a vytvoří se nový uzel, kde transakční počátek je roven transakčnímu konci původního uzlu, a hrana spojující nový uzel s předkem. Situace je lépe

patrná z obrázku 2.5.



Obrázek 2.5: Transakční údaje na uzlu - případ reprezentace metadat pouze pomocí atributů

2.1.3.2 Údaje na hraně

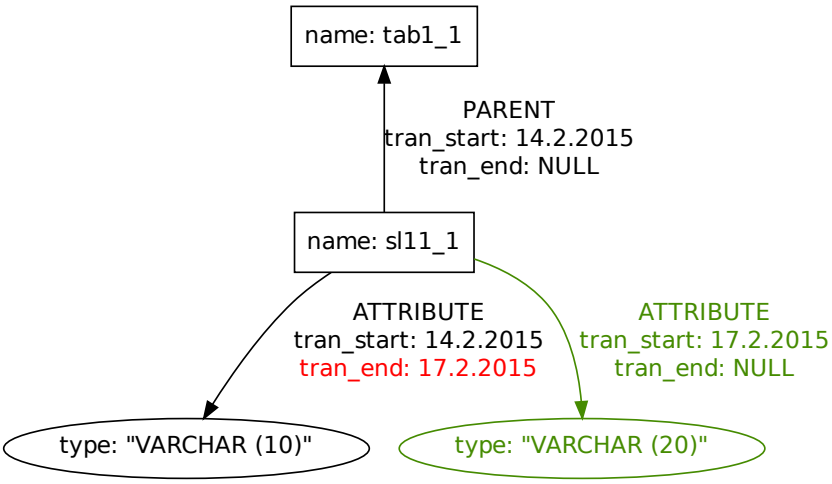
V případě změny na hraně je řešení podobné. Vytvoří se nový uzel, analogicky se nastaví transakční časy, ovšem tentokrát to není na uzlu, ale přímo na hraně. Změna datového typu sloupce může vypadat následovně (2.6).

Toto řešení je pro dotazování rychlejší než první možnost, jelikož se z uzlu dívám na okolní hrany, zatímco v případě držení temporálních dat na uzlu musím ze zdrojového uzlu přejet přes všechny hrany na sousední uzly a až tam kontrolovat transakční čas.

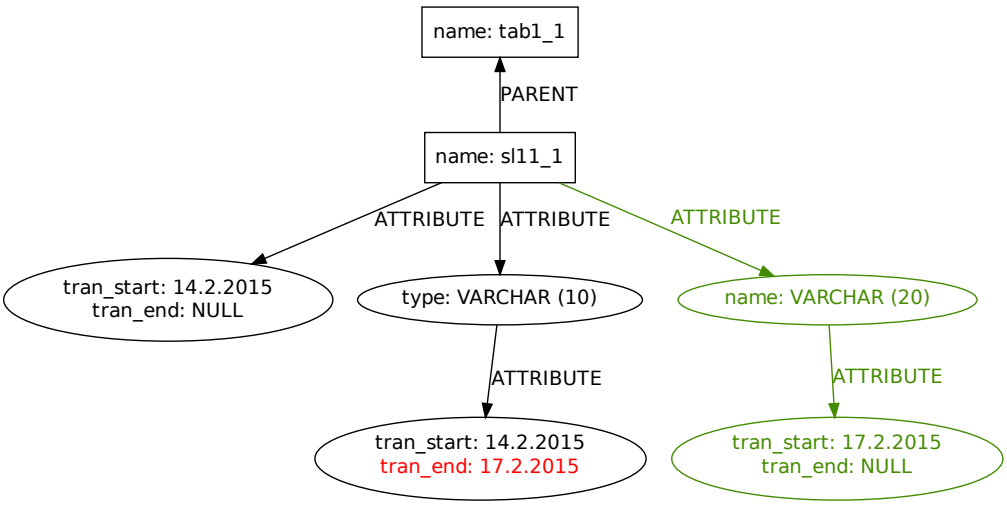
2.1.3.3 Údaje v samostatných uzlech

V případě uložení temporálních dat v samostatných uzlech je operací k provedení změny více než v předchozích dvou způsobech uložení temporálních metadat. Místo vytvoření jedné hrany a jednoho uzlu jsou potřeba dvě hrany a dva uzly, jelikož musíme vytvořit nový uzel s temporálními metadaty. Lépe je situace patrná z následujícího obrázku (2.7).

Z výkonnostního hlediska je z výše navržených řešení toto nejpomalejší, ačkoliv rozdíl není markantní. Důvody jsou dva - jak je zde uvedeno, změna atributů obnáší změnu transakčního času původního atributu, vytvoření nového (upraveného) uzlu reprezentující atribut, propojení tohoto uzlu s předkem,



Obrázek 2.6: Transakční údaje na hraně - případ reprezentace metadat pouze pomocí atributů



Obrázek 2.7: Transakční údaje na hraně - případ reprezentace metadat pouze pomocí atributů

dále vytvoření nového uzlu s temporálními metadaty a propojením tohoto uzlu s nově vytvořeným uzlem reprezentujícím nově upravený atribut. V případě držení temporálních metadat na hraně je to o dvě operace navíc (5 operací vůči třem operacím). Druhým důvodem je možnost následného temporálního dotazování na grafem. V případě, že má uzel více potomků a našim cílem je získat data z potomka platného k zadanému datu, je nutné projít přes všechny potomky a na každém potomkovi projít přes všechny atributové hrany, najít atribut s transakčními údaji a až na základě tohoto údaje se rozhodnout, zda je potomek validní vzhledem k zadanému času.

2.1.4 Dotazování na ovlivněná daty vzhledem k času

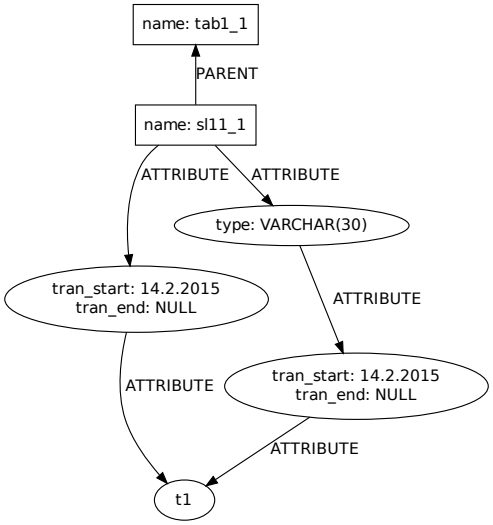
V předchozích kapitolách byly zohledněny způsoby uložení času transakcí. Držením těchto informací máme možnost dotazování se nad grafem typu "Jak vypadala tabulka v čase t_x?"apod. V případě, kdy budeme chtít zjistit, co všechno bylo v čase t_x ovlivněno (nebo například "Jaké datové toky byly ovlivněny spuštěním daného skriptu v čase t_x?"), dostáváme se do problémů. Z podstaty problému vyplývá, že je nutné v grafu držet reference na změněné objekty v určitém čase. Přirozeným řešením je si držet všechny okamžiky a propojovat je s temporálními daty jednotlivých uzlů.

- Údaje na uzlu - pokud bychom chtěli získat všechna ovlivněná data v daný okamžik t_x, museli bychom situaci řešit propojením všech uzlů, které byly ovlivněny v čase t_x. Pak ale vyvstává otázka, proč by měly všechny uzly držet tuto informaci - stačilo by tedy, aby informaci o čase t_x držel pouze jeden uzel. Pokud by tuto informaci měl jenom jeden uzel, byl by problém s prvním způsobem dotazování, tj. stav grafu v čase, jelikož bychom museli kontrolovat platnost uzlu prohledáním všech uzlů propojených v rámci sdílení časového okamžiku. Toto řešení je však výpočetně náročné.

- Údaje na hraně - zde je obdobný problém - není jak sdílet temporální informaci mezi hranami. Možností by bylo každou hranu s temporálními daty změnit na konstrukt vazba-uzel-vazba, ve kterém by uzel obsahovat temporální informace. Tyto uzly by se pak daly vzájemně referencovat, ovšem toto řešení je prakticky změna na reprezentaci v samostatných uzlech s nevýhodou nutnosti duplikace hran.

- Údaje v samostatných uzlech - zde máme možnost temporální data vzájemně referencovat. Tím získáme možnost se na data dotazovat "v obou směrech"- tj. při průchodu stromu od kořenového uzlu (dotazování na validitu k času t_x) a také k průchodu směrem od časového uzlu (dotazování se všechna ovlivněná data v čase t_x)

Reprezentace by tedy mohla vypadat následovně (2.8).



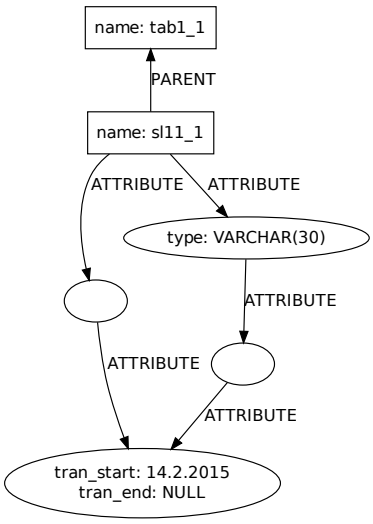
Obrázek 2.8: Reprezentace temporálních údajů vzhledem k dotazování na ovlivněné objekty v daný čas

Z obrázku je však evidentní datová redundance. Data jsou uložena na více uzlech najednou. Reference na jedno místo lze využít a přesunout data sem. Dostáváme následující strukturu (2.9).

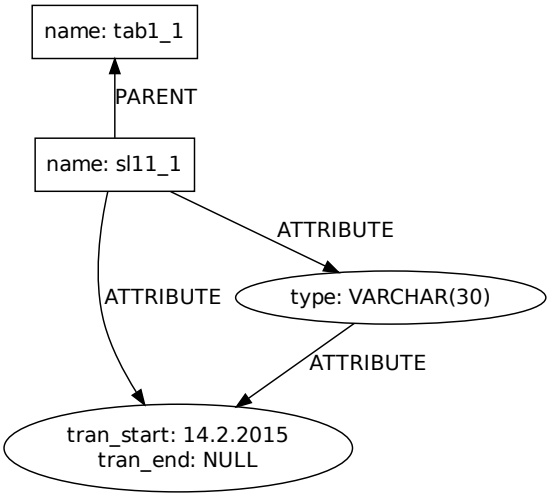
Temporální data jsou sice už uložena pouze na jednom místě, ale v grafu jsou nyní prázné a nepoužitelné uzly. Jejich odstraněním dostáváme následujícíc grafovou strukturu (2.10).

Z hlediska reprezentace dat v grafu je tato struktura v pořádku. Nabízí se však otázka, zda je reprezentace časových intervalů v pořádku. Pokud bychom měli například uzly s transakční platností od 1.1.2015 do 24.12.2015 a uzly od 18.6.2015 do 24.12.2015, dostali bychom se k podobné datové redundanci, jelikož oba uzly reprezentující temporální data by v sobě obsahovaly složku se stejným datem (transakční konec). Tato situace by vypadala následovně (2.11)

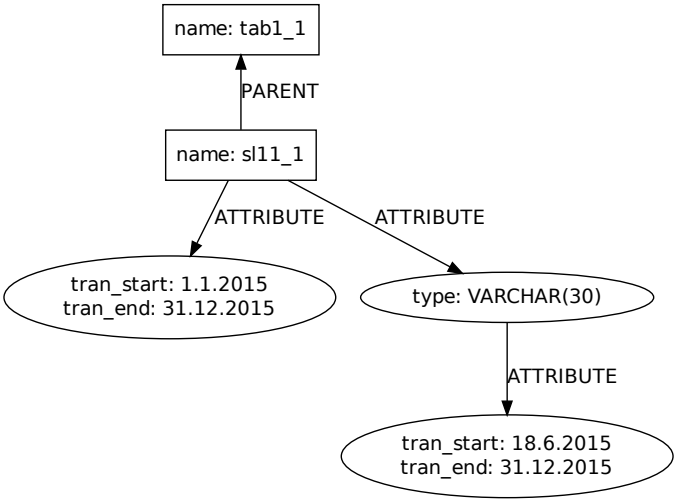
Řešením toho problému je rozdělit temporální data do dvou uzlů - tj. každý uzel uzel držící temporální data by obsahoval pouze jednu složku - jedno blíže nespecifikované datum. Typem hran pak bude nutné rozlišit, zda se jedná o transakční počátek nebo transakční konec (tyto uzly s temporálními daty by šlo případně použít i pro historické databáze, kdy by časový okamžik mohl být i okamžikem počátku nebo konce validního času). Rozlišení lze docílit nastavením typu hrany - pro transakční časy např. `TRAN_START` a `TRAN_END`. Pokud by uzel neměl ukončen transakční čas, jednoduše mu nebude vytvořena hrana pro nastavení konce transakčního času. Předchozí příklad po této úpravě



Obrázek 2.9: Oprave datové redundance z obrázku 2.8



Obrázek 2.10: Oprava prázných uzlů z obrázku 2.9



Obrázek 2.11: Graf s uzly obsahující stejnou část temporální složky (`TRAN_END`)

by vypadal následovně (2.12)

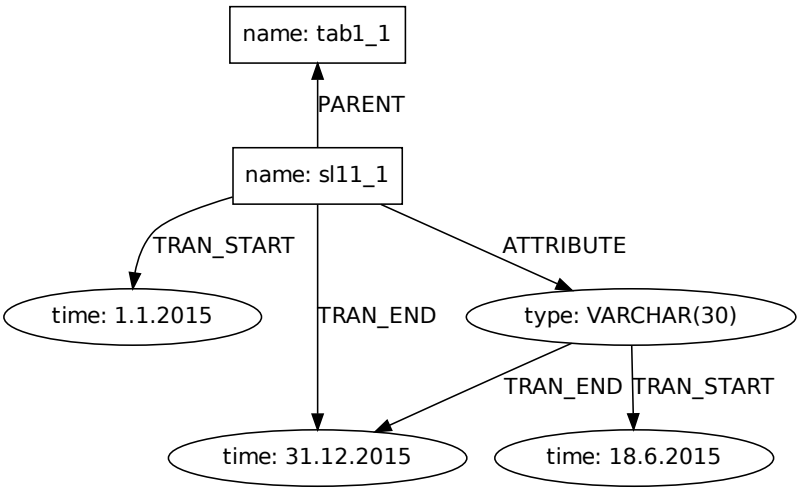
Tímto řešením nám takto vzniká množina časových bodů.

2.1.5 Diskrétní časová osa

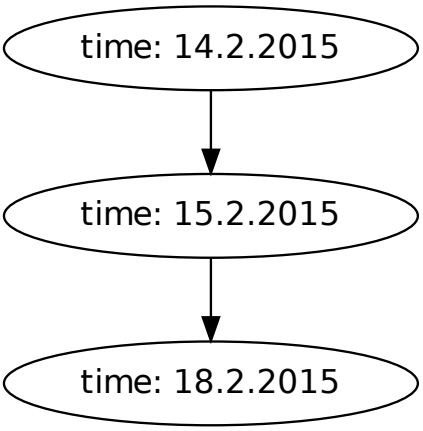
Dostáváme se tedy k řešení, kdy jsou všechny časové okamžiky reprezentovány uzlem. Množina těchto uzlů reprezentuje množinu časových okamžiků. Tyto časové okamžiky však mezi sebou nemají žádné vazby. Pokud bychom se dotazovali na časový interval, museli bychom procházet všechny časové uzly a na každém kontrolovat, zda patří do vyhledávaného časového intervalu. Složitost tohoto vyhledávání by se však dala snížit vytvořením vazeb mezi těmito uzly.

Jakmile by došlo k datové události (např. spuštění skriptu), vytvořil by se uzel reprezentující časový okamžik. Další událost by vedla k vytvoření nového uzlu, který by bylo propojen se starým uzlem (směr od staršího uzlu k novějšímu). Tím nám vznikne posloupnost časových okamžiků, jejíž body budou referencovány datovými uzly, a která může vypadat následovně (2.13).

Pomocí tohoto konstruktu je možné procházet jednotlivé časové okamžiky a k nim navázané ovlivněné databázové objekty.



Obrázek 2.12: Oprava stejné temporální složky z obrázku 2.11



Obrázek 2.13: Posloupnost časových okamžiků

2.1.6 Problémy diskrétní časové osy

Diskrétní časová osa však s sebou nese problémy, které jsou blíže popsány v následujících podkapitolách.

2.1.6.1 Náročnost průchodu časovou osou

Prvním problémem, který je potřeba brát v potaz, je možná paměťová a výpočetní náročnost procházení samotné časové osy. Pokud by teoreticky byly skripty spouštěny příliš často, tak by se během delšího časového intervalu vytvořila dlouhá linie, kterou by bylo potřeba traverzovat uzel po uzlu, jelikož na časové ose neexistují vazby mezi více uzly naráz. Řešením tohoto problému je tvořit hrany s určitou časovou granularitou. Tím je myšleno, že by se vytvářely dodatečné hrany, které by například spojovaly první časové uzly vytvořené v daný den. Lépe je situace patrná z následujícího obrázku (2.14), kde jsou vytvořeny hrany pro přechod mezi měsíci. Tyto hrany by šlo tedy použít s libovolnou granularitou časových jednotek (tj. mezi dny, týdny, měsíci, lety).

Další možností je tvořit hrany pro určitý počet časových okamžiků (např. vytvořit hranu po sto okamžicích). Situaci je lépe patrná z následujícího obrázku (2.15), kde jsou vytvořeny pomocné hrany mezi čtveřicemi uzlů.

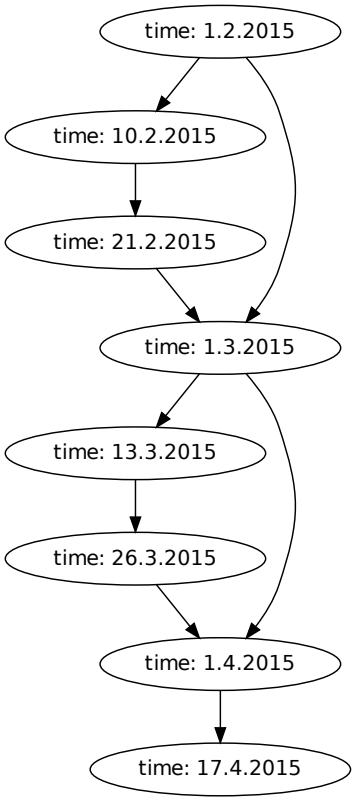
Výše zmíněné dvě řešení vedou sice k vyšší paměťové náročnosti, ale výpočetní náročnost procházení časové osy se tím může poměrně dramaticky snížit, ovšem je potřeba správně zvolit časovou granularitu pro tvoření pomocných hran vzhledem k intenzitě datových operací.

2.1.6.2 Dohledání prvního uzlu v časové ose

Pokud bychom pracovali s časovou osou tak, jak je uvedeno výše, byla by výpočetní náročnost operace dohledání změn vzhledem k datu příliš vysoká z důvodu neefektivního dohledání uzlu s počátkem časové osy. Jednoduché řešení je analogické podobnému problému původního grafu - dohledání uzlu `SUPER ROOT`. Stačí vytvořit podobný kořenový typ pro časovou osu (např. `TIMELINE_ROOT`) a ten zaindexovat, aby databáze dohledala tento uzel v co nejkratším čase.

2.1.7 Optimalizace vzhledem k typům dotazů

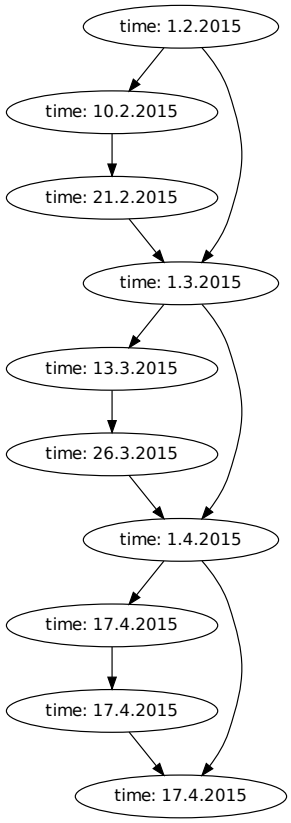
Jelikož se v aplikaci Manta předpokládá, že dotazy budou z většiny vzhledem k stávajícímu času (tj. dotazy budou na aktuální stav), je vhodné výše zmíněné řešení navrhnout tak, aby co nejméně ovlivnilo dotazy, které se temporálními daty nezabývají. Ve výše navrženém řešení se v případě změny uzlu vytváří nový uzel, který se propojí se svým předkem a s časovou osou. Pokud by se například změnil 1000x datový typ sloupečku v tabulce, bylo by vytvořeno 1000 nových uzlů. V případě procházení grafem a hledání aktuální



Obrázek 2.14: Posloupnost časových okamžiků

verze by pak bylo nutné přes všechny uzly proiterovat, na každém uzlu zjistit, zda má nastaven konec platnosti a pokud nemá, tak ho považovat za aktuální a teprve poté by bylo možné pokračovat v průchodu grafem.

Řešením může být vytvoření pomocného příznaku `ACTUAL` na každé hraně, který nabývá hodnot `TRUE` nebo `FALSE`. Aktuální verze by tak držela příznak `TRUE`, v případě změny by se hraně k předkovi nastavil příznak na `FALSE` a nově vytvořená hrana propojující nový uzel a předka by si nastavila tento příznak na `TRUE`. V následných dotazech by pak stačilo kontrolovat příznak odchozích hran. V případě smazání aktuálního uzlu by se hraně nastavil příznak na `FALSE`, uzel by tedy nebyl propojen s potomkem stejného typu hranou s hodnotou příznaku `ACTUAL` na `TRUE`.



Obrázek 2.15: Posloupnost časových okamžiků

2.1.7.1 Počet hran uzlů diskrétní časové osy

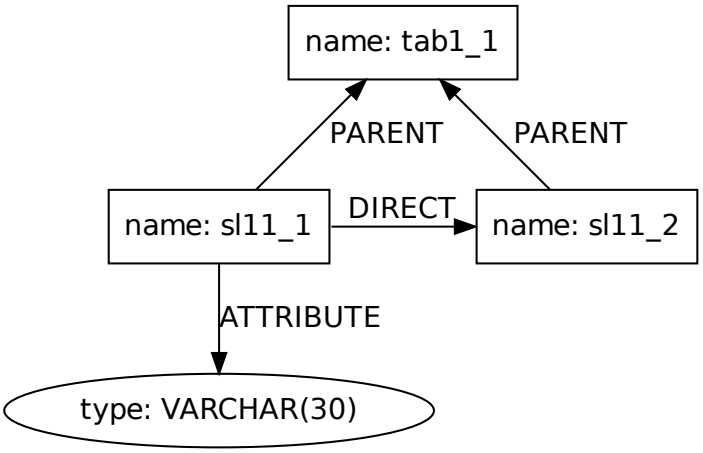
Druhým a poměrně zásadním problémem je možnost obrovského počtu hran směřujících k jednomu časovému okamžiku. Pokud by například nějaký rozsáhlý skript vytvořil v grafu 10000 uzlů, bude zároveň vytvořeno 10000 hran do časového okamžiku. Problémem je jak paměťová náročnost, tak následná výpočetní náročnost v případě dotazování nad daným časovým okamžikem. Řešení tohoto problému bude rozebráno dále v práci, jelikož se jedná o poměrně zásadní problém.

2.1.8 Temporální reprezentace vzhledem k datovým tokům

Výše zmíněná část analýzy se věnovala způsobu reprezentaci temporálních dat datových uzlů a atributů. Cílem této části je rozebrat možnosti reprezentace temporálních dat datových toků.

2.1.8.1 Reprezentace datových toků

Jak již bylo zmíněno v kapitole 1.5.4.2, hrany mohou být v grafu i horizontálně. Tyto hrany reprezentují skutečnost, že jeden databázový objekt ovlivňuje druhý databázový objekt. Příkladem může být následující graf (2.16).



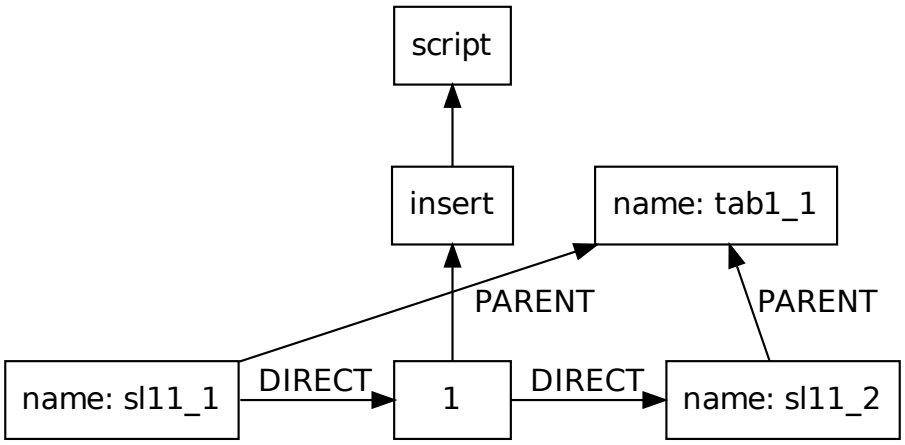
Obrázek 2.16: Příklad hrany datového toku

V grafu se však drží další databázové objekty. Pokud například databázový skript vyvolá insert skript, který vkládá do sloupečku `sl11_2` data vytažená ze sloupečku `sl11_1`, reprezentace v grafu může vypadat následovně (2.17).

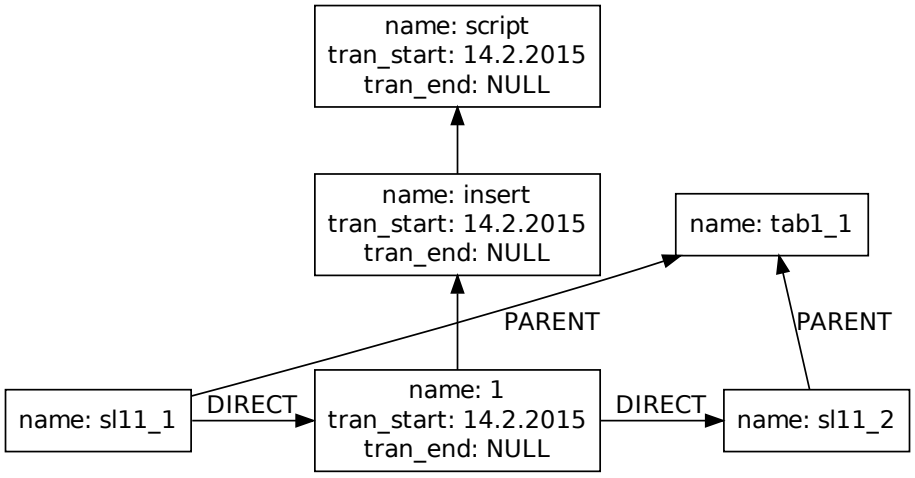
2.1.8.2 Reprezentace temporality datových toků

Zde máme také možnosti reprezentace dat na uzlu, hraně a jako samostatný uzel.

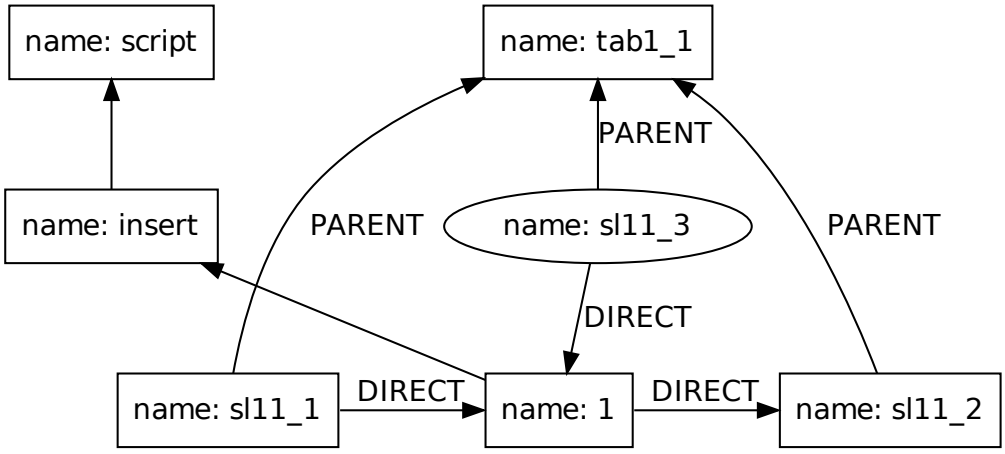
- Repererzntace na uzlu - reprezentace by mohla vypadat následovně (2.18). Ten způsob reprezentace je však nevhodný. Příkladem datových toků, kde by zachycení temporality na uzlu nedávalo smysl, je obrázek 2.19. Zde se berou data ze dvou sloupečků a vkládají se (například pomocí součtu) do posledního sloupce. Temporální data na uzlu by nepostihly fakt a o datových tocích, ale pouze o existenci skriptu (a konkrétního insert příkazu nad sloupečkem). Řešení pomocí duplikací uzlů nepřipadá v úvahu, tato varianta je proto nevhodná.
- Repererzntace na hraně - graf s touto reprezentací by vypadal takto (2.20). Toto řešení se zdá být nejvhodnější. V případě ukončení da-



Obrázek 2.17: Příklad datového toku

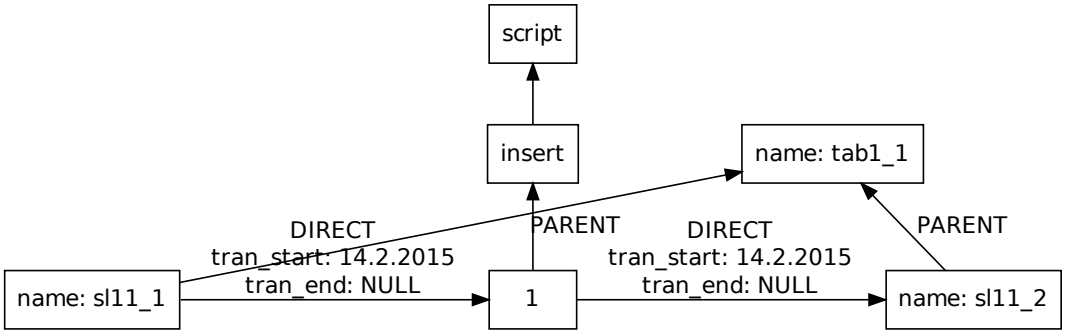


Obrázek 2.18: Reprezentace temporality datového toku na uzlu



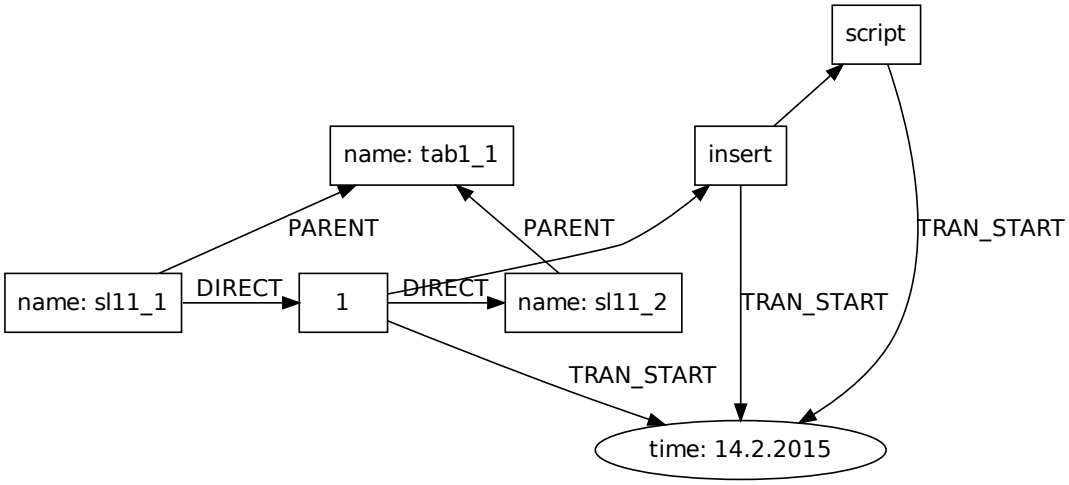
Obrázek 2.19: Nevhodnost temporality datového toku na uzlu

tového taku se nastaví transakční konec na hraně, v případě nových toků se pouze vytváří hrany.



Obrázek 2.20: Reprezentace temporality datového toku na hraně

- Repererntace samostatným uzlem - tato reprezentace je založená na stejné myšlence, jako tomu je u výše zmiňované reprezentace pro datové uzly, tj. časy se odkazují na posloupnost časových bodů (2.21). Jelikož tato reprezentace obnáší stejné problémy jako reprezentace na uzlu, lze ji pokládat za nepřijatelnou.



Obrázek 2.21: Reprezentace temporality datového toku samostatným uzlem

2.1.8.3 Výběr nejvhodnější varianty

Výše zmíněné možnosti reprezentace temporality datových toků obnáší prakticky stejné problémy jako ty, jenž byly zohledněny v reprezentaci temporality datových uzlů. Jako nejvhodnější varianta by se zde vzhledem k existující reprezentaci temporality datových uzlů mohla jevit třetí možnost - reprezentovat data v samostatných uzlech z důvodu využití uzlů reprezentující časové body a případné zmíněné možnosti optimalizace (přidání pomocných hran mezi časovými okamžiky vzdálenými dobu t_x apod.), ovšem z výše uvedených důvodů je v podstatě jedinou schůdnou variantou reprezentace pomocí hran.

2.1.9 Diskuse řešení vzhledem k použitelnosti v projektu Manta

Výše navržené řešení je sice plně funkční, ovšem v projektu Manta (a tedy i obecně) se lze setkat s výkonnostními problémy, které je nutné zohlednit v případných dalších způsobech práce nad grafem. Problém zmíněný v kapitole 2.1.7.1 může vést k zdlouhavému výpočtu, ale zásadním problémem je zde možnost přetečení zásobníku. Aplikace Manta je postavená na Javě, API pro použitou databázi Titan v případě prohledávání grafu poskytuje na uzlu možnost pro získání hran pomocí iterátoru. Java si v případě iterátoru načítá celou kolekci do paměti. V případě velkého množství hran (tak, jak je to zmíněno v kapitole 2.1.7.1) je zde možnost přetečení zásobníku, jelikož se

načtení všech hran nevejde do operační paměti. Důsledkem by v tomto případě byla nemožnost prohledávat diskrétní časovou osu, jelikož zde nejsou hrany odlišeny a při pokusu o průchod dále v časové ose by bylo nutné dotahovat všechny hrany. Tento problém je vzhledem k návrhu počítajícím s velkými objemy dat tak zásadní, že je nutné prodiskutovat všechna řešení a pokusit se vybrat vhodnější variantu, kde bude tomuto problému zamezeno.

2.1.10 Možnost využití indexů

Databáze Titan, nad kterou projekt Manta běží, umožňuje využití dvou typů indexů - standardní index a externí index. [2] Standardní index slouží k zaindexování uzlů nebo hran dle zadání přesného názvu určitého atributu. Externí indexy umožňují komplexnější a flexibilnější práci s hranami a uzly, podporují dotazy vzhledem ke geo-lokaci, číslenému rozsahu a také fulltextové vyhledávání. Titan v současné době podporuje připojení ke dvěma indexovým backendům - Lucene a Elastic Search.

2.1.10.1 Použitelné indexy pro reprezentaci temporálních dat

Externí indexy umožňují použít následující typy indexace:

- Geo-indexy - tento typ je vytvořen za účelem prohledávání s ohledem na prostorovou dimenzi. Funguje na bázi dvou-dimenzionálího prostoru, v němž každý prvek nese 2 atributy (v kontextu geo informací jako longitude a latitude), vzhledem k nimž je index prohledáván. Lze volit dotazy typu 'Který je nejbližší bod vzhledem k zadanému bodu?' nebo 'Kolik se nachází nejbližších bodů ve vzdálenosti ϵ vzhledem k zadanému bodu?'. Jelikož temporální data v našem případě obsahují dvě složky - počátek a konec transakčního času, lze do proměnné osy `x` dosadit hodnotu `TRAN_START` a do proměnné `y` hodnotu `TRAN_END`. Tato reprezentace však není příliš vhodná vzhledem k typům dotazování a možnostem dotazování nad indexem, který je optimalizovaný opravdu pro geolokace. Umožněny by byly například rychlé dotazy nad dobou platnosti určitého uzlu, ovšem pro získání nejbližších uzlů (circular region search) prakticky v našem projektu není použití, operace bounding box search[1] je podobně nepoužitelná. Celkově se tak je použití geo indexu spíše jako hackování a přiohýbání funkcionality pro naše potřeby, tento index lze považovat pro projekt Manta za nevhodný.

- Numeric range indexy - tyto indexy slouží primárně k dotazování nad uzly/hranami dle podmínky, zda hodnota uzlu/hrany indexované atributy

[1] Metoda v literatuře známá jako vyhledání MBR - minimal bounding rectangle (v češtině jako minimální ohraničující obdélník) je takový minimální obdelník, který zcela obsahuje jeden nebo více objektů v 2D prostoru

patří do zadaného intervalu. Tato varianta se zdá být nejvhodnější, jelikož lze očekávat dotazy typu 'Jak vypadal stav grafu k určitému datu?' - interval by se tedy ohraničil shora i zdola stejným datem, vzhledem ke kterému by pak bylo nad uzly/hranami dotazováno.

- Full-text search index - tento typ indexu se v současné chvíli používá pro full-textové vyhledávání nad názvy uzlů v grafu. Pro případ vyhledávání nad temporálními daty je však prakticky neaplikovatelný a nevhodný.

2.1.10.2 Rozbor vhodnosti intervalového indexu

Z existujících typů indexů implementovaných indexovými backendy se zdá být nevhodnější varianta numeric range indexu. Dle dokumentace na [1] není ani nutné převádět datové typy, protože range queries umí pracovat přímo s časovými objekty, ovšem z uživatelského hlediska lze očekávat nejčastěji dotazy typu "Jaké byly změny v daných revizích"? Z tohoto hlediska se zdá být lepší, když bude temporalita reprezentována číslem revize. Jelikož je velikostně integer menší než objekt Date a temporální data budou na všech hranách (kterých lze očekávat řádově miliony), dojde i k úspoře prostoru. Dotazy pro vyhledání by tedy vypadaly následovně:

1. Vybere se revize, ke které se mají zobrazit změněné (nové, upravené, smazané) databázové objekty
2. Vyberou se typy uzlů, vzhledem ke kterým se mají data dohledávat - tímto se omezí možnost nedostatku paměti pro případné dohledání příliš velkého počtu hran
3. Databázové objekty drží na hraně nebo uzlu časové údaje. Přes tyto údaje je vytvořen index pro rychlé vyhledávání
4. Index backend provede prohledání indexu a nalezne všechny hrany s propery `TRAN_START` nebo `TRAN_END` rovné zadané revizi + které jsou navázané na určitý typ uzlu (například vyhledání hran, které vedou od uzlu typu tabulka)
5. Dohledané objekty databáze vrátí

2.1.10.3 Reprezentace temporálních dat vzhledem k možnosti využití indexů

Jak již bylo zmíněno, databáze Titan poskytuje indexaci nad uzly i hranami. Proto přichází v potaz využití indexů jak pro reprezentaci temporálních dat na hraně, tak reprezentaci temporálních dat na uzlech. Samostatné uzly nebudou probírány, jelikož řešení postavené nad samostatnými uzly a popsané výše vede k zásadnímu výkonnostnímu problému, který nelze pomocí indexů

vyřešit, jelikož indexace nemůže vyřešit problém superuzlu, tedy uzlu, na nějž je navázáno takové množství hran, které se nevejde do operační paměti.

V úvahu tedy připadají dvě následující možnosti, reprezentovat data na uzlu tak, jak je to popsáno v kapitole 2.1.3.1 nebo na hraně tak, jak je popsáno v kapitole 2.1.3.2. Jelikož zde byla reprezentace pomocí hran vyhodnocena jako mírně vhodnější, bude pro konečnou reprezentaci zvolena tato možnost. Tato možnost sice částečně omezí možnosti dohledání takového počtu databázových objektů, který se nevejde do operační paměti, ovšem nezabrání mu úplně. S přihlédnutím na stávající chování uživatelů lze očekávat, že prohlížení všech ovlivněných databázových objektů v rámci jedné revize nebude tak časté, jako prohlížení obecného shrnutí databázové revize. Vzhledem k tomuto faktu lze řešit tento problém vyhodnocením statistik během operací a jejich následnému uživatelsky přívětivému zápisu. Během operace merge tedy můžou být počítány statistiky o revizi, jako například počet ovlivněných objektů, kolik je nových, upravených a smazaných objektů, kolik bylo ovlivněno jednotlivých typů objektů apod.

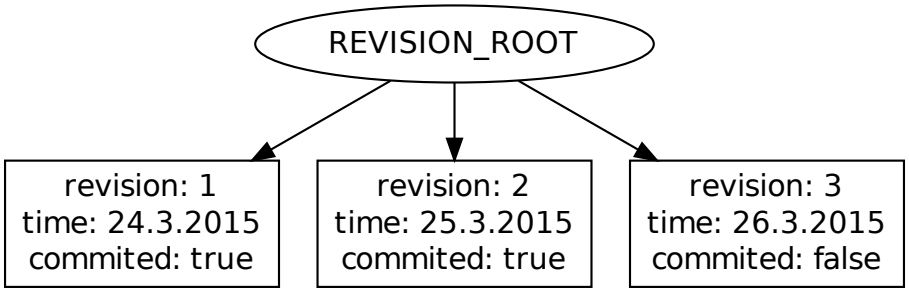
2.1.11 Verzování celé repository

Dalším problémem, který je potřeba vzhledem k zavedení temporality vyřešit, je verzování. Klienti zpracovávající skripty, vytváří z nich grafy a ty odesílají na server, kde se dílčí grafy (reprezentující skripty) mergují do globálního grafu (reprezentující veškeré datové toky) a vytváří tak verze grafu. Nástroj Manta by měl držet informace o těchto verzích - tedy číslo verze (podobně jako je to u stávajících verzovacích systémů), čas počátku existence nové verze a příznak, zda byl přenos nové verze od klienta dokončen.

Pro tyto informace je vhodná reprezentace pomocí uzlů reprezentujících verzi. Řešení je podobné jako navržené řešení pro reprezentaci temporality zdrojových uzlů pomocí samostatného uzlu, ovšem uzly zde nebudou vzájemně propojeny, ale budou spojeny s kořenovým uzlem pro graf verzí. Tento kořenový uzel bude mít z důvodu rychlého dohledání zaidexovaný atribut (`REVISION_ROOT`). Graf s reprezentací verzí repozitáře tak bude kompletně oddělen od stávajícího grafu. Graf s reprezentací verzí může vypadat následovně (2.22):

2.1.12 Vznik a zánik objektů při nahrávání nové verze repository

V případě operace MERGE vznikají a zanikají databázové objekty. Načítání většího grafu se může pohybovat v řádu minut až hodin. Pokud si klient vyžádá aktuální verzi grafu, server mu začne posílat data, mezitím dojde k úpravě grafu na serveru, tak je klientovi zasílán nekonzistentní graf. Řešením tohoto problému je postupovat následovně:



Obrázek 2.22: Příklad uzlů reprezentujících verze repozitáře

1. Klient si zažádá o novou revizi
2. Server zjistí, zda revizní uzel s nejvyšší revizí je ve stavu `COMMITED`. Pokud ano, vytvoří nový uzel s číslem revize vyšším o jedna a toto číslo vrátí server klientovi
3. Klient vezme toto číslo a pošle serveru požadavek na merge (v rámci jednoho MERGE vícekrát) s číslem revize
4. Server tyto požadavky přijme, zkontroluje, zda je nejnovější revize rovna revizi zaslané klientem a současně zda revizní uzel reprezentující tuto revizi není ve stavu commited. Pokud jsou tyto podmínky splněny, tak se grafy spojí
5. Po dokončení spojování grafů pošle klient na server požadavek na COMMIT
6. Server tento požadavek přijme, najde dle zaslané revize patřičný revizní uzel a následně zkontroluje, zda není commitnutý. Pokud ne, tak nastaví příznak `COMMITED` na true.

Tímto postupem jsou vyřešeny tyto problémy

- pokud klient pošle požadavek na novou revizi a poslední revize není ve stavu commited, tak server vyhodí chybu a nový revizní uzel nevytvoří
- pokud klient pošle požadavek na merge s vyšší nebo nižší revizí, než je nejvyšší revize revizního uzlu v grafu, tak server vyhodí chybu a operaci merge neprovede

- pokud klient pošle požadavek na merge s číslem revize, která už je commitnuta, tak server vyhodí chybu a merge neprovede
- pokud klient pošle požadavek na commit verze, která už je ve stavu commited, nebo se snaží commitnout revizi, která ještě neexistuje, server vyhodí chybu a neudělá nic

2.1.13 Transakční platnost databázových objektů

Transakční data jsou uložena jako atributy hran. Temporalita uzlů se tedy musí určit z přilehlých hran. Transakční platnost jednotlivých typů uzlů k určité revizi nebo k určitému reviznímu intervalu bude tedy určena následujícím způsobem

- SUPER_ROOT - jelikož se při operacích kontroluje existence tohoto uzlu a tento uzel se zároveň vytváří při inicializaci databáze, lze předpokládat existenci tohoto uzlu ve všech revizích. Dotaz na existenci super rootu v dané revizi nebo v daném revizním intervalu tedy vždy vrátí hodnotu TRUE
- RESOURCE - existence uzlu typu RESOURCE se bude kontrolovat na základě typu hrany hasResource a směru ven z uzlu (tato hrana směřuje do super rootu)
- NODE - existence uzlu typu NODE bude kontrolována přes hrany hasResource a hasParent směrem ven.
- ATTRIBUTE - existence tohoto typu uzlu bude kontrolována přes typ hrany hasAttribute směrem dovnitř atributu

2.1.14 Problémy s verzováním a délkou operací

V rámci verzování narážíme na problémy související s problematikou datových skladů.

Možné problémy a jejich řešení:

- Přišel požadavek na server pro merge grafu, to se zahájilo, ovšem před dokončením přišel požadavek na merge nové verze - tento problém není nereálný, vzhledem k délce trvání operace merge (řádově i hodiny). Lze:
  - Zahájit merge a vytvořit nové revize pro oba grafy - tato možnost by vedle k nekonzistenci, jelikož nové revize předpokládají návaznost na starší revize, tato možnost tedy nepřipadá v úvahu
  - Dát požadavek do fronty, provést merge prvního grafu a po dokončení zahájit merge prvního grafu s druhým - v případě nedokončení operace už by nebylo možné provést merge grafů - to by šlo vyřešit

timeoutem pro merge, ovšem ten nelze stanovit obecně, jelikož velké grafy se musí zákonitě spojovat dlouho, tato možnost tedy také nepřipadá v úvahu

– Novější požadavky odmítat, dokud nebude dokončen merge starší revize - nejjednodušší ale zároveň nejbezpečnější řešení. V případě probíhající operaci merge na serveru bude klientovi při požadavku na merge zaslána zpráva o nemožnost provedení.

- Přijde pokus o další data pro starou verzi - v případě, že operace zrovna probíhá se požadavek jednoduše odmítne
- Uživatel si chce prohlížet data a běží load - čtení v takovémto případě bude probíhat pouze ze starých, commitnutých verzí

Pozn.: Pro operaci LOAD se předpokládá, že v rámci jedné revize se bude provádět nahrávání celého repozitáře. Při operaci merge mohou nastat následující situace:

- Uzel neexistuje, ale ve spojeném grafu uzel je - vytvoří se uzel, nastaví se mu transakční počátek a transakční konec roven aktuální verzi.
- Uzel již existuje, v spojeném grafu je uzel také - algoritmus pro merge najde stejný uzel (podle společných předků, názvu a typu uzlu), zvýší se číslo poslední revize, dojde tedy k inkrementaci temporálního atributu `TRAN_END`, do níž se uloží číslo poslední (aktuální) revize.
- Uzel již existuje, v spojeném grafu už uzel není - tento případ znamená, že uzel byl odstraněn. V takovémto případě neměníme uzlu temporální atribut `TRAN_END`

2.2 Závěr analýzy

Vzhledem k zhodnocení zmíněných možností reprezentace, paměťové a výpočetní náročnosti nad temporálními daty a také očekávaným typům dotazům nad temporálními daty vypadá konečná reprezentace následovně:

- Temporální data pro datové uzly budou držena na hranách a zaindexována pomocí intervalového indexu. Temporální složkou na hranách nebude čas, ale čísla první (`TRAN_START`) a poslední (`TRAN_END`) revize (tj. první revize, kde uzel vznikl a buď číslo revize, kde uzel zanikl, nebo číslo aktuální revize). Tyto atributy budou mít datový typ Integer vzhledem k použití intervalového indexu.
- Pro revize repozitáře budou existovat uzly mimo základní graf. Každou revizi bude reprezentovat jeden uzel, ve kterém budou uložena metadata o revizích (číslo revize, datum vytvoření) a který bude napojen na

kořenový uzel pro verze (`REVISION_ROOT`). Tyto uzly nebudou napojeny na původní graf a samy budou tvořit malý graf.

- Kořenový uzel pro revize `REVISION_ROOT` bude mít příznak `REVISION_NODE` s hodnotou true. Ostatní uzly grafu budou mít tento příznak s hodnotou false. V grafu bude existovat index přes tento atribut, vzhledem k rychlosti dohledání kořenového uzlu pro revize.
- Temporální data pro datové toky budou taktéž držena na hranách.

Dále byly vzhledem očekávaným uživatelským akcím k možným výkonnostním problémům stanoveny tyto omezující předpoklady:

- Vzhledem k charakteru dat je implementována pouze druhá temporální dimenze - tedy s ohledem na transakční čas. Procesní metadata v této práci implementována nebudou.
- V rámci jedné revize se bude provádět nahrávání pouze celého repozitáře
- Současný návrh umožňuje dotazy na všechny změny v časovém okamžiku, ovšem v rámci této práce tato funkcionalita nebude implementována, jelikož by mohla přinášet možné výkonnostní problémy týkající se nedostatku paměti

KAPITOLA 3

Realizace

3.1 Cíle

Cílem realizace je naimplementovat funkcionalitu rozebranou v analýze. Cílem této kapitoly je seznámit čtenáře se způsobem, jakými byly klíčové prvky implementovány v jednotlivých modulech.

3.2 Implementace

Tato sekce slouží k popisu jednotlivých funkcionalich v rámci daných modulů s uvedením příkladů nového Java kódu.

3.2.1 Connector

V rámci implementace došlo k největším změnám v rámci modulu Connector. Tento modul zastřešuje základní grafové operace. Mezi ně patří ty, které se týkají tvorby grafu, tj. inicializace, vytvoření kořenového uzlu, vytvoření potřebných indexů ale také pomocné metody pro tvorbu grafu, jako například vytváření nových uzlů typu resource, node, attribute a také nových hran.Třída obhospodařující tuto funkcionalitu se nazývá GraphCreation.

Zde patří mezi klíčové úpravy zavedení nového externího indexu nad novými atributy hran - transakčním počátkem a transakčním koncem. Vytvoření indexu se dosáhlo pomocí následujícího kódu:

```
if (!indexTransaction.getIndexedKeys(Edge.class).
        contains(EdgeProperty.TRAN_START.t())) {
    try {
                indexTransaction.makeKey(EdgeProperty.TRAN_START.t())
                        .dataType(Integer.class)
                        .indexed("search", Edge.class).make();
```

```
                LOGGER.info("Create index for " +
                    EdgeProperty.TRAN_START.t());
    } catch (UnsupportedOperationException e) {
                LOGGER.error("Error during creating index.", e);
    }
}
```

Později (během měření) byly zjištěny výkonnostní problémy, které byly řešeny vytvořením tzv. vertex-centric indexu. Tento typ indexů byl implementován následujícím způsobem.

```
TitanKey tranStartKey = null;
  if (!indexTransaction.getIndexedKeys(Edge.class)
      .contains(EdgeProperty.TRAN_START.t())) {
    try {
      tranStartKey =
        indexTransaction.makeKey(EdgeProperty.TRAN_START.t())
          .dataType(Integer.class)
          .indexed("search", Edge.class).make();
      LOGGER.info("Create index for " + EdgeProperty.TRAN_START.t());
    } catch (UnsupportedOperationException e) {
      LOGGER.error("Error during creating index.", e);
    }
}
// ...
if (tranStartKey != null && tranEndKey != null) {
  indexTransaction
  .makeLabel(EdgeLabel.HAS_PARENT.t())
  .sortKey(keyChildName, tranStartKey, tranEndKey)
  .manyToOne()
  .make();
}
```

Další klíčovou funkcionalitou je zavedení třídy, jejíž úlohou je práce s revizním kořenovým uzlem - `REVISION_ROOT`. Tato funkcionalita byla implementována dle vzoru třídy SuperRootHandler, která slouží k práci s kořenovým uzlem `SUPER_ROOT`.

Nová třída v sobě obsahuje metody pro vytvoření a kontrolu existence uzlu, ale také logiku vytváření uzlů typů `REVISION_NODE`. Příkladem z této třídy je například následující metoda sloužící k získání instance revizního kořenového uzlu pomocí callbacku.

```
/**
 * Ziska Vertex revision root databaze vice viz {@link
     RevisionRootHandler#getRevisionRoot(TitanTransaction)}.
 * @param databaseHolder service pro ziskani transakce
 * @return id revision rootu databaze
 */
public synchronized Long getRevisionRoot(DatabaseHolder
    databaseHolder) {
```

```
        return (Long) databaseHolder.runInReadTransaction(new
            TransactionCallback<Vertex>() {
                @Override
                public Vertex callMe(TitanTransaction transaction) {
                        return getRevisionRoot(transaction);
                }

                @Override
                public String getModuleName() {
                        return MODULE_NAME;
                }
        }).getId();
}
```

Jednoznačně nejvíce úprav bylo nutné provést ve třídě GraphOperation. Tato třída, jak již název napovídá, slouží k základním operacím nad grafem - například získání předka daného uzlu, získání všech atributů uzlu, přímých potomků, celého podstromu, uzlu typu resource, na který je daný uzel navázán, ale také metodu určující, zda zadaný uzel je listový uzel apod. Tyto metody bylo nutné v rámci implementace upravit tak, aby se prováděly vzhledem k určité revizi. Tedy například získání všech atributů uzlu v revizi r_x.

Ukázkou tedy může být následující metoda, v níž je zohledněna temporální složka a která zároveň využívá indexu.

```
/**
 * Metoda vyhleda pomoci indexu vsechny uzly navazane na urcity uzel
     v ramci dane revize.
 * @param node uzel, k nemuz se uzly vyheldavaji
 * @param revision cislo revize
 * @param direction smer hran
 * @param labels typy hran, pres ktere se ma hledat
 * @return
 */
public static Iterable<Vertex>
    getAllRevisionValidAdjacentNodes(Vertex node, Integer revision,
                Direction direction, String... labels) {

        if (node == null) {
                throw new IllegalArgumentException("The argument node
                    was null.");
        }
        if (revision == null) {
                throw new IllegalArgumentException("The argument
                    revision was null.");
        }

        return node.query().
                has(EdgeProperty.TRAN_START.t(), Cmp.LESS_THAN_EQUAL,
                    revision).
```

```
                has(EdgeProperty.TRAN_END.t(),Cmp.GREATER_THAN_EQUAL,
                    revision).
                labels(labels).direction(direction).vertices();
}
```

Jelikož bylo potřeba vytvořit další metody pro práci s temporální složkou, byla vytvořena další pomocná třída pro grafové operace. Těmito operacemi je například inkrementace transakčního konce (využitá při operaci MERGE), zjištění, zda uzel nebo hrana existovala v určité revizi apod.

Dále byly v tomto modulu nutné úpravy vzhledem ke změnám rozhraní metod, přidání nového typu uzlu do výčtového typu VertexType a také úprava a přidání jednotkových testů ke kontrole, zda nebyla narušena stávající funkcionalita a ke kontrole správnosti nové funkcionality.

Jádrem logiky temporality je následující metoda, která se provolává při zjištění, zda databázový objekt spadá do určité revize.

```
/**
  * Kontrola, zda zadany revizni interval patri do intervalu
      <TRAN_START, TRAN_END> na hrane
  * @param edge
  * @param revision
  * @return
  */
public static boolean isEdgeRevisionIntervalValid(Edge edge, Integer
    tranStart, Integer tranEnd) {
  if (edge == null) {
    throw new IllegalArgumentException("The argument edge was null.");
  }
  if (tranStart == null) {
    throw new IllegalArgumentException("The argument tranStart was
        null.");
  }
  if (tranEnd == null) {
    throw new IllegalArgumentException("The argument tranEnd was
        null.");
  }
  if (tranEnd < tranStart) {
    throw new IllegalArgumentException("Violation of inequality!
        tranEnd < tranStart");
  }


  Integer edgeTranStart =
      edge.getProperty(EdgeProperty.TRAN_START.t());
  Integer edgeTranEnd = edge.getProperty(EdgeProperty.TRAN_END.t());

  if (edgeTranStart == null || edgeTranEnd == null) {
    throw new IllegalStateException("Edge doesn't have all
        transaction data.");
```

```
  }

  return ( (tranStart >= edgeTranStart) && (tranEnd <= edgeTranEnd));
}
```

3.2.2 Core

Tento modul obsahuje převážně konfigurační třídy (například připojení k databázi). Se zavedením temporální složky bylo nutné přidat nové vlastnosti pro uzly a hrany. Tyto vlastnosti jsou definovány v modulu Core ve třídě popisující databázovou strukturu. Přidání nových vlastností a označení hran bylo dosaženo jednoduchým přidání záznamů do výčtového typu. Příkladem může být následující kód.

```
public enum EdgeLabel {
        ...
        /** Zdroj hrany ma cil hrany jako atribut. */
        HAS_ATTRIBUTE("hasAttribute", "attribute"),
        /** Zdroj hrany ma cil hrany jako revizi. */
        HAS_REVISION("hasRevision", "revision"),
        ...
}
```

3.2.3 Merger

Tento modul, jak název napovídá, slouží k zastřešení operace MERGE. Požadavek na operaci MERGE se prováděl pomocí klientova zavolání RESTové služby na serveru. Toto RESTové rozhraní je nadefinováno pomocí Springovského MVC frameworku. Konfigurace bean je nadefinována v souboru mvc-dispatcher-servlet.xml, který je obsažen také v modulu MERGER. Jelikož stačí uchovávat pouze jednu instanci třídy RevisionRootHandler, je přidán záznam do souboru mvc-dispatcher-servlet.xml, který nastavuje defaultně Java beanu jako singleton. Definice pak v souboru pro testy vypadá následovně:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    ...

    <bean id="superRootHandler" class="eu.profinit.manta.dataflow
        .repository.connector.titan.service.SuperRootHandler" />

    <bean id="revisionRootHandler" class="eu.profinit.manta.dataflow
```

```
        .repository.connector.titan.service.RevisionRootHandler" />
    ...

</beans>
```

V analýze bylo stanoveno, že úvodním krokem je klientovo vyžádání nového čísla revize. Voláním RESTové služby na adrese '/createRevision' na serveru dojde ke kontrole, zda byla poslední verze commitnuta a pokud ano, server založí nový revizní uzel s příznakem COMMITED na false, nastaví tomuto uzlu číslo revize, které je rovno nejvyššímu stávajícímu číslu revize inkrementovaného o 1 a toto číslo klientovi vrátí. RESTová služba pro vyžádání čísla vypadá následovně:

```
/**
  * Vytvori novy revizni uzel v grafu a vrati cislo teto nove revize.
  * @return revize
  */
@Secured(MergerRoles.MERGER)
@RequestMapping(value = "/revision", method = RequestMethod.GET)
public Integer getHeadRevision() {
    return new RevisionHelper(databaseService).createNewRevisioNode();
}
```

Obsah metody createRevisionNode je následující:

```
@Controller
public class MergerController {
        ...
    @Autowired
    private DatabaseHolder databaseService;

    @Autowired
    private SuperRootHandler superRootHandler;

    @Autowired
    private RevisionRootHandler revisionRootHandler;

    @Autowired
    private MergerProcessor mergerProcessor;

    @Autowired
    private RevisionHelper revisionHelper;
        ...

    /**
     * Vytvori novy revizni uzel v grafu a vrati cislo teto nove
         revize.
     * @return revize
     */
    @Secured(MergerRoles.MERGER)
```

```
    @RequestMapping(value = "/createRevision", method =
        RequestMethod.POST)
    public ResponseEntity<Map<String, Object>> createRevisionNode() {
        GraphCreation.initDatabase(databaseService, superRootHandler,
            revisionRootHandler);

        HttpStatus statusCode = HttpStatus.OK;
        Map<String, Object> resultMap = new HashMap<String, Object>();

        try {
            resultMap.put("revision",
                revisionHelper.createNewRevisioNode());
        } catch (IllegalArgumentException e) {
            statusCode = HttpStatus.INTERNAL_SERVER_ERROR;
            resultMap.put("error", "Error during revision retrieval.
                Invalid argument.");
            return new ResponseEntity<Map<String, Object>>(resultMap,
                statusCode);
        } catch (IllegalStateException e) {
            statusCode = HttpStatus.INTERNAL_SERVER_ERROR;
            resultMap.put("error", "Error during revision retrieval -
                invalid application state.");
            return new ResponseEntity<Map<String, Object>>(resultMap,
                statusCode);
        }
        return new ResponseEntity<Map<String, Object>>(resultMap,
            statusCode);
    }
        ...
}
```

Podobným způsobem bylo naimplementována služba pro operaci COMMIT.

Zavedení temporality se netýkalo pouze modelu, ale také způsobu, jakým se prochází graf do hloubky. Různé průchodové algoritmy implementují rozhraní GraphTraverser, jehož metody byly rozšířeny o temporální složku. Nutné bylo samozřejmě upravit všechny průchodové algoritmy.

Příkladem metody rozhraní je následující kus kódu.

```
public interface GraphTraverser {
    /**
     * Provede pruchod grafem vzhledem k urcite revizi a na
         jednotlivych objektech vola visitora.
     * @param visitor pouzivany visitor pro navstevu jednotlivych
         objektu
     * @param superRoot start prohledavani v superrootu grafu
     * @param revision revize, vzhledem ke ktere se graf prochazi
     */
    void traverse(GraphVisitor visitor, Vertex superRoot, Integer
        revision);
```

```
    /**
     * Provede pruchod grafem vzhledem k urcitemu transakcnimu
         intervalu a na jednotlivych objektech vola visitora.
     * @param visitor pouzivany visitor pro navstevu jednotlivych
         objektu
     * @param superRoot start prohledavani v superrootu grafu
     * @param tranStart transakcni pocatek
     * @param tranEnd transakcni konec
     */
    void traverse(GraphVisitor visitor, Vertex superRoot, Integer
        tranStart, Integer tranEnd);

        ...
}
```

Dalšími důležitými opravami byla změna způsobu vyhodnocování, zda databázový objekt již existuje. Následující příklad reprezentuje způsob, jakým se zjišťuje, zda uzel typu resource existoval v poslední revizi. Podmínky na transakční konec jsou záměrně dvě, jelikož lze očekávat během jedné operace merge více stejných objektů (v níže uvedeném případě tedy uzlů typu resource). Podmínka tedy zachytí takové uzly typu resource, které byly v poslední revizi a nebo byly už přidány během probíhající operace merge. Nalezené uzly se porovnají na základě identifikátoru, kterým je v případě uzlu typu resource jméno a typ. Pokud tyto atributy souhlasí, jedná se o ten a samý resource a tím pádem se pouze zvýší transakční platnost inkrementací atributu tranEnd.

```
TitanVertexQuery query = ((TitanVertex) root)
        .query()
        .has(EdgeProperty.CHILD_NAME.t(), resName)
        .has(EdgeProperty.TRAN_START.t(), Cmp.LESS_THAN_EQUAL,
            clientRevision)
        .has(EdgeProperty.TRAN_END.t(), Cmp.GREATER_THAN_EQUAL,
            lastCommitedRevision)
        .has(EdgeProperty.TRAN_END.t(), Cmp.LESS_THAN_EQUAL,
            clientRevision)
        .direction(Direction.IN);

Iterator<Vertex> vertexIterator = query.vertices().iterator();
boolean notYetExist = true;
while (vertexIterator.hasNext()) {
  Vertex checkedVertex = vertexIterator.next();

  if (checkedVertex.getProperty(NodeProperty.RESOURCE_NAME.t())
      .equals(resName)
      && checkedVertex.getProperty(NodeProperty.RESOURCE_TYPE.t())
      .equals(resType)) {

    notYetExist = false;
```

```
    context.getMapResourceIdToDbId().put(resId,
        checkedVertex.getId());

    // kontrola, zda nebyl resource pridan v ramci jednoho merge
    if (!RevisionUtils.resourceExistsInRevision(checkedVertex,
        clientRevision)) {
      RevisionUtils.setResourceTransactionEnd(checkedVertex,
          clientRevision);
    }
  }
}

if (notYetExist) {
  Vertex newNode = GraphCreation.createResource(
      context.getDbTransaction(), root, resName, resType,
          resDescription, clientRevision);
  context.getMapResourceIdToDbId().put(resId, newNode.getId());
  return ResultType.NEW_OBJECT;
} else {
  return ResultType.ALREADY_EXIST;
}
```

3.2.4 Dispatcher

V modulu dispatcher je klíčovým souborem mvc-dispatcher-servlet.xml, který (podobně jako tomu bylo v modluu Merger pro testy) obsahuje definici Java bean. Proto i zde byly přidány záznamy.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    ...

    <bean id="revisionRootHandler"
        class="eu.profinit.manta.dataflow.repository.
               connector.titan.service.RevisionRootHandler" />

    <bean id="revisionHelper"
        class="eu.profinit.manta.dataflow.repository.
               merger.server.helper.RevisionHelper" />
    ...

</beans>
```

3.2.5 Client

V klientském modulu bylo nutné vytvořit nové třídy, reprezentující určité uživatelské akce (tasky). Těmito akcemi je zavolání serverové služby pro vytvoření nového revizního uzlu a také volání služby pro potvrzení určité revize (operace commit). Ukázkou této uživatelské akce je vytvoření nové revize

```
/**
 * Task pro zavolani get requestu na server, kde se vytvori novy
    revizni uzel a vrati
 * se klientovi jeho cislo. Klient toto cislo revize ulozi do souboru
 *
 * @author pholecek
 */
public class NewRevisionTask extends AbstractTask<Null, Null> {
    ...
    /** Url adresa serveru, kam se klient dopta. */
    private String serverUrl;

    /** Jmeno souboru, do nejz se revize zapise. */
    private String revisionFilename;

    /** Objekt pro prihlaseni do repository. */
    private LoginHelper loginHelper;


    @Override
    protected void doExecute(Null arg0, Null arg1) {
        Integer revision = getRevisionOnServer();
        writeRevisionToFile(revision);
    }

    /**
     * Zavola na serveru prikaz pro vytvoreni noveho revizniho uzlu,
        server vrati cislo nove revize
     * a metoda tuto revizi vrati
     * @return cislo vytvorene revize
     */
    private Integer getRevisionOnServer() {
        if (serverUrl == null || serverUrl.trim().equals("")) {
            System.out.println(serverUrl);
            throw new IllegalStateException ("ServerUrl not set");
        }

        HttpClient httpClient = HttpClientBuilder.create().build();

        Integer revision = null;

        if (loginHelper != null && loginHelper.login(httpClient)) {
            HttpPost request = new HttpPost(serverUrl);
```

```
            BufferedReader responseReader = null;
            BufferedWriter outputWriter = null;
            try {
                long start = System.currentTimeMillis();
                LOGGER.info("Sending new revision request to {}",
                    serverUrl);
                HttpResponse response = httpClient.execute(request);
                long end = System.currentTimeMillis();
                if (response.getStatusLine().getStatusCode() ==
                    HttpStatus.SC_OK) {

                    String retSrc =
                        EntityUtils.toString(response.getEntity());
                    revision = new
                        JSONObject(retSrc).getInt(REVISION_JSON);

                    LOGGER.info("OK response: revision {}, time: {}ms",
                        revision, (end - start));

                } else {
                    LOGGER.error("Response was not ok: " +
                        response.getStatusLine().getStatusCode());
                }

            } catch (ClientProtocolException e) {
                LOGGER.error("Error during http comunication. ", e);
            } catch (IOException e) {
                LOGGER.error("Error during saving response. ", e);
            } catch (JSONException e) {
                LOGGER.error("Error during getting json response. ",
                    e);
            } finally {
                IOUtils.closeQuietly(responseReader);
                IOUtils.closeQuietly(outputWriter);
                request.releaseConnection();
            }
        } else {
            LOGGER.error("Cannot process demand url request, because
                login has failed.");
        }
        return revision;
    }


    // gettery a settery
}
```

Uživatelské akce jsou zabaleny do uživatelských scénářů. Jde o jakousi posloupnost uživatelských akcí, která vezme nějaký objekt na vstupu, provede uživatelské akce a pošle objekt na výstup. Další ukázkou je část definice nového

scénáře, který se skládá z tasku pro načtení konfigurace a zavolání tasku pro vytvoření nové revize. Třída task má určité proměnné, které jsou nastaveny pomocí konfiguračních souborů. Příkladem tohoto souboru je následující XML, které nastavuje proměnné z konfiguračního souboru.

```
<bean id="newRevisionScenario"
    class="eu.profinit.manta.platform.automation.NullScenario">
  <property name="name" value="New revision scenario" />
  <property name="tasks">
  <list>
    <bean class="eu.profinit.manta.dataflow.repository.merger.
                    client.NewRevisionTask">
    <property name="serverUrl"
        value="${manta.merger.createRevisionUrl}" />
    <property name="revisionFilename"
        value="${manta.merger.revisionFilename}" />
    <property name="loginHelper">
      <bean
          class="eu.profinit.manta.dataflow.repository.utils.LoginHelper">
        <property name="serverUrl" value="${manta.repository.url}" />
        <property name="loginName"
            value="${manta.repository.loginName}" />
        <property name="loginPassword"
            value="${manta.repository.loginPassword}" />
      </bean>
    </property>
    </bean>
  </list>
  </property>
</bean>
```

Samotné hodnoty proměnných jsou uloženy v souboru config.properties následovně:

```
manta.merger.url=${manta.repository.url}/local
manta.merger.createRevisionUrl=${manta.repository.url}/createRevision
manta.merger.commitRevisionUrl=${manta.repository.url}/commit
manta.merger.truncateUrl=${manta.repository.url}/truncate
manta.merger.propagateEdgesUrl=${manta.repository.url}/propagate-edges
```

Koncový uživatel pak pouze spustí soubor, který tyto scénáře spustí. Spustitelným souborem je klasicky .bat (na platformě Windows) nebo .sh (Unix a Linux). Příkladem jsou tyto spustitelné soubory, které spustí scénáře pro vytvoření nové revize.

Pro platformu Windows:

```
@echo off
echo
    ------------------------------------------------------------------------
echo Manta New Revision Scenario (version ${version})
```

```
echo
    ------------------------------------------------------------------------

echo newRevisionScenario start: %DATE%, %TIME%
cd "%~dp0"
set
    MANTA_LICENSE_LOADER=eu.profinit.manta.platform.licensing.LicenseLoaderImpl
call "%~dp0..\..\..\platform\bin\mantar.bat" ${manta.module.name}
    newRevisionScenario
echo newRevisionScenario end: %DATE%, %TIME%
echo.
```

Pro platformu Unix/Linux:

```
#!/bin/bash
echo
    ------------------------------------------------------------------------
echo "Manta New Revision Scenario (version ${version})"
echo
    ------------------------------------------------------------------------

cd `dirname $0`
SCRIPT_DIR=`pwd`
export
    MANTA_LICENSE_LOADER=eu.profinit.manta.platform.licensing.LicenseLoaderImpl
bash "$SCRIPT_DIR/../../../platform/bin/mantar.sh"
    ${manta.module.name} newRevisionScenario
cd -
echo
```

3.2.6 Závěr realizace

V rámci realizace byla naimplementována funkcionalita definovaná dle analýzy.

KAPITOLA 4

Testování

4.1 Testování aplikace Manta

V této kapitole bude čtenář seznámen se způsoby otestované funkcionality. Budou zde uvedeny druhy testů, jakým způsobem se testovalo a s jakými výsledky.

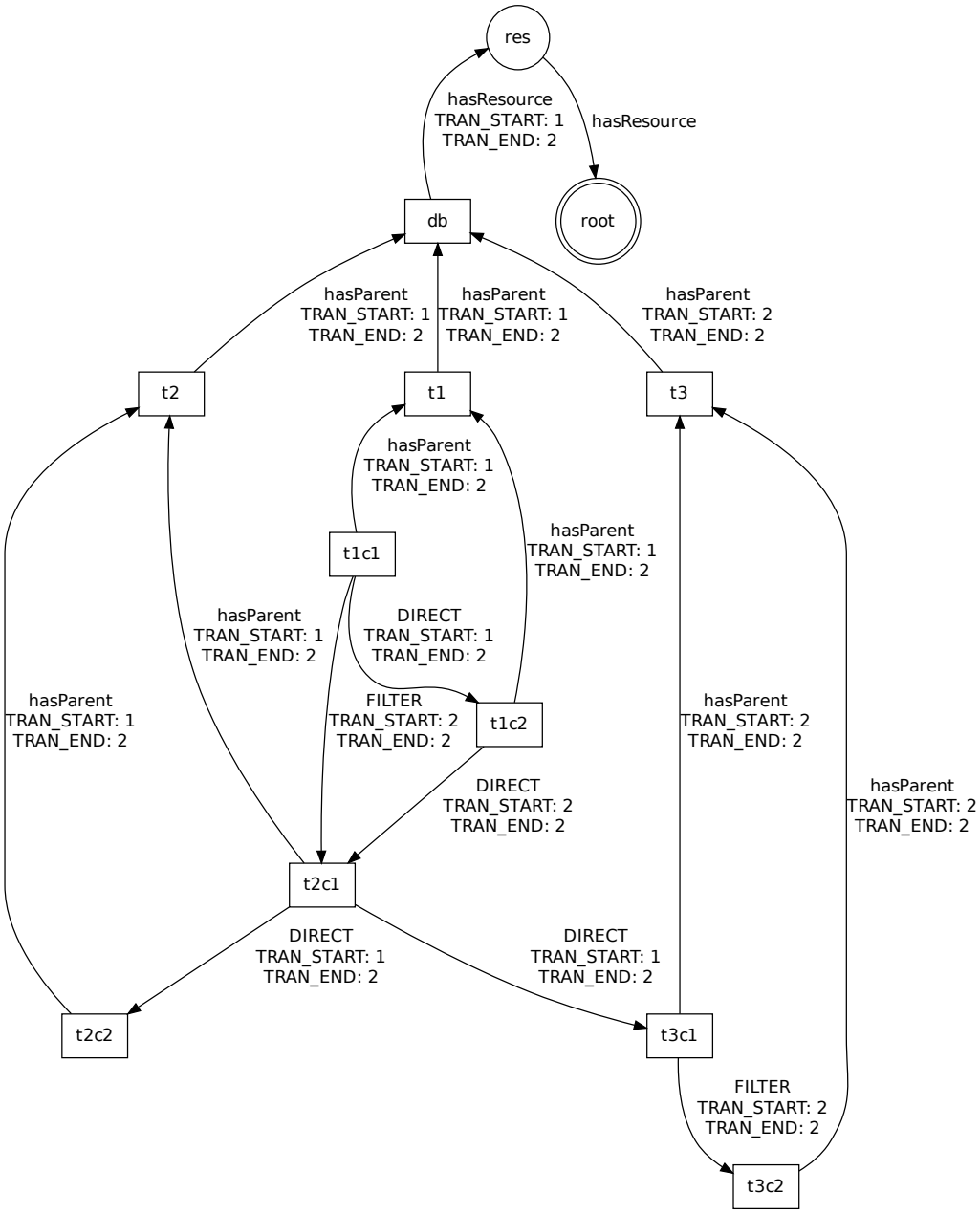
4.2 Jednotkové testy

Jednotkové testy se obecně zaměřují na otestování určité, co nejmenší testovatelné části aplikačního programu. V procedurálním programování bývá touto jednotkou určitá funkce, nebo procedura. V objektově orientovaném programování bývá testovanou částí aplikačního programu metoda (případně celá třída nebo rozhraní).

V aplikaci Manta je v současné chvíli použita knihovna JUnit. Jedná se o standardní knihovnu používanou pro jednotkové testování Java aplikací. Závislost projektu na knihovně je deklarována v souboru pom.xml takto:

```
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.10</version>
        <scope>test</scope>
</dependency>
```

V případě zavádění nové funkcionality do aplikace Manta byly využity existující jednotkové testy pro ověření, zda nebyla ovlivněna stávající funkcionalita. Dále byly vytvořeny nové jednotkové testy k testování operací týkajících se temporality. V rámci jednotkových testů byl vytvořen nový testovací graf, v němž byla provedena v rámci druhé revize změna struktury a datového toku(4.1):



Obrázek 4.1: Graf využitý pro jednotkové testování temporality

Funkcionalita byla testována jak z hlediska tvorby grafu - tj. že se při inicializaci vytvoří nový kořenový uzel pro revize, tak z hlediska metod práce na grafovými objekty, tedy například získání jednoho určitého atributu, všech atributů, všech přidružených uzlů, všech listových potomků, celého podstromu či přímých potomků vzhledem k určité revizi. Příklad testu může vypadat následovně.

```
/**
 * Test atributu uzlu.
 */
@Test
public void testGetAllNodeAttributes() {
        cleanGraph();
        createExtendedGraph();
        getDatabaseHolder().runInWriteTransaction(new
            TestCallback<Object>() {
                @Override
                public Object callMe(TitanTransaction transaction) {
                        Vertex t1 =
                            transaction.getVertices(NodeProperty.NODE_NAME.t(),
                            "table1").iterator().next();
                        Vertex val1 =
                            GraphCreation.createNodeAttribute(transaction,
                            t1, "attr", "val1", REVISION_1);
                        GraphCreation.createNodeAttribute(transaction,
                            t1, "attr", "val2", REVISION_2);
                        Vertex val3 =
                            GraphCreation.createNodeAttribute(transaction,
                            t1, "attr", "val3", REVISION_2);
                        GraphCreation.createNodeAttribute(transaction,
                            t1, "attr", "val4", REVISION_3);
                        Map<String, List<Object>> attrs =
                            GraphOperation.getAllNodeAttributes(t1,
                            REVISION_1);
                        Assert.assertEquals(2, attrs.size());
                        Assert.assertEquals(Collections.singletonList("TABLE"),
                            attrs.get("TABLE_TYPE"));
                        List<Object> list = (List<Object>)
                            attrs.get("attr");
                        Assert.assertEquals(1, list.size());
                        Assert.assertTrue(list.contains("val1"));
                        GraphTransactionEnding.endNode(val1,
                            REVISION_2);
                        attrs = GraphOperation.getAllNodeAttributes(t1,
                            REVISION_2);
                        list = (List<Object>) attrs.get("attr");
                        Assert.assertEquals(2, list.size());
                        Assert.assertTrue(list.contains("val2"));
                        Assert.assertTrue(list.contains("val3"));
```

```
                    ...

                    return null;
                }
        });
}
```

4.3 Testy operací

Tyto testy slouží k ověření funkčnosti vytvoření revizního uzlu, operace merge a potvrzení revize. V rámci těchto testů byly vytvořeny trojice testovacích CSV dat, které server přijímá na vstupu. Každý z těchto CSV souborů reprezentuje jednu revizi. Tyto soubory reprezentující grafy se pak předávají serveru, který je zpracuje (provede merge). Jsou kontrolovány správné počty a stavy uzlů a stav samotného grafu. Příkladem je následující kus kódu.

```
public class MergeProcessorRevisionTest extends
    TestTitanDatabaseProvider {

  @Test
  public void testProcess() throws FileNotFoundException {

  ...

  // vytvoreni revize 1
  revisionHelper.createNewRevisioNode();

  // kontrola zahajeni operace MERGE
  revisionHelper.checkMergeConditions(REVISION_1);

  // v db by mel byt pouze revizni uzel, root a revision root
  getDatabaseHolder().runInWriteTransaction(new
      TestCallback<Object>() {
    @Override
    public Object callMe(TitanTransaction transaction) {
      Assert.assertEquals(3, getVertexCount(transaction));
      return null;
    }
  });

  // nactu prvni CSV
  mergerHelper.process(new FileInputStream(new
      File("src/test/resources/revision_1.csv")), REVISION_1);

  // zkontroluji, zda existuji objekty spolecne pro vsechny revize
  testCommonObjects();
```

```
  // kontrola objektu specifickych pro revizi 1
  testRevision1();

  // commit prvni revize
  revisionHelper.commitRevision(REVISION_1);

  // v db by mel byt po commitu stejny pocet uzlu jako pred commitem
  getDatabaseHolder().runInWriteTransaction(new
      TestCallback<Object>() {
    @Override
    public Object callMe(TitanTransaction transaction) {
      Assert.assertEquals(RevisionTestConstants.REV_1_VERTEX_COUNT,
          getVertexCount(transaction));
      return null;
    }
  });

  // vytvoreni revize 2
  revisionHelper.createNewRevisioNode();

  ...
  }
  ...
}
```

4.4 Testy webového rozhraní

Aplikace Manta umožňuje testovat spojování grafů pomocí webového rozhraní. Toto rozhraní zobrazuje formulář s možností nahrání CSV souboru, které se spojí do existujícího grafu. V rámci těchto testů bylo rozšířen formulář o zadání čísla revize. Po úpravě je tedy možné nahrát soubor k určité revizi. Rozhraní bylo testováno pomocí sad testovacích CSV souborů, na nichž bylo podle výstupu serveru kontrolováno, zda souhlasí počet nových objektů, počet existujících objektů, zda neexistují chybové objekty a také čas operací. Sada testů je součástí příloh.

Touto sadou testů byla úspěšně otestována funkčnost RESTových služeb pro založení nové revize, spojení grafů a také commit revize.

4.5 Testy klienta

Jelikož byl v rámci implementace upraven také klient, byla otestována jeho funkčnost způsobem, jakým s klientem pracují koncoví uživatelé - do adresáře pro vstupní data byl nahrán testování skript a poté byl spuštěn .bat soubor (testováno na OS Windows), který vyvolal scénář pro zpracování skriptu a

vytvoření CSV reprezentace, vytvoření nové revize na serveru, zaslání CSV souboru na server k spojení k dané revizi a commit revize.

Těmito testy byla úspěšně otestována celková funkčnost systému, jelikož zahrnuje jak práci na upraveném klientovi, tak volání služeb pro práci s revizemi a také operace s grafem.

KAPITOLA 5

Benchmarking

Po otestování funkcionality byly provedeny testy pro porovnání časové a paměťové náročnosti operací vůči aplikaci bez zavedené temporality. Operace jsou testovány v případech nahrání nového grafu (měření času a paměti vytváření nových objektů) a nahrání existujícího grafu (měření času a paměti při upravování objektů). Pořadí nahrání grafu je uvedeno v měření ve sloupečcích 'Pořadí' (v případě verze bez temporality) a 'Revize' (verze s temporalitou).

Testy byly prováděny na notebooku se následující konfigurací.

Notebook	Toshiba Satellite 505 -13G
Operační systém	Windows 7 Home (64-bit)
Procesor	Intel(R) Core(TM) i5 M-430, 2.27 GHz
Paměť RAM	4 GB

Použity byly dvě konfigurace indexů

5.0.1 Základní index

Touto variantou je myšleno vytvoření 2 indexů - jeden index pro transakční počátek a druhý index pro transakční konec. Tyto indexy využívají externího index backendu (původně se v projektu Manta využíval backend Lucene[2], ten je v testech použit také, pokud není upřesněno jinak). Důvodem použití externího index backendu je použití tzv. intervalového vyhledávání, zmíněného v kapitole 2.1.10.2, které standardní Titan index nepodporuje.

[2]Backend Lucene se původně využíval jen pro fulltextové vyhledávání podle názvů

5.0.2 Rozšířený index

Touto variantou je myšleno vytvoření 2 indexů - jeden index pro transakční počátek a druhý index pro transakční konec a dále vytvoření tzv. vertex-centric indexů. Tyto indexy se tvoří nad jednotlivými uzly. V této konfiguraci byly vytvořeny 3 nové vertex-centric indexy - pro hrany hasResource, DIRECT a FILTER s řazením přes atributy reprezentující transakční počátek a konec. Dále byl upraven stávající vertex-centric index pro atribut typu HAS_CHILD tak, aby byly hrany řazeny nejenom dle tohoto atributu, ale také dle transakčních údajů.

5.1 Umělá databáze

Testování bylo provedeno nejprve na umělé databázi s použitím základního indexu (5.0.1). Tato databáze zahrnuje 1 resource, na který je navázána jedna databáze. Tato databáze obsahuje 1 schéma, na které je navázáno 50000 databázových procedur. Testy jsou pouze pro první revizi, jelikož se nepředpokládá tento umělý podgraf v provozu. V případě dalších revizí by byl minimální počet operací 50000^2 (kvůli cyklům).

5.1.1 Bez temporality

# db objektů	Pořadí	Čas[ms]	Čas	Celková velikost	Velikost indexu
50000	1	77938	1 min. 17 sec.	87,9 MB	1,27 MB
50000	1	85473	1 min. 25 sec.	48 MB	1,27 MB
50000	1	75380	1 min. 15 sec.	89,1 MB	2,54 MB
50000	1	74992	1 min. 15 sec.	48,1 MB	1,27 MB

Tabulka 5.1: Umělá databáze - měření bez temporality

5.1.2 S temporalitou

# db objektů	Revize	Čas[ms]	Čas	Celková velikost	Velikost indexu
50000	1	124915	2 min. 5 sec.	94,7 MB	2,49 MB
50000	1	134931	2 min. 14 sec.	95,4 MB	2,49 MB
50000	1	130951	2 min. 11 sec.	94,8 MB	2,49 MB
50000	1	138750	2 min. 18 sec.	96,1 MB	2,49 MB

Tabulka 5.2: Umělá databáze - měření s temporalitou

5.2 Části produkční databáze

Pro objektivní zhodnocení paměťové a výpočetní náročnosti je nejvhodnější možností testovat výkon na reálných datech. Protože je k dispozici produkční databáze, která má přes 4,5 milionu objektů, bylo jako nejvhodnější variantou zvoleno testovat na částech této produkční databáze.
Produkční databáze je ve formě CSV souboru o více než 4,5 milionu řádcích (každý řádek reprezentuje 1 databázový objekt). Pro testy byly záměrně zvoleny velikosti 100, 1000, 5000, 10000, 50000 a 100000 řádků pro porovnání řešení s temporalitou a bez temporality. Způsob získání dat z produkční databáze je popsán v příloze v kapitole A.4, index byl použit základní (5.0.1).

5.2.1 Test prvního načtení grafu

Tyto testy porovnávají časovou a výpočetní náročnost operace vytvoření nového grafu - tedy první načtení dat. Konkrétní hodnoty jsou zaneseny do tabulek v příloze, kapitola A.1.

5.2.2 Test druhého načtení grafu

Tyto testy porovnávají časovou a výpočetní náročnost operace merge druhého grafu, který je identický s prvním grafem - veškeré uzly jsou vyhodnoceny jako existující. V případě verze bez temporality byla zaslána stejná data, v případě s temporalitou byla zaslána také stejná data, ale číslo revize se zvedlo o 1. Konkrétní hodnoty jsou v příloze v kapitole A.2.

5.3 Produkční databáze

Měření bylo dále prováděno na produkční databázi s použitím základního indexu (5.0.1). Tato databáze v sobě obsahuje 992276 atributů, 1195766 uzlů, 2376791 hran, a 19 resource, tedy celkově 4564852 objektů. Délku trvání operací a paměťovou náročnost popisuje následující tabulka:

5.3.1 Bez temporality

Typ db	Pořadí	Čas[ms]	Čas	Celková velikost	Velikost indexu
Produkční db	1	3984381	1 hr. 6 min.	1,35 GB	27,9 MB
Produkční db	1	3780781	1 hr. 3 min.	1,35 GB	27,9 MB
Produkční db	2	nedoběhlo	4 hr. +	x	x

Tabulka 5.3: Produkční databáze - měření bez temporality

5.3.2 S temporalitou

Typ db	Revize	Čas[ms]	Čas	Celková velikost	Velikost indexu
Produkční db	1	8337033	2 hr. 19 min.	1,66 GB	139 MB
Produkční db	1	8516788	2 hr. 22 min.	1,66 GB	139 MB
Produkční db	2	nedoběhlo	4 hr. +	x	x

Tabulka 5.4: Produkční databáze - měření s temporalitou

Jak je vidět v tabulkách, ani v jednom případě nebylo dokončeno spojení identických grafů do 4 hodin, proto se s měřením dále nepokračovalo. V případě verze bez temporality jde o očekávané chování. Důvodem je fakt, že v původní verzi nedocházelo ke spojování celých grafů, ale pokud se nahrál nový graf, tak se původní graf smazal a vytvořil se nový, spojování celých grafů tedy nebylo očekávané.

5.4 Zhodnocení měření

5.4.1 Umělá databáze

Výpočetní náročnost se zvýšila zhruba o 55%. Paměťovou náročnost a velikost indexu z tohoto příkladu lze jen těžko stanovit, jelikož si databáze Titan během operací ukládá 2 soubory, z nichž jeden po určitém počtu operací spojí s druhým. Velikost výsledného souboru však není rovna součtu velikostí těchto dvou souborů, ale podstatně nižší. Z tohoto důvodu měření paměti na umělé databázi vykazují odchylky.

5.4.2 Části produkční databáze

Pro vyhodnocení a porovnání rozdílnosti paměťové a výpočetní náročnosti byly vyprůměrovány hodnoty z výše uvedených měření. Data byla zanesena pro lepší přehlednost do grafů. Průměrné hodnoty jsou obsaženy v následujících tabulkách.[3]

5.4.2.1 S temporalitou

Nahrání nového grafu

[3]Hodnoty byly v případě malých rozdílů zaokrouhleny

# db objektů	Revize	Čas[ms]	Celková velikost	Velikost indexu
100	1	1269	15,8 MB	10,4 KB
1000	1	2908	16,5 MB	48,7 KB
5000	1	10249	19,7 MB	250 KB
10000	1	20786	23,8 MB	364 KB
50000	1	103519	90,6 MB	2170 KB
100000	1	221278	173 MB	3480 KB

Tabulka 5.5: Části produkční databáze - měření s temporalitou, revize 1

Nahrání grafu, který již v databázi existuje

# db objektů	Revize	Čas[ms]	Celková velikost	Velikost indexu
100	2	770	15,8 MB	11,8 KB
1000	2	1642	17,1 MB	43,7 KB
5000	2	3154	22,9 MB	358 KB
10000	2	5229	29,8 MB	579 KB
50000	2	29834	124 MB	3420 MB
100000	2	88784	251 MB	6150 MB

Tabulka 5.6: Části produkční databáze - měření s temporalitou, revize 2

5.4.2.2 Bez temporality

Nahrání nového grafu

# db objektů	Pořadí	Čas[ms]	Celková velikost	Velikost indexu
100	1	979	15,7 MB	2,33 KB
1000	1	1834	16,3 MB	13,3 KB
5000	1	4370	18,9 MB	122 KB
10000	1	7711	22,1 MB	109 KB
50000	1	42320	65,2 MB	976 KB
100000	1	78295	130 MB	991 KB

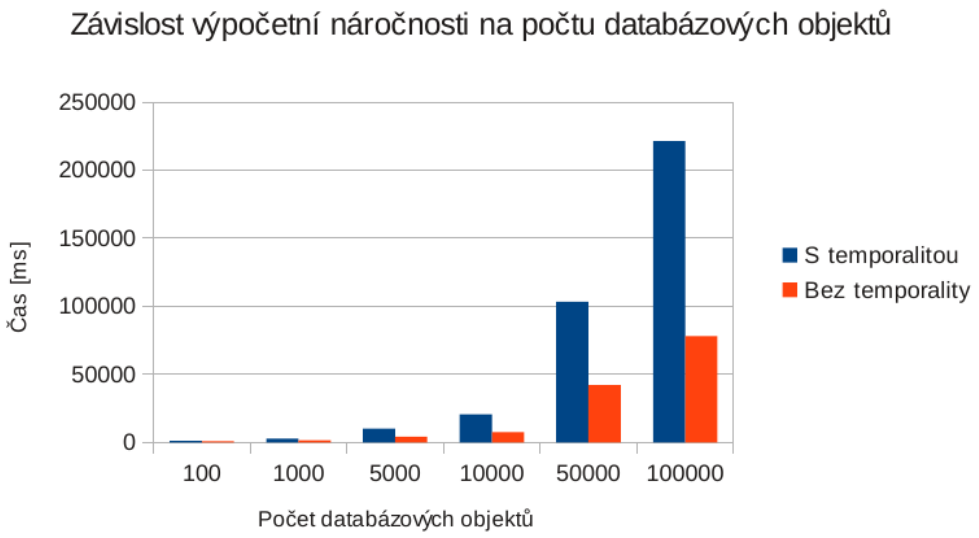
Tabulka 5.7: Části produkční databáze - měření bez temporality, nahrání 1

Nahrání grafu, který již v databázi existuje

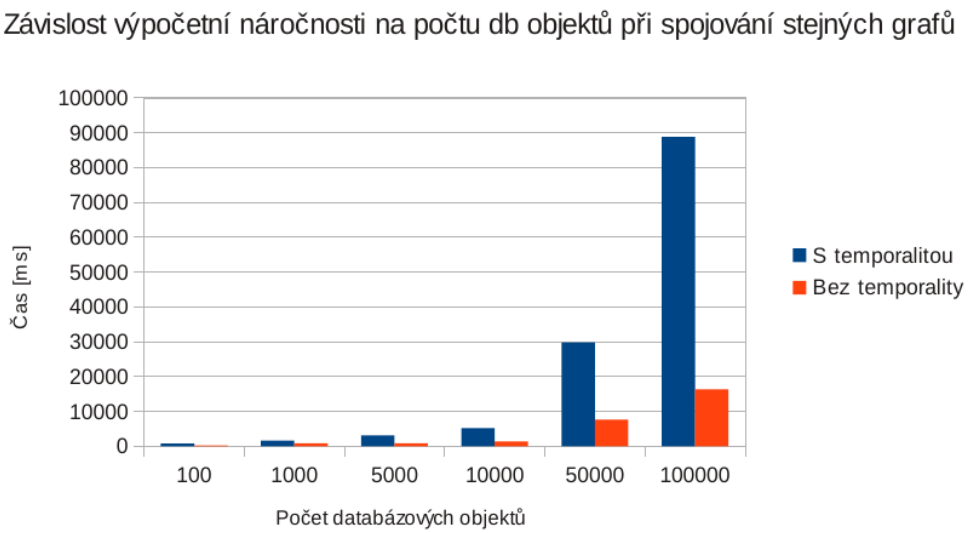
# db objektů	Pořadí	Čas[ms]	Celková velikost	Velikost indexu
100	2	257	15,7 MB	2,33 KB
1000	2	902	16,3 MB	13,3 KB
5000	2	852	18,9 MB	122 KB
10000	2	1358	22,1 MB	109 KB
50000	2	7617	65,2 MB	976 KB
100000	2	16379	131 MB	991 KB

Tabulka 5.8: Části produkční databáze - měření bez temporality, nahrání 2

Následující grafy vizualizují rozdíly v náročnostech čas mezi verzí bez zavedené temporality a verzí se zavedenou temporalitou. Porovnání z hlediska paměti a velikosti indexu je součástí přílohy v kapitole A.3.



Obrázek 5.1: Graf výpočetní náročnosti, revize 1



Obrázek 5.2: Graf výpočetní náročnosti, revize 2

Z vizualizace je patrné, že došlo k očekávánému paměťovému nárůstu a také vyššímu počtu výpočetních operací.

Paměťová náročnost při nahrávání nových databázových objektů stoupla zhruba o 35%, výpočetní náročnost se zvýšila zhruba o 100-200% a nárůst velikosti indexu se pohybuje také v těchto řádech.

Paměťová náročnost při nahrávání existujících databázových objektů stoupla zhruba o 100%, výpočetní náročnost se zvýšila až o 700% a velikosti indexu se zvýšila až o 600%.

Nárůst velikostí indexů je poměrně zanedbatelný, jelikož se nepohybuje v tak velkých řádech. Naopak nárůst výpočetní náročnosti existujících objektů se zvýšil poměrně dramaticky, proto by bylo vhodné analyzovat příčinu nárůstu a pokusit se o optimalizaci.

5.4.3 Produkční databáze

Benchmarking produkční databáze vedl poznatkům, že vytváření nové databáze s temporálními daty zvýšilo výpočetní náročnost zhruba o 100%, paměťová náročnost se zvýšila zhruba o 25% a velikost indexu se zvýšila zhruba o 400%.

5.5 Výběr index backendu

Časová náročnost se v případě operace merge zvýšila poměrně zásadně, proto byly rozebrány možnosti zrychlení těchto operací. Prvním krokem bylo

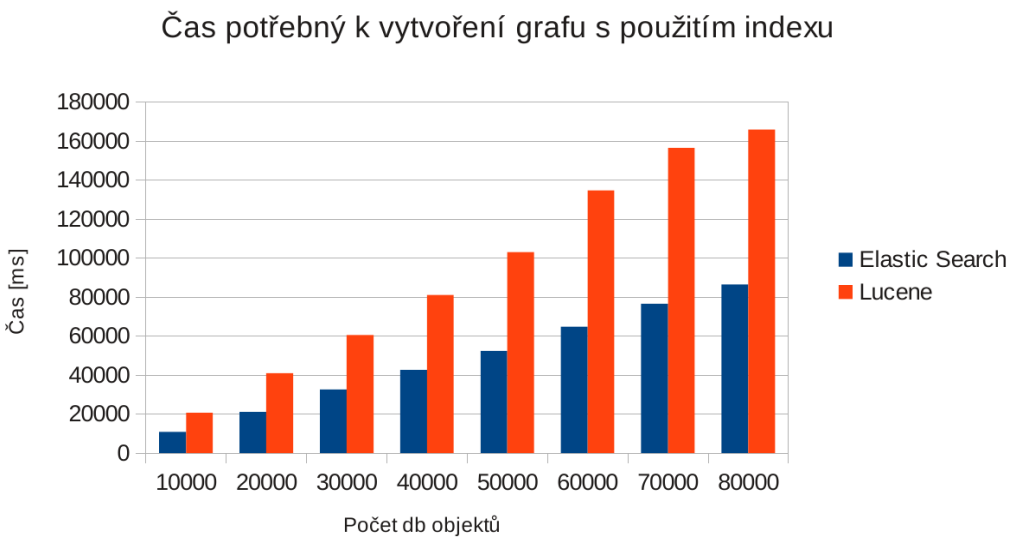
změření náročnosti při použití podporovaných indexových backendů. Těmi jsou v případě databáze Titan, nad kterou projekt Manta běží, backendy Lucene (dosud používaný) a Elastic Search.

Pro tyto testy byyl upraveny indexy na variantu rošířených indexů, blíže popsanou dle (5.0.2). Těmito úpravami bylo dosaženo stavu, že dotazy na transakční platnosti jsou pokryty pomocí vertex-centric indexů. Takto vytvořené indexy byly pak porovnány na částech produční databáze. Výsledky zachycuje následující tabulka.

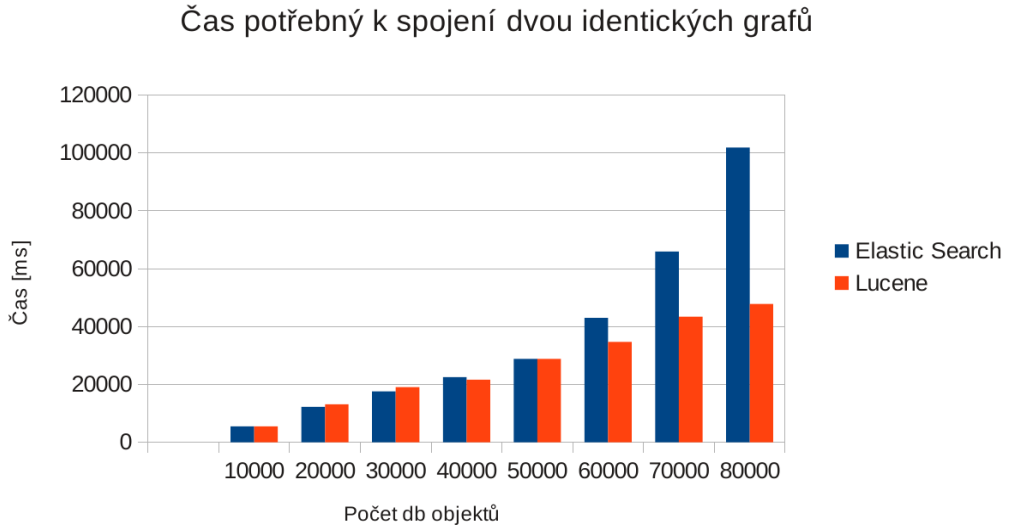
# db objektů	Revize	Čas[ms] - Elastic Search	Čas[ms] - Lucene
10000	1	10889	20546
20000	1	21032	40917
30000	1	32631	60517
40000	1	42546	81091
50000	1	52272	103038
60000	1	64766	134657
70000	1	76511	156555
80000	1	86428	165861
10000	2	5588	5521
20000	2	12258	13194
30000	2	17564	19123
40000	2	22465	21641
50000	2	28818	28850
60000	2	43042	34720
70000	2	65897	43417
80000	2	101903	47856

Tabulka 5.9: Porovnání index backendů

Data z předchozích měření jsou pro lepší porovnání zanesena do grafů.



Obrázek 5.3: Porovnání indexů - nový graf



Obrázek 5.4: Porovnání indexů - merge

Výsledky jsou však nepříznivé. V případě backendu Elastic Search je sice tvorba nového grafu téměř 2x rychlejší, ovšem v případě spojování stejných grafů dosahuje Elasic Search mnohem horších výsledků, jelikož čas evidentně

nestoupá lineárně, ale exponenciálně. Z těchto dvou možností vychází jako přijatelnější varianta backendu Lucene, jelikož exponenciální nárůst času u backendu Elasic Search je nepřijatelný. S tímto novým indexem bylo provedeno nové měření. Záměrně byly zvoleny násobky 10000 počtu databázových objektů, pro lepší možnost porovnání. Naměřené hodnoty zachycují následující tabulky.

5.6 Lucene, 2 indexy

Hodnoty jsou naměřeny s použitím Lucene backendu a dvou indexů - index přes všechny atributy TRAN_START a TRAN_END (5.0.1). Konkrétní hodnoty jsou zaneseny do tabulek, které jsou součástí přílohy, kapitoly A.5.

5.7 Bez temporality

Z důvodu možného porovnávání bylo provedeno také měření na verzi bez temporálních dat. Namřené hodnoty jsou součástí přílohy A.6. Sloupeček revize zde představuje, o kolikáté nahrání grafu se jedná.

5.8 Lucene, kompletní indexace

Následující tabulky obsahují data změřená při využití backendu Lucene s použitím vertex-centric indexů přes hrany typu hasResource, DIRECT, FILTER a úpravou stávajícího vertex-centric indexu dle (5.0.2). Konkrétní hodnoty jsou součástí přílohy, kapitoly A.7.

5.9 Elastic Search, kompletní indexace

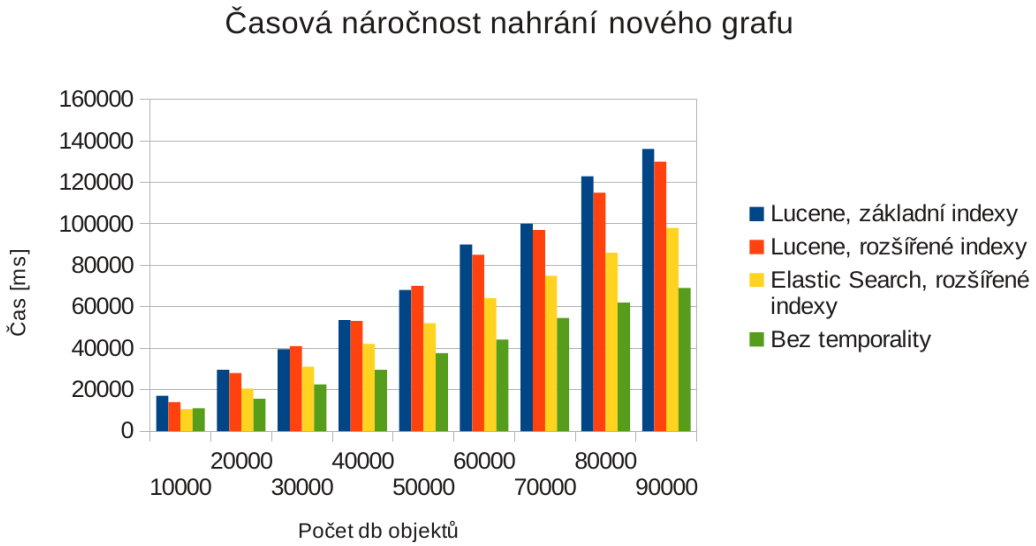
Následující tabulky obsahují data změřená při využití backendu Elastic Search s použitím rozšířených indexů dle (5.0.2). Konkrétní hodnoty jsou součástí přílohy, kapitoly A.8.

5.10 Závěr měření

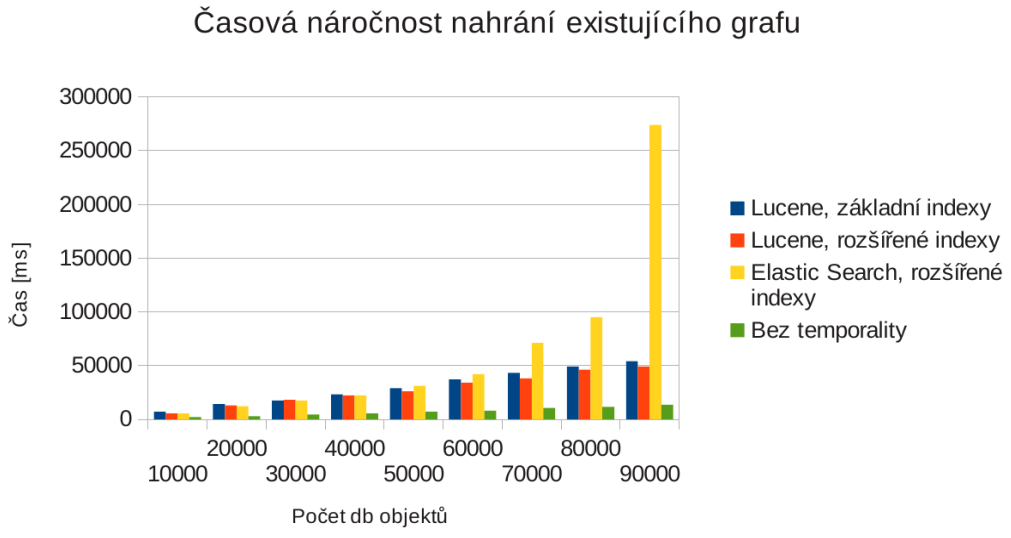
Byla provedena měření, v nichž byly na částech produkční databáze porovány možné varianty konfigurace indexů. Tyto varianty zahrnují

- Backend Lucene, základní konfigurace indexů dle kapitoly 5.0.1
- Backend Lucene, rozšířená konfigurace indexů dle kapitoly 5.0.2
- Backend Elastic Search, rozšířená konfigurace indexů dle kapitoly 5.0.2

Kompletní porovnání časové náročnosti je patrnější z následujících grafů. Grafy pro paměťovou náročnost a velikost indexu jsou obsahem přílohy, kapitoly A.9.



Obrázek 5.5: Zhodnocení variant indexových konfigurací - revize 1



Obrázek 5.6: Zhodnocení variant indexových konfigurací - revize 2

Z naměřenýcho hodnot vyplývá, že jako nejvhodnější varianta se jeví použití backendu Lucene s rozšířenými indexy (dle 5.0.2). Oproti backendu Elastic Search nabízí podstatný výkonnostní rozdíl patrný ve spojování grafů. Varianta s použitím základních dvou indexů je srovnatelná, ale jelikož je čas trvání operací klíčový (na rozdíl od velikosti paměti a indexu), jeví se tedy varianta s použitím rozšířených indexů jako vhodnější.

5.11 Úzké hrdlo operací

V rámci měření byla také orientačně změřena doba potřebná k vykonání určitých akcí. Těmito akcemi jsou klíčové dotazy na temporální data, vytváření dat nebo jejich úprava. Časy jsou uvedeny záměrně v určitém intervalu a se zaokrouhlením, v tomto případě nejsou důležité přesné hodnoty, ale jakých řádů v nanosekundách operace nabývá.

Důvodem realizace těchto měření je podstatné zvýšení časové náročnosti spojování grafů. Pokud by se jako časově nejnáročnější jevily dotazy na dohledání databázových objektů vzhledem k temporálním datům, přicházela by v úvahu další možná optimalizace. Měření zachycují následující tabulky [4]

5.11.1 První revize

Operace	# db objektů	Index	Čas[ns]
Vyhledání uzlu v tran. intervalu	10000	NE	5000 - 10000
	10000	ANO	3500 - 8000
	50000	NE	450 - 900
	50000	ANO	450 - 900
Vytváření nového uzlu	10000	NE	800000 - 1200000
	10000	ANO	800000 - 1100000
	50000	NE	700000 - 800000
	50000	ANO	700000 - 800000

Tabulka 5.10: Měření temporálních operací, revize 1

[4] Konfigurace indexů byla zvolena základní, dle kapitoly(5.0.1)

5.11.2 Druhá revize

Operace	# db objektů	Index	Čas[ns]
Vyhledání uzlu v tran. intervalu	10000	NE	400 - 900
	10000	ANO	400 - 900
	50000	NE	450 - 900
	50000	ANO	450 - 900
Dotaz na existenci uzlu v revizi	10000	NE	16000 - 20000
	10000	ANO	16000 - 20000
	50000	NE	12000 - 16000
	50000	ANO	12000 - 16000
Inkrementace tran. konce uzlu	10000	NE	48000 - 52000
	10000	ANO	38000 - 44000
	50000	NE	36000 - 44000
	50000	ANO	36000 - 44000

Tabulka 5.11: Měření temporálních operací, revize 2

Databáze Titan si evidentně ukládá pomocné struktury během výpočtů, jelikož se délka trvání operací postupně snižovala. Příkladem může být operace 'Vyhledání uzlu v tran. intervalu', která na 50000 zpracovaných databázových objektech trvá dále už jen desetinu času trvání na 10000 databázových objektech. Tyto pomocné struktury však neovlivnily původní záměr - tedy zhodnocení, kde je úzké hrdlo těchto operací. Nově vytvořené dotazy na dohledání dat se zohledněním temporality tedy trvají jen zlomek času ostatních operací. V případě nových databázových objektů tedy zabírá nejvíce času vytváření konkrétního uzlu. V případě spojení uzlů úprava transakčního konce uzlu (zahrnující dohledání určitých hran, na nichž bude inkrementován transakční konec). Pravděpodobně je v tomto případě časově náročné upravování indexů při každé operaci.

KAPITOLA 6

Závěr

Cílem této práce bylo navrhnout, naimplementovat a otestovat zavedení temporální složky do projektu Manta, který se zabývá analýzou datových toků v organizaci. V rámci této práce bylo také nutné se seznámit s grafovou databází Titan, zjistit, jak jsou v současné době reprezentována temporální data v relačních databázích, zda existují řešení temporality v grafových databázích a v neposlední řadě také seznámení se samotným projektem Manta.

V analýze byly rozebrány jednotlivé možnosti reprezentace temporálních dat a to s důrazem na efektivitu řešní z hlediska časové a paměťové náročnosti. Z těchto řešení bylo vybráno to, které bylo považováno za nejvhodnější a následně bylo naimplementováno. Samotná implementace a celková funkčnost byla následně otestována a změřena pomocí výkonnostních testů. Na základě výsledků těchto testů bylo zvolena nejvhodnější varianta použitých indexů.

6.1 Možná návaznost

V rámci analýzy byly stanoveny omezující předpoklady z důvodu těžko řešitelných výkonnostních a také návrhových problémů. Z důvodu těchto problémů byla realizována pouze první časová dimenze (viz 2.1.7.1, 2.2). Návazností na tuto práci by mohla být analýza těchto problémů a hlubší zamyšlení nad možnými řešeními.

Dalším pokračováním by mohl být celkový refactoring dotazů a hlubší zamyšlení se nad optimalizací dotazů a použitých struktur a provedení výkonnostních testů na výkonnějších strojích s použitím většího množství dat. Součástí refactoringu by mohlo být také vytvoření určité formy pro jednotlivé databázové objekty - v současné chvíli není definováno a ošetřeno, jaké mají různé typy databázových objektů své jednoznačné identifikátory.

Literatura

[1] Apache Lucene - Query Parser Syntax. 2013. Dostupné z WWW: <http://lucene.apache.org/core/2_9_4/queryparsersyntax.html>

[2] Indexing Backend Overview. 2014. Dostupné z WWW: <https://github.com/thinkaurelius/titan/wiki/Indexing-Backend-Overview>

[3] Finger, M.: A Logical Reconstruction of Temporal Databases. únor 2015. Dostupné z WWW: <http://www.mi.sanu.ac.rs/~uros.m/logcom/hdb/Volume_10/Issue_06/pdf/100847.pdf>

[4] Halawani, S. M.; Al-Romema, N. A.: Memory Storage Issues of Temporal Database Applications on Relational Database Management Systems. *Journal of Computer Science*, 2010, ISSN 1549-3636.

[5] Halawani, S. M.; AlBidewi, I.; Ahmad, A. R.; aj.: Retrieval Optimization Technique for Tuple Timestamp Historical Relation Temporal Data Model. *Journal of Computer Science*, 2012, ISSN 1549-3636.

[6] Labs, T.: *Temporal Table Support*. 2012. Dostupné z WWW: <http://cs.ulb.ac.be/public/_media/teaching/infoh415/teradata_temporal_support.pdf>

[7] Montanari, A.; Chomicki, J.: Time Domain. 2014. Dostupné z WWW: <http://www.cse.buffalo.edu/~chomicki/EDS_TD.pdf>

[8] Patel, J.: Temporal Database System. únor 2015. Dostupné z WWW: <http://www.doc.ic.ac.uk/~pjm/teaching/student_projects/jaymin_patel.pdf>

[9] Salazar, J. S.: Autogeneration of Code for Creating Web Application That Handles Temporal Data with Struts2. 2014.

[10] Valenta, M.; Hermann, L.: Datový sklad - 2. zpráva projektu. Technická zpráva, Department of Software Engineering, Czech Technical University in Prague, únor 2015.

[11] Valenta, M.; Hermann, L.: Návrh datového úložiště projektu Nástroje pro automatizaci Quality Assurance rozsáhlých Business Intelligence systémů a datových skladů. Technická zpráva, Department of Software Engineering, Czech Technical University in Prague, únor 2015.

PŘÍLOHA A

Naměřené hodnoty

A.1 Test prvního načtení grafu

A.1.1 100 databázových objektů

A.1.1.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
100	1	1632	1 sec. 632 ms	15,7 MB	2,33 KB
100	1	607	607 ms	15,7 MB	2,33 KB
100	1	698	698 ms	15,7 MB	2,33 KB

A.1.1.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
100	1	1424	1 sec. 424 ms	15,8 MB	10,4 KB
100	1	1037	1 sec. 37 ms	15,7 MB	10,4 KB
100	1	1345	1 sec. 345	15,7 MB	10,4 KB

A.1.2 1000 databázových objektů

A.1.2.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
1000	1	2849	2 sec. 849 ms	16,3 MB	13,3 KB
1000	1	1445	1 sec. 445 ms	16,3 MB	13,3 KB
1000	1	1208	1 sec. 208 ms	16,3 MB	13,3 KB

A.1.2.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
1000	1	4087	4 sec. 87 ms	16,5 MB	48,7 KB
1000	1	2353	2 sec. 353 ms	16,5 MB	48,7 KB
1000	1	2283	2 sec. 283 ms	16,4 MB	48,7 KB

A.1.3 5000 databázových objektů

A.1.3.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
5000	1	5253	5 sec. 253 ms	18,9 MB	122 KB
5000	1	4370	4 sec. 370 ms	18,9 MB	122 KB
5000	1	3486	3 sec. 486 ms	18,9 MB	122 KB

A.1.3.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
5000	1	11149	11 sec. 149 ms	19,7 MB	250 KB
5000	1	10387	10 sec. 387 ms	19,7 MB	250 KB
5000	1	9211	9 sec. 211 ms	19,7 MB	250 KB

A.1.4 10000 databázových objektů

A.1.4.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
10000	1	8053	8 sec. 53 ms	22,1 MB	109 KB
10000	1	7891	7 sec. 891 ms	22,1 MB	109 KB
10000	1	7189	7 sec. 189 ms	22,1 MB	109 KB

A.1.4.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
10000	1	20285	20 sec. 285 ms	23,8 MB	364 KB
10000	1	21728	21 sec. 728 ms	23,8 MB	364 KB
10000	1	20345	20 sec. 345 ms	23,8 MB	364 KB

A.1.5 50000 databázových objektů

A.1.5.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
50000	1	42886	42 sec. 886 ms	65,2 MB	976 KB
50000	1	42584	42 sec. 584 ms	65,2 MB	976 KB
50000	1	41490	41 sec. 490 ms	65,2 MB	976 KB

A.1.5.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
50000	1	101399	1 min. 41 sec.	90,6 MB	2,17 MB
50000	1	101326	1 min. 41 sec.	90,6 MB	2,17 MB
50000	1	107832	1 min. 48 sec.	90,7 MB	2,17 MB

A.1.6 100000 databázových objektů

A.1.6.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
100000	1	76378	1 min. 16 sec.	130 MB	991 KB
100000	1	79946	1 min. 20 sec.	130 MB	991 KB
100000	1	78562	1 min. 18 sec.	130 MB	991 KB

A.1.6.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
100000	1	209391	3 min. 29 sec.	170 MB	3,48 MB
100000	1	220378	3 min. 40 sec.	171 MB	3,48 MB
100000	1	234066	3 min. 54 sec.	173 MB	3,48 MB

A.2 Test druhého načtení grafu

A.2.1 100 databázových objektů

A.2.1.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
100	2	357	357 ms	15,7 MB	2,33 KB
100	2	202	202 ms	15,7 MB	2,33 KB
100	2	212	212 ms	15,7 MB	2,33 KB

A.2.1.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
100	2	1072	1 sec. 72	15,9 MB	11,8 KB
100	2	664	664 ms	15,8 MB	11,8 KB
100	2	575	575 ms	15,8 MB	11,8 KB

A.2.2 1000 databázových objektů

A.2.2.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
1000	2	1356	1 sec. 356 ms	16,3 MB	13,3 KB
1000	2	939	939 ms	16,3 MB	13,3 KB
1000	2	410	410 ms	16,3 MB	13,3 KB

A.2.2.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
1000	2	2404	2 sec. 404 ms	17,2 MB	43,7 KB
1000	2	1506	1 sec. 506 ms	17 MB	43,7 KB
1000	2	1015	1 sec. 15 ms	17 MB	43,7 KB

A.2.3 5000 databázových objektů

A.2.3.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
5000	2	985	985 ms	18,9 MB	122 KB
5000	2	876	876 ms	18,9 MB	122 KB
5000	2	695	695 ms	18,9 MB	122 KB

A.2.3.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
5000	2	3484	3 sec. 484 ms	22,7 MB	358 KB
5000	2	3094	3 sec. 94 ms	22,7 MB	358 KB
5000	2	2884	2 sec. 884 ms	23 MB	358 KB

A.2.4 10000 databázových objektů

A.2.4.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
10000	2	1324	1 sec. 324 ms	22,1 MB	109 KB
10000	2	1387	1 sec. 387 ms	22,1 MB	109 KB
10000	2	1364	1 sec. 364 ms	22,1 MB	109 KB

A.2.4.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
10000	2	5361	5 sec. 361 ms	29,8 MB	579 KB
10000	2	5232	5 sec. 232 ms	29,8 MB	579 KB
10000	2	5095	5 sec. 95 ms	29,8 MB	579 KB

A.2.5 50000 databázových objektů

A.2.5.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
50000	2	7456	7 sec. 456 ms	65,2 MB	976 KB
50000	2	7120	7 sec. 120 ms	65,2 MB	976 KB
50000	2	8274	8 sec. 274 ms	65,2 MB	976 KB

A.2.5.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
50000	2	29455	29 sec. 455 ms	123 MB	3,42 MB
50000	2	30728	30 sec. 728 ms	128 MB	3,42 MB
50000	2	29320	29 sec. 320 ms	121 MB	3,42 MB

A.2.6 100000 databázových objektů

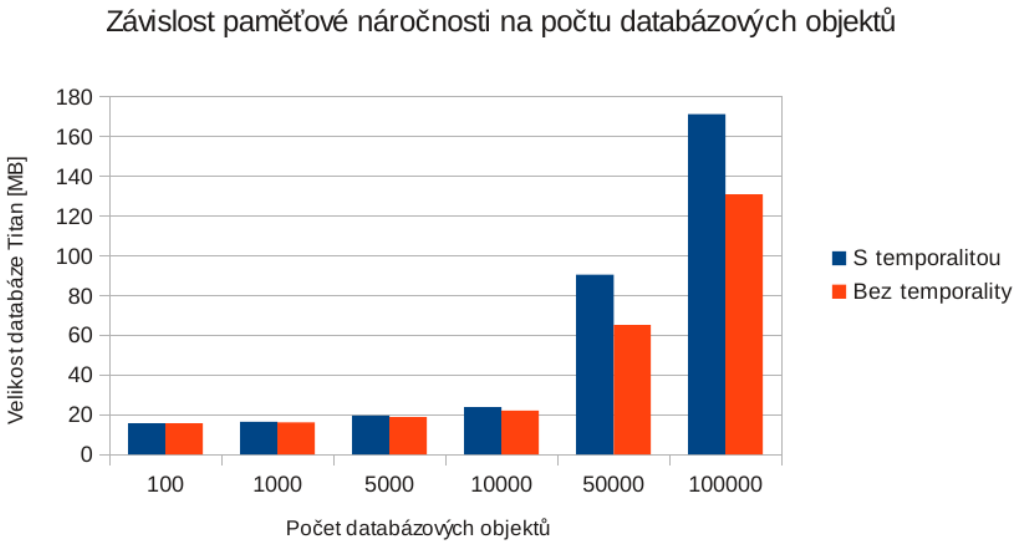
A.2.6.1 Bez temporality

Import	Pořadí	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
100000	2	16347	16 sec. 347 ms.	131 MB	991 KB
100000	2	16635	16 sec. 635 ms	131 MB	991 KB
100000	2	16154	16 sec. 154 ms	131 MB	991 KB

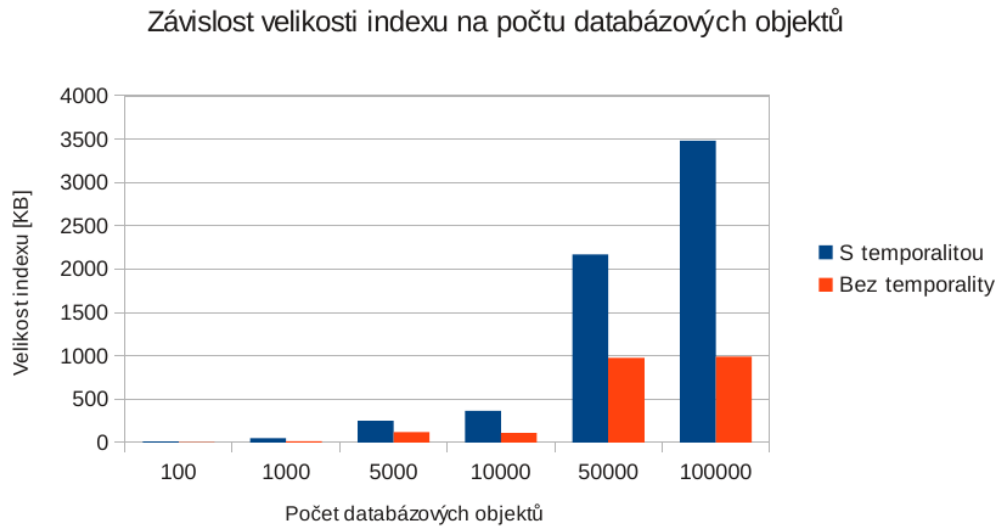
A.2.6.2 S temporalitou

Import	Revize	Čas[ms]	Čas[HR]	Celková velikost	Velikost indexu
100000	2	65571	1 min. 6 sec.	246 MB	6,15 MB
100000	2	98841	1 min. 38 sec.	254 MB	6,15 MB
100000	2	101942	1 min. 42 sec.	254 MB	6,15 MB

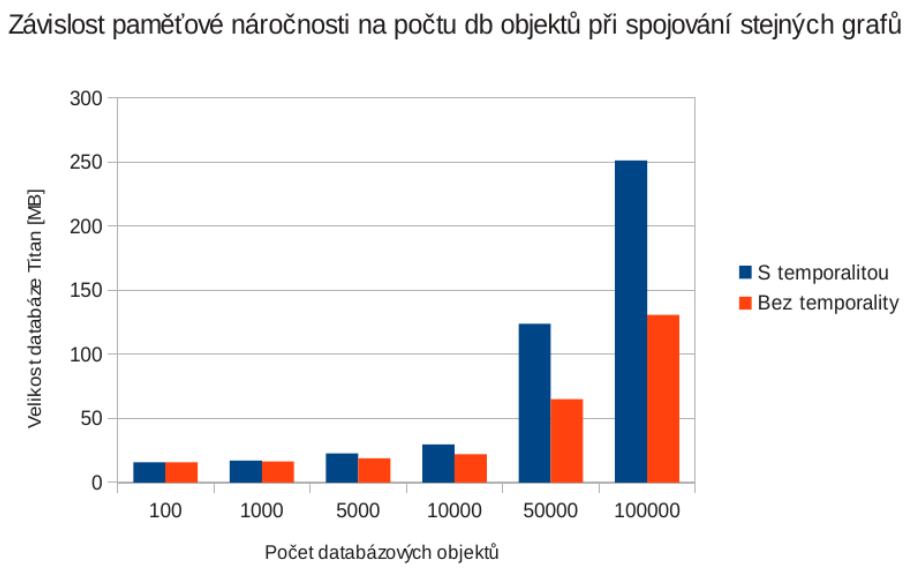
A.3 Porovnání velikosti databáze a indexu ve verzích s temporalitou a bez temporality



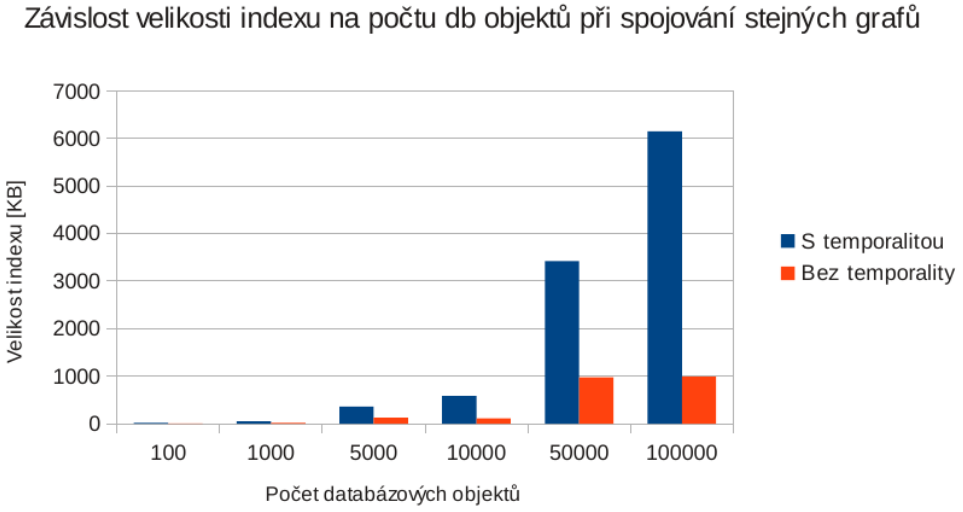
Obrázek A.1: Paměťová náročnost, revize 1



Obrázek A.2: Paměťová náročnost, revize 2



Obrázek A.3: Velikost indexu, revize 1



Obrázek A.4: Velikost indexu, revize 2

A.4 Vytvoření testovacích souborů z produkční databáze

Pro vytvoření souborů byly použit následující Windows PowerShell příkazy:

```
gc export_O2.csv | select -first 100 | Out-File 100.csv -encoding utf8
gc export_O2.csv | select -first 1000 | Out-File 1000.csv -encoding
    utf8
gc export_O2.csv | select -first 5000| Out-File 5000.csv -encoding
    utf8
gc export_O2.csv | select -first 10000 | Out-File 10000.csv -encoding
    utf8
gc export_O2.csv | select -first 50000 | Out-File 50000.csv -encoding
    utf8
gc export_O2.csv | select -first 100000 | Out-File 100000.csv
    -encoding utf8
```

Ovšem výstupní soubory obsahovaly na počátku souboru neviditelné znaky, které způsobovaly, že první záznam byl vyhodnocen jako neexistující typ. Proto byl použit příkaz head v Bourne-again shellu (OS Linux).

```
head -100 export_O2.csv > linux/100.csv
head -1000 export_O2.csv > linux/1000.csv
head -5000 export_O2.csv > linux/5000.csv
head -10000 export_O2.csv > linux/10000.csv
head -50000 export_O2.csv > linux/50000.csv
head -100000 export_O2.csv > linux/100000.csv
head -250000 export_O2.csv > linux/250000.csv
```

Tyto soubory byly vygenerovány nativně v kódování utf-8 a žádné nežádoucí znaky neobsahovaly, proto byly použity pro měření.

A.5 Měření v případě použití backendu Lucene a 2 základních indexů

A.5.1 Nahrání nového grafu

A.5.1.1 10000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
10000	1	22448	23.8 MB	365 KB
10000	1	14851	23.8 MB	365 KB
10000	1	13078	23.8 MB	365 KB

A.5.1.2 20000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
20000	1	39343	32,2 MB	728 KB
20000	1	27814	31,9 MB	728 KB
20000	1	26250	31,9 MB	728 KB

A.5.1.3 30000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
30000	1	40345	55,7 MB	1005 KB
30000	1	39413	55,7 MB	1005 KB
30000	1	38631	55,7 MB	1005 KB

A.5.1.4 40000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
40000	1	53333	64.2 MB	1390 KB
40000	1	54210	64.2 MB	1390 KB
40000	1	53054	64.2 MB	1390 KB

A.5.1.5 50000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
50000	1	68336	89.1 MB	2170 KB
50000	1	67569	89.1 MB	2170 KB
50000	1	66571	89.1 MB	2170 KB

A.5.1.6 60000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
60000	1	89041	97,8 MB	2050 KB
60000	1	96973	97,8 MB	2050 KB
60000	1	84226	97,8 MB	2050 KB

A.5.1.7 70000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
70000	1	104231	122 MB	2430 KB
70000	1	98583	122 MB	2430 KB
70000	1	102437	122 MB	2430 KB

A.5.1.8 80000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
80000	1	126151	133 MB	2800 KB
80000	1	126435	133 MB	2800 KB
80000	1	119984	133 MB	2800 KB

A.5.1.9 90000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
90000	1	132852	144 MB	5400 KB
90000	1	141201	144 MB	5400 KB
90000	1	133089	144 MB	5400 KB

A.5.2 Nahrání grafu, který již v databázi existuje

A.5.2.1 10000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
10000	2	9028	30.5 MB	579 KB
10000	2	6118	30.5 MB	579 KB
10000	2	5129	30.5 MB	579 KB

A.5.2.2 20000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
20000	2	17589	45 MB	1140 KB
20000	2	13122	47 MB	1140 KB
20000	2	12498	46 MB	1140 KB

A.5.2.3 30000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
30000	2	17750	74 MB	1710 KB
30000	2	17971	83 MB	1710 KB
30000	2	17716	76 MB	1710 KB

A.5.2.4 40000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
40000	2	26069	92.1 MB	2370 KB
40000	2	23258	92.1 MB	2370 KB
40000	2	21292	92.1 MB	2370 KB

A.5.2.5 50000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
50000	2	30805	89.1 MB	2170 KB
50000	2	28520	89.1 MB	2170 KB
50000	2	29980	89.1 MB	2170 KB

A.5.2.6 60000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
60000	2	36069	143 MB	3530 KB
60000	2	37963	143 MB	3530 KB
60000	2	38496	143 MB	3530 KB

A.5.2.7 70000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
70000	2	41724	178 MB	6090 KB
70000	2	46012	178 MB	6090 KB
70000	2	41741	178 MB	6090 KB

A.5.2.8 80000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
80000	2	51500	202 MB	4430 KB
80000	2	49448	202 MB	4430 KB
80000	2	47793	202 MB	4430 KB

A.5.2.9 90000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
90000	2	55099	223 MB	5550 KB
90000	2	56912	223 MB	5550 KB
90000	2	52554	223 MB	5550 KB

A.6 Měření původní verze bez temporality

A.6.1 Nahrání nového grafu

A.6.1.1 10000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
10000	1	8792	22.1 MB	109 KB
10000	1	14380	22.1 MB	109 KB
10000	1	9262	22.1 MB	109 KB

A.6.1.2 20000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
20000	1	15109	28.7 MB	217 KB
20000	1	15846	28.7 MB	217 KB
20000	1	16108	28.7 MB	217 KB

A.6.1.3 30000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
30000	1	22243	50.9 MB	307 KB
30000	1	22640	50.9 MB	307 KB
30000	1	23614	50.9 MB	307 KB

A.6.1.4 40000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
40000	1	29259	57.5 MB	397 KB
40000	1	29969	57.5 MB	397 KB
40000	1	29575	57.5 MB	397 KB

A.6.1.5 50000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
50000	1	36296	64.9 MB	976 KB
50000	1	37059	64.9 MB	976 KB
50000	1	38495	64.9 MB	976 KB

A.6.1.6 60000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
60000	1	44615	86.9 MB	590 KB
60000	1	44668	86.9 MB	590 KB
60000	1	44150	86.9 MB	590 KB

A.6.1.7 70000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
70000	1	53838	94.1 MB	704 KB
70000	1	54728	94.1 MB	704 KB
70000	1	56710	94.1 MB	704 KB

A.6.1.8 80000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
80000	1	61451	101 MB	803 KB
80000	1	62437	101 MB	803 KB
80000	1	61664	101 MB	803 KB

A.6.1.9 90000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
90000	1	68081	124 MB	898 KB
90000	1	69187	124 MB	898 KB
90000	1	65487	124 MB	898 KB

A.6.2 Nahrání grafu, který již v databázi existuje

A.6.2.1 10000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
10000	2	2356	22.1 MB	109 KB
10000	2	1778	22.1 MB	109 KB
10000	2	1879	22.1 MB	109 KB

A.6.2.2 20000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
20000	2	2979	28.7 MB	217 KB
20000	2	2796	28.7 MB	217 KB
20000	2	2543	28.7 MB	217 KB

A.6.2.3 30000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
30000	2	4197	50.9 MB	307 KB
30000	2	4399	50.9 MB	307 KB
30000	2	4277	50.9 MB	307 KB

A.6.2.4 40000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
40000	2	5585	57.5 MB	397 KB
40000	2	5242	57.5 MB	397 KB
40000	2	5373	57.5 MB	397 KB

A.6.2.5 50000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
50000	2	7462	64.9 MB	976 KB
50000	2	6801	64.9 MB	976 KB
50000	2	6801	64.9 MB	976 KB

A.6.2.6 60000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
60000	2	8174	86.9 MB	590 KB
60000	2	7945	86.9 MB	590 KB
60000	2	8001	86.9 MB	590 KB

A.6.2.7 70000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
70000	2	10809	94.1 MB	704 KB
70000	2	11842	94.1 MB	704 KB
70000	2	10288	94.1 MB	704 KB

A.6.2.8 80000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
80000	2	10764	101 MB	803 KB
80000	2	11241	101 MB	803 KB
80000	2	12199	101 MB	803 KB

A.6.2.9 90000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
90000	2	13241	124 MB	898 KB
90000	2	13963	124 MB	898 KB
90000	2	14001	124 MB	898 KB

A.7 Měření v případě použití backendu Lucene a rozšířených indexů

A.7.1 Nahrání nového grafu

A.7.1.1 10000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
10000	1	13998	24 MB	365 KB
10000	1	14822	24 MB	365 KB
10000	1	14287	24 MB	365 KB

A.7.1.2 20000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
20000	1	28439	32.1 MB	729 KB
20000	1	29208	32.1 MB	729 KB
20000	1	27999	32.1 MB	729 KB

A.7.1.3 30000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
30000	1	41612	56 MB	1050 KB
30000	1	41279	56 MB	1050 KB
30000	1	40826	56 MB	1050 KB

A.7.1.4 40000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
40000	1	53333	64 MB	1390 KB
40000	1	55380	64 MB	1390 KB
40000	1	54121	64 MB	1390 KB

A.7.1.5 50000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
50000	1	71630	90 MB	2200 KB
50000	1	69715	90 MB	2200 KB
50000	1	68721	90 MB	2200 KB

A.7.1.6 60000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
60000	1	85087	98 MB	2050 KB
60000	1	84946	98 MB	2050 KB
60000	1	83980	98 MB	2050 KB

A.7.1.7 70000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
70000	1	97730	124 MB	2430 KB
70000	1	96919	122 MB	2430 KB
70000	1	98001	122 MB	2430 KB

A.7.1.8 80000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
80000	1	113781	133 MB	2800 KB
80000	1	114930	133 MB	2800 KB
80000	1	116435	133 MB	2800 KB

A.7.1.9 90000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
90000	1	129278	144 MB	5400 KB
90000	1	132287	144 MB	5400 KB
90000	1	130054	144 MB	5400 KB

A.7.2 Nahrání grafu, který již v databázi existuje

A.7.2.1 10000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
10000	2	5337	30.7 MB	579 KB
10000	2	5588	30.5 MB	579 KB
10000	2	4759	30.7 MB	579 KB

A.7.2.2 20000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
20000	2	13260	45 MB	1140 KB
20000	2	13312	45 MB	1140 KB
20000	2	12195	45 MB	1140 KB

A.7.2.3 30000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
30000	2	18111	77.6 MB	1710 KB
30000	2	18274	77.6 MB	1710 KB
30000	2	18240	77.6 MB	1710 KB

A.7.2.4 40000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
40000	2	21374	93 MB	2370 KB
40000	2	22364	93 MB	2370 KB
40000	2	22484	93 MB	2370 KB

A.7.2.5 50000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
50000	2	27406	120 MB	3420 KB
50000	2	26340	120 MB	3420 KB
50000	2	26777	120 MB	3420 KB

A.7.2.6 60000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
60000	2	34836	137 MB	3530 KB
60000	2	34687	137 MB	3530 KB
60000	2	34571	137 MB	3530 KB

A.7.2.7 70000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
70000	2	38193	178 MB	6090 KB
70000	2	37977	178 MB	6090 KB
70000	2	37841	178 MB	6090 KB

A.7.2.8 80000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
80000	2	48123	193 MB	4500 KB
80000	2	44589	193 MB	4500 KB
80000	2	46872	193 MB	4500 KB

A.7.2.9 90000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
90000	2	46941	224 MB	5550 KB
90000	2	47812	224 MB	5550 KB
90000	2	49468	224 MB	5550 KB

A.8 Měření v případě použití backendu Elastic Search a rozšířených indexů

A.8.1 Nahrání nového grafu

A.8.1.1 10000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
10000	1	11022	25.6 MB	1730 KB
10000	1	10889	25.6 MB	1730 KB
10000	1	10244	25.6 MB	2340 KB

A.8.1.2 20000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
20000	1	20524	32.6 MB	1153 KB
20000	1	21032	33.1 MB	1187 KB
20000	1	22134	33.1 MB	1153 KB

A.8.1.3 30000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
30000	1	33724	59.3 MB	4700 KB
30000	1	31841	59.3 MB	4600 KB
30000	1	32631	59.3 MB	4700 KB

A.8.1.4 40000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
40000	1	41083	65,4 MB	3000 KB
40000	1	42314	65,4 MB	2900 KB
40000	1	43210	64.2 MB	3000 KB

A.8.1.5 50000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
50000	1	50763	94 MB	7570 KB
50000	1	52721	94 MB	7570 KB
50000	1	52103	94 MB	7570 KB

A.8.1.6 60000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
60000	1	64766	101 MB	6200 KB
60000	1	65933	101 MB	6200 KB
60000	1	63898	101 MB	6200 KB

A.8.1.7 70000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
70000	1	78566	129 MB	10600 KB
70000	1	76511	129 MB	10600 KB
70000	1	76474	129 MB	10600 KB

A.8.1.8 80000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
80000	1	86428	135 MB	8440 KB
80000	1	87915	134 MB	7300 KB
80000	1	85765	134 MB	7300 KB

A.8.1.9 90000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
90000	1	97628	152 MB	14800 KB
90000	1	98778	152 MB	14800 KB
90000	1	99874	152 MB	14800 KB

A.8.2 Nahrání grafu, který již v databázi existuje

A.8.2.1 10000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
10000	2	5857	30.3 MB	1170 KB
10000	2	5588	30.5 MB	1170 KB
10000	2	4759	31.2 MB	1160 KB

A.8.2.2 20000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
20000	2	12258	47 MB	2200 KB
20000	2	13122	47 MB	2200 KB
20000	2	11443	48 MB	2300 KB

A.8.2.3 30000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
30000	2	17564	77.6 MB	3900 KB
30000	2	18049	77.6 MB	3900 KB
30000	2	17610	77.6 MB	3900 KB

A.8.2.4 40000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
40000	2	22929	96.3 MB	4590 KB
40000	2	21978	96.3 MB	4590 KB
40000	2	23004	96.3 MB	4590 KB

A.8.2.5 50000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
50000	2	29871	126 MB	5040 KB
50000	2	33145	126 MB	5040 KB
50000	2	31256	126 MB	5040 KB

A.8.2.6 60000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
60000	2	43042	145 MB	5370 KB
60000	2	41623	144 MB	5370 KB
60000	2	42351	145 MB	5370 KB

A.8.2.7 70000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
70000	2	65897	174 MB	5000 KB
70000	2	76143	174 MB	5100 KB
70000	2	70212	174 MB	5100 KB

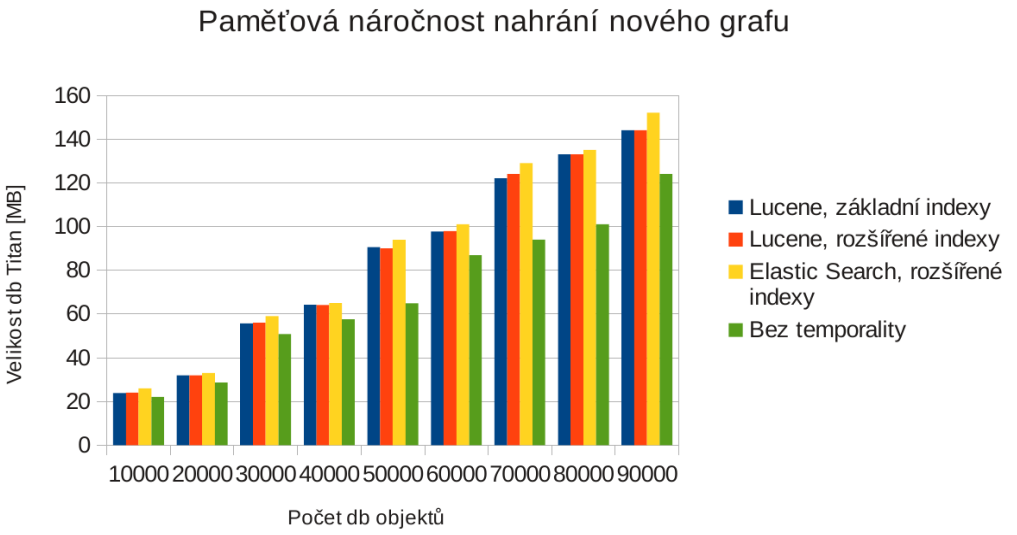
A.8.2.8 80000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
80000	2	95184	193 MB	10400 KB
80000	2	93076	194 MB	6460 KB
80000	2	97450	194 MB	6460 KB

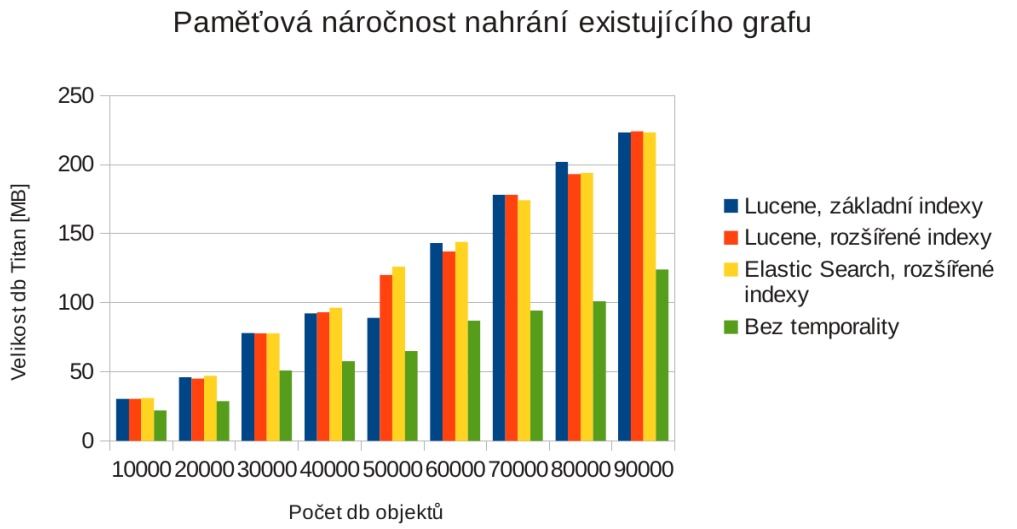
A.8.2.9 90000 db objektů

Import	Revize	Čas[ms]	Celková velikost	Velikost indexu
90000	2	162161	223 MB	5550 KB
90000	2	237439	223 MB	5550 KB
90000	2	423360	223 MB	5550 KB

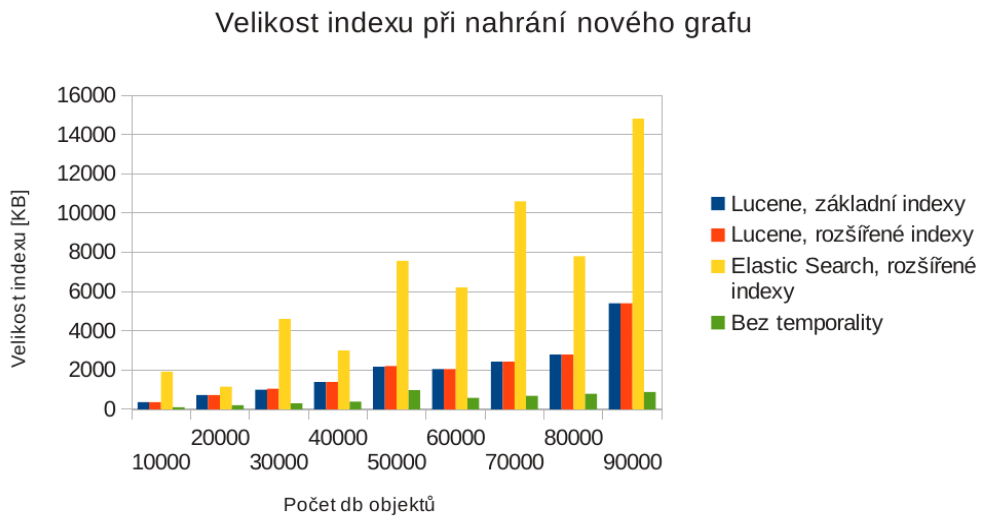
A.9 Celkové zhodnocení z hlediska velikosti databáze a indexu



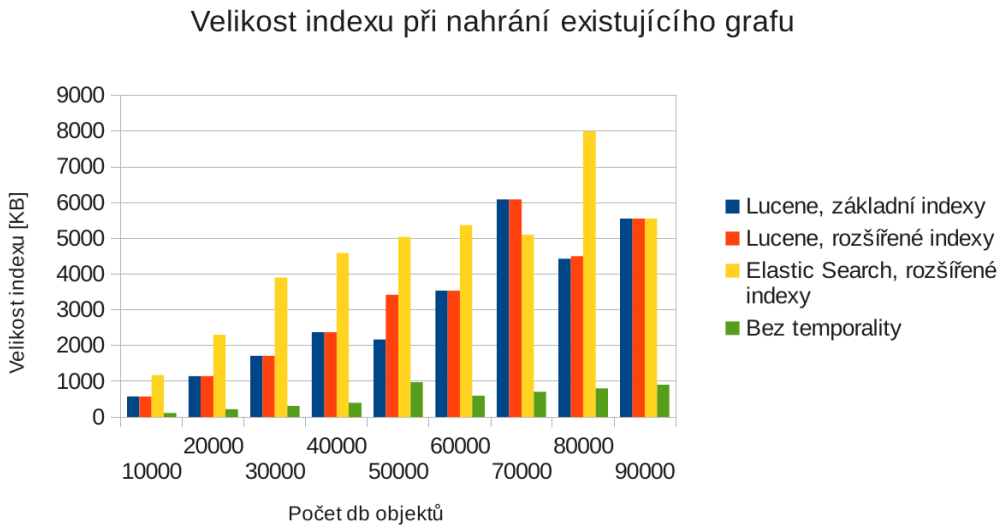
Obrázek A.5: Zhodnocení paměťové náročnosti indexových variant - revize 1



Obrázek A.6: Zhodnocení paměťové náročnosti indexových variant - revize 2



Obrázek A.7: Zhodnocení velikosti indexu jednotlivých variant - revize 1



Obrázek A.8: Zhodnocení velikosti indexu jednotlivých variant - revize 2

PŘÍLOHA B

Použité zkratky

RDBMS Relational database management system - systém řízení báze dat

GPL (GNU GPL) - GNU General Public License - "všeobecná veřejná licence GNU"- licence pro svobodný software

DB Database - databáze

PL/SQL Procedural Language/Structured Query Language - procedurální rozšíření jazyka SQL od firmy Oracle

REST - Representational state transfer - architektura rozhraní, navržená pro distribuované prostředí, kterou navrhl v roce 2000 Roy Fielding v rámci disertační práce Architectural Styles and the Design of Network-based Software Architectures

TCP/IP - Transmission Control Protocol / Internet Protocol - rodina protokolů pro komunikaci v počítačové síti

SQL - Structured Query Language - standardizovaný strukturovaný jazyk pro dotazování nad relačními databázemi

TSQL - Transact-SQL - proprietární rozšíření jazyka SQL o procedurální programování, proměnné, a další podporu zpracování dat

TX - Database transaction - skupina příkazů, které převedou databázi z jednoho konzistentního stavu do druhého

OS - Operating system - operační systém

CSV - Comma-separated values - jednoduchý souborový formát určený pro textovou reprezentaci tabulkových dat

XML - Extensible Markup Language - obecný značkovací jazyk

HR - Human readable - formát zápisu času

PŘÍLOHA C

Obsah přiloženého CD

- `gremlin` ..... adresář s Gremlin skriptem
  - `CreateDbGremlin.txt` ..... skript pro vytvoření testovací databáze
- `graphviz` ..... adresář se zdrojovými kódy grafů v gv formátu
  - `output` ..... adresář s GraphViz grafy převedenými do pdf formátu
- `testy_web`
  - `test1` ..... adresář pro jednotlivé revize prvního testu
    - `krok1` ..... adresář s testovacími soubory
    - `krok2` ..... adresář s testovacími soubory
    - `krok3` ..... adresář s testovacími soubory
  - `test2` ..... adresář pro jednotlivé revize druhého testu
    - `krok1` ..... adresář s testovacími soubory
    - `krok2` ..... adresář s testovacími soubory
    - `krok3` ..... adresář s testovacími soubory
  - `test3` ..... adresář pro jednotlivé revize třetího testu
    - `krok1` ..... adresář s testovacími soubory
    - `krok2` ..... adresář s testovacími soubory
    - `krok3` ..... adresář s testovacími soubory
  - `test4` ..... adresář pro jednotlivé revize čtvrtého testu
    - `krok1` ..... adresář s testovacími soubory
    - `krok2` ..... adresář s testovacími soubory
    - `krok3` ..... adresář s testovacími soubory
- `src` ..... zdrojové kódy práce
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `figures` ..... obrázky použité v této práci
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF