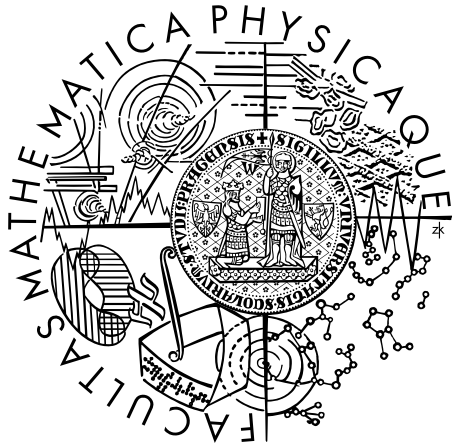
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Roman Krejčík

Relační modelování biologických dat

Katedra softwarového inženýrství

Vedoucí diplomové práce:
Prof. RNDr. Jaroslav Pokorný, CSc.

Studijní program:
Informatika

Na tomto místě bych chtěl poděkovat vedoucímu diplomové práce Prof. RNDr. Jaroslavu Pokornému, CSc. za jeho rady a připomínky, které pomohly při vytváření tohoto textu. Také děkuji RNDr. Ireně Mlýnkové a Doc. RNDr. Tomáši Skopalovi, Ph.D. za ochotu a poskytnuté konzultace.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 7. srpna 2007 Roman Krejčík

Obsah

Název práce: Relační modelování biologických dat
Autor: Roman Krejčík
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Prof. RNDr. Jaroslav Pokorný, CSc.
e-mail vedoucího: Jaroslav.Pokorny@mff.cuni.cz

Abstrakt: Některé aplikace zpracování dat v databázových systémech vyžadují práci s hierarchicky uspořádanými daty. Informace takového charakteru se často vyskytují v biologických databázích, ale lze se s nimi setkat i v jiných oblastech. Efektivní dotazování specifické pro hierarchie vyžadují vhodnou reprezentaci grafových struktur v relační databázi. Tato práce se zabývá popisem známých metod pro ukládání hierarchických dat do relační databáze. Zaměřuje se zejména na vzájemné odlišnosti jednotlivých metod a zkoumá vhodnost jejich použití pro různé typy hierarchií. Součástí práce jsou také experimenty nad reálnými kolekcemi biologických dat, které porovnávají efektivitu jednotlivých způsobů uložení prakticky.

Klíčová slova: biologická data, SQL, hierarchie, stromy

Title: Relational modeling of biological data
Author: Roman Krejčík
Department: Department of Software Engineering
Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.
Supervisor's e-mail address: Jaroslav.Pokorny@mff.cuni.cz

Abstract: Some applications of data processing in database systems work with hierarchically structured data. Information of such kind is often present in biological data, but it can be also found in another scope. An efficient execution of queries specific for hierarchy needs proper representation of graph structures in a relational database. This thesis deals with a description of known methods for storing of hierarchical data into a relational database. It aims particularly to distinct properties of separate methods and examines fitness of usage for various hierarchies. The thesis includes experiments with real collections of biological data, which are used to compare effectiveness of particular methods in practical way.

Keywords: biological data, SQL, hierarchy, trees

Kapitola 1

Úvod

Současná bioinformatika pracuje s rozsáhlými soubory dat, které je nutné kategorizovat a strukturovat pro jejich další zpracování. Obsáhlé kolekce biologických dat pocházejí především z procesu mapování genomu živých organizmů a výzkumu v oblasti molekulární biologie. Vztahy mezi databázovými objekty mohou mít obecně různou podobu. V bilogických datech je jedním z častých vztahů relace nadřazenosti a podřízenosti. V takovém případě informace obsahují hierarchické struktury. S hierarchiemi se setkáváme i v jiných oborech, avšak biologická data jsou specifická rozsahem a tedy i nutností efektivního zpracování. Příkladem hierarchických dat může být taxonomie živočišných druhů nebo klasifikace proteinů podle jejich původu a podobnosti. Z nebiologických oblastí je příkladem souborový systém disku, organizační struktura zaměstnanců nebo katalog zboží v elektronickém obchodě.

Tyto zdroje dat jsou často zpřístupňovány uživatelům pomocí informačních systémů a webových aplikací. Jako jejich úložiště je ve většině případů volena relační databáze. Výhodou je množství implementací, od řešení poskytovaných zdarma[1] až po sofistikované komerční produkty[2]. Aplikace poskytují data paralelně mnoha uživatelům, což klade nároky na vhodné uložení a použité indexační struktury.

1.1 Cíl práce

Cílem této diplomové práce je zmapovat existující metody pro ukládání stromových struktur do relační databáze a především pak prakticky ověřit jejich vhodnost pro různá použití. Relační databáze je dobrým nástrojem pro data tabulkového charakteru. Hierarchická data mají jinou podobu a jejich uložení

[1] například MySQL či PostgreSQL
[2] například Oracle nebo MSSQL

v tabulce je složitější. I když standard SQL:1999 umožňuje rekurzivní dotazy, jejich dostatečná efektivita pro potřeby rozsáhlých hierarchií není zaručena. V mnoha databázových implementacích navíc nejsou rekurzivní dotazy zatím podporovány. Některé databázové produkty také poskytují vlastní rozšíření jazyka SQL pro dotazování nad hierarchiemi, jsou to ovšem jen speciální případy standardního rekurzivního dotazu[3]. Tato rozšíření ani standardní rekurzivní dotazy nebudou v této práci zkoumána. Zaměříme se pouze na to, jak ukládat hierarchická data pomocí prostředků, které jsou široce podporovány různými databázovými systémy. Způsobů uložení hierarchií do tabulky existuje několik, proto u jednotlivých metod porovnáme jejich možnosti a tvorbu dotazů.

Výhodou takového řešení je přenositelnost a použitelnost na libovolné implementaci relační databáze. Znalost vlastností jednotlivých metod umožňuje zvolení té nejvhodnější pro konkrétní případ, ve kterém jsou specifikovány dotazy a jejich požadavky na výkon. Použití běžných databázových tabulek také umožňuje snadné rozšíření popsaných metod o další informace relevantní k ukládané hierarchické struktuře ve specifických případech.

Část metod uvedená v této práci je popsána v anglické literatuře, zejména pak v [1]. Existující české popisy metod se příliš nezabývají vhodným použitím indexů, řazením výsledků dotazů a ne vždy používají optimální varianty dotazů. Proto jedním z cílů je vytvoření komplexního přehledu k dané problematice v českém jazyce.

Další přínos práce by měl být v praktickém změření efektivity různých metod nad reálnými biologickými daty. Porovnání metod bude prováděno nad daty, která jsou snadno dostupná na webu ve veřejných archivech. Velikost kolekcí dat pak zajišťuje dostatečnou relevanci měření. Během porovnávání bude zohledňována zejména rychlost pro typické dotazy a podpora dalších stromových dotazů. Dalšími kritérii budou velikost pomocné indexační struktury a použitelnost při častých aktualizacích dat.

1.2 Členění práce

Práce je rozdělena na několik kapitol.

První kapitola obsahuje úvod do problematiky.

Kapitola 2 se zabývá obecnými vlastnostmi a klasifikací hierarchií a požadavky na jejich uložení do relační databáze.

Kapitola 3 pak popisuje známé metody pro ukládaní hierarchií z teoretického hlediska. Zkoumá strukturu tabulek, tvorbu dotazů a unikátní vlastnosti

[3]Důvodem vlastního formatu dotazu je vznik nativních rozšíření před vydáním rekurzivních dotazů ve standardu SQL.

jednotlivých uložení.

V kapitole 4 jsou jednotlivé metody podrobeny praktickému testu. Nad reálnými kolekcemi dat je měřen čas provádění nejběžnějších dotazů.

V poslední kapitole je pak zhodnocena vhodnost použití jednotlivých metod v závislosti na ukládané hierarchii.

K této práci je také přiloženo CD, jehož obsah je podrobně popsán v příloze A. Disk obsahuje zejména nástroje pro provedení testů (například kompletní podobu skriptů uvedených v následující příloze).

Příloha B obsahuje výpis skriptů (nebo alespoň jejich podstatné části) použitých pro převedení reálných dat z poskytovaných formátů do relační databáze a skripty pro samotné porovnání jednotlivých metod.

Kapitola 2

Uložení hierarchií do databáze

Tato kapitola se zabývá metodami pro ukládání hierarchických struktur do relační databáze. Nejprve však nadefinujeme, co přesně znamená pojem hierarchie, jak mohou hierarchie vypadat a jaké mají společné vlastnosti.

2.1 Hierarchické struktury

Intuitivně hierarchií rozumíme systém, kde mezi objekty existují vztahy nadřazenosti a podřízenosti. Abychom mohli systém nazývat hierarchií, klademe na takový systém ještě další podmínky. Například není možné, aby byly dva objekty sobě nadřazené navzájem.

Pro přesnou definici hierarchie budeme potřebovat pojmy z teorie grafů. Základní termíny lze nalézt v [2].

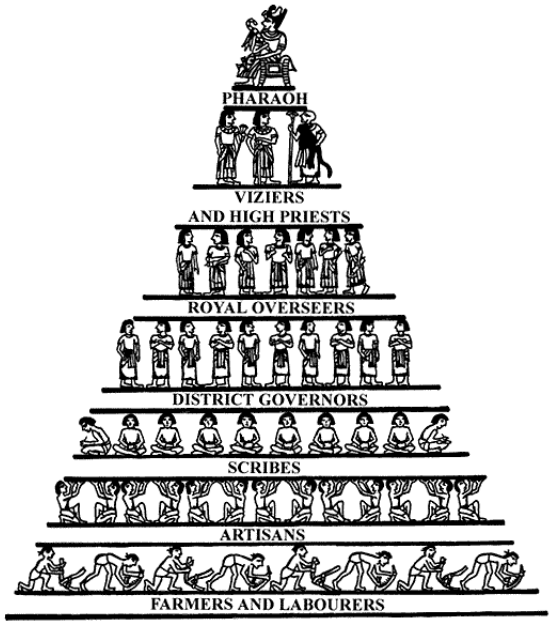
Definice 1. *Hierarchie je acyklický orientovaný graf, kde neexistuje kružnice s pouze jednou odlišně orientovanou hranou*[1].

Uzly grafu reprezentují jednotlivé objekty systému a hrany vedou mezi přímými nadřízenými a podřízenými. Jakým směrem jsou hrany orientovány není pro naše účely příliš důležité, buď mohou představovat vztahy nadřazenosti nebo podřízenosti. Otočením všech hran grafu dostaneme graf opačné vlastnosti.

Acyklický graf může obsahovat kružnici, ale nesmí být všechny hrany kružnice orientovány stejným směrem a tvořit tak smyčku. Tím je vyřešena intuitivní podmínka o neexistenci nadřazenosti dvou objektů navzájem. Také tím není dovolena kruhová nadřazenost navzájem více než dvou objektů.

Existence orientované cesty mezi dvěma uzly představuje nepřímou nadřazenost (a podřízenost). Nepřímý vztah je takový, kde mezi objekty figuruje

[1] Důvod této podmínky vysvětlíme dále.



Obrázek 2.1: Hierarchie (zdroj www.civilization.ca)

jeden nebo více prostředníků. Z toho také plyne, že nepřímé vlastnosti jsou tranzitivní. To odpovídá další intuitivní představě o hierarchii, kde podřízení podřízeného jsou podřízeni i původnímu objektu. Pokud budeme mluvit pouze obecně o nadřazenosti nebo podřízenosti, budeme tím mínit přímý i nepřímý vztah.

Podmínka z definice na neexistenci kružnice s pouze jednou opačně orientovanou hranou zaručí, že graf nebude obsahovat hranu tam, kde už je nepřímá nadřazenost přes další uzly. Jinými slovy zamezí tomu, abychom hranou spojili uzly, mezi kterými již vede orientovaná cesta. Příklad zakázané a povolené kružnice je na obrázku 2.2. Z podmínky také plyne, že graf nemůže obsahovat kružnici délky tři.

Někdy se za hierarchii považuje pouze systém, kde každý objekt mimo nejvýše nadřízeného je podřízen právě jednomu objektu. Takové omezení naše definice pro obecné hierarchie nezavádí, neboť existují reálné struktury, které bychom intuitivně jako hierarchii označili a přitom toto omezení nesplňují. Jednoduchým příkladem takového systému je katalog zboží, kde některé produkty spadají do více kategorií. Příkladem z oblasti biologických dat je genová ontologie, použitá jako testovací sada dat a popsaná v odstavci 4.1.3.

Obrázek 2.2: Zakázaná a povolená kružnice

2.1.1 Hierarchie jako uspořádání

Tranzitivní uzávěr hierarchie obsahuje všechny přímé i nepřímé vztahy[2]. Můžeme si všimnout, že takto vzniklý graf nepřímých vztahů odpovídá relaci ostrého uspořádání. To není náhoda, neboť slovo hierarchie v běžné řeči označuje systém kde jsou objekty také uspořádány. Funguje to i opačně, každé uspořádání definuje hierarchii. Inverzním postupem (odebráním tranzitivních hran) dostaneme z uspořádání graf hierarchie, který obsahuje pouze přímé vztahy.

Znázorňování hierarchií

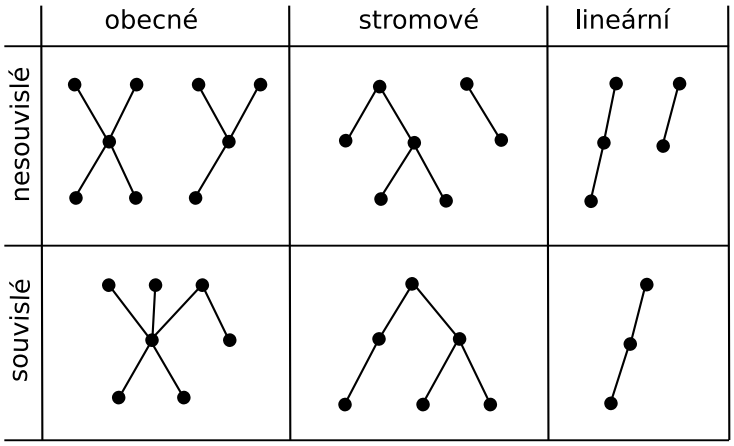
V obecných grafech se orientace běžně znázorňuje šipkou mezi uzly. Pro hierarchie je mnohem přirozenější znázorňovat nadřazenost zakreslením nadřazeného uzlu nad podřízené a spojením hranou bez šipky. To odpovídá zobrazení uspořádání, tak jak je popsáno v [2]. Jelikož hierarchii lze reprezentovat také uspořádáním, je takové znázornění vždy možné.

Pro znázornění všech hierarchií ve zbytku práce bude použita uvedená konvence.

2.1.2 Typy hierarchií

Hierarchie může mít různé podoby. Pracovat efektivně v databázi s obecnou hierarchií je složitější, než pracovat s vybranou užší třídou hierarchií. Zkusíme tedy najít kritéria, podle kterých je můžeme klasifikovat. Na obrázku 2.3 jsou příklady hierarchií s dále popsanými vlastnostmi.

[2]Tranzitivní uzávěr orientovaného grafu přidá orientovanou hranu všude tam, kde existuje mezi uzly orientovaná cesta (orientace hrany je shodná s orientací cesty), což je přesně naše definice nepřímého vztahu.



Obrázek 2.3: Typy hierarchií

Souvislé hierarchie

Definice 2. *Řekneme, že hierarchie je souvislá, jestliže mezi každými dvěma uzly existuje cesta bez ohledu na orientaci*[3].

Metody pro ukládání hierarchických dat uvedené v této práci lze bez obtíží aplikovat i na nesouvislé struktury. Efektivita uvedených metod není pro nesouvislé hierarchie horší než pro souvislé. Touto vlastností se proto při popisu jednotlivých metod uložení nebudeme příliš zabývat. Všechny hierarchie budeme implicitně považovat za souvislé, kromě několika málo případů, kde to bude explicitně zmíněno.

Stromové hierarchie

Definice 3. *Řekneme, že hierarchie je stromová, jestliže každý uzel má nejvýše jednoho přímého nadřízeného.*

Pro efektivní ukládání hierarchických dat do databáze je tato vlastnost podstatná. Souvislá stromová hierarchie je strom. Nesouvislá stromová hierarchie je les, tedy množina několika stromů.

Struktura stromových hierarchií je jednoduší než u obecné hierarchíe a umožňuje použití většího množství metod uložení. Dobrá zpráva je, že převážná většina reálných dat má právě takový charakter. Tato práce je proto zaměřena především na popis a měření metod pro ukládání stromů do databáze. O ukládání obecných hierarchií se zmíníme v sekci 3.7.

[3]V grafové terminologii je taková vlastnost nazývána slabou souvislostí.

Pro popis stromových hierarchií budou na některých místech pro názornost používány pojmy z teorie grafů:

- Přímo nadřazený uzel se nazývá otec.
- Přímo podřízený uzel je syn.
- Synové jsou vzájemně sourozenci.
- Uzly bez podřízených označíme jako listy.

Pro strom vždy existuje uzel nazývaný kořen, který je nadřazený všem ostatním. Ve stromové hierarchii také pro každé dva uzly platí, že mají buď společného nadřízeného nebo jsou ve vztahu nadřízený podřízený.

Pro účely této práce se budou také hodit některé definice z teorie grafů:

Definice 4. *Hloubka uzlu je rovna délce cesty od uzlu ke kořeni stromu.*

Hloubka kořene je tedy 0, hloubka ostatních uzlů je hloubka rodiče zvětšená o 1.

Definice 5. *Hloubka stromu je maximum z hloubek uzlů stromu.*

Lineární hierarchie

Triviálním případem hierarchie je lineární uspořádání. Taková hierarchie se nikde nevětví a všechny uzly mají nejvýše jednoho podřízeného i nadřízeného. Pro efektivní uložení do databáze a následné dotazování by stačilo jednotlivé uzly indexovat přirozeným číslem podle jejich pozice. Tímto případem se proto nebudeme vůbec zabývat.

2.2 Požadavky na uložení

Na uložení uzlů hierarchie do tabulky v relační databázi klademe určité požadavky.

V prvé řadě je nutné zachytit samotné vztahy mezi uzly. Základní a nejjednodušší způsob jak to provést nazveme triviální uložení a podrobně ho rozebereme v sekci 3.1. Každému uzlu grafu odpovídá jeden řádek tabulky. V závislosti na kontextu tak budeme objekty hierarchie označovat různými termíny. Na uzly též můžeme nahlížet databázovým pohledem a mluvit tak o řádkách či záznamech.

Hierarchická data ukládáme do databáze proto, abychom se na ně mohli později dotazovat. Po databázi vyžadujeme složité informace, které nelze

snadno vyčíst z atributů jednotlivých uzlů. Dotazy na hierarchické struktury jsou typické tím, že často pracují s tranzitivním uzávěrem hierarchie[4]. Příkladem je dotaz na množinu všech podřízených k vybranému uzlu. Tranzitivní uzávěr relace ovšem nelze v běžném SQL-92 vyjádřit[5], takže nelze ani zapsat SQL dotaz nad tabulkou, která strukturu hierarchie zachycuje pouze pomocí přímých vazeb.

Proto dalším úkolem uložení je rozšíření vlastních dat o další pomocné sloupce, které umožní získat výsledek takových dotazů pomocí jednoho SQL příkazu. Bez pomocných dat je možné provádět příkazy `SELECT` rekurzivně a jejich výsledky zpracovávat mimo databázi (konstrukce a nutnost použití dalších dotazů závisí na výsledcích předchozích). Takové řešení je sice funkční, ale už si nevystačíme pouze s jazykem SQL a navíc ani předem nevíme, kolik příkazů `SELECT` můsíme pro získání žádané informace provést[6].

Účel pomocných dat je také zefektivnit provádění dotazů. Použití pomocných sloupců by také mělo dotaz co nejvíce zjednodušit, zejména omezit operace spojování tabulek a množinové operace nad poddotazy.

Místo dalších sloupců v tabulce uzlů mohou být pomocná data uložena též samostatně ve vlastní tabulce. Pomocná data, ať už jsou uložená v databázi jakkoliv, budeme v dalším textu také někdy nazývat indexační strukturou.

Nevýhodou rozšířených metod je režie na udržování konzistentních pomocných hodnot. Možností, jak rozšířit triviální uložení, je více. Každá taková metoda bude pro konkrétní dotazy jinak efektivní.

2.2.1 Aktualizace hierarchií

Aktualizací hierarchie rozumíme operaci, při které se v grafu změní množina hran. To znamená, že se přidávají nebo odebírají vazby nadřízený-podřízený. Takovou operací je například i jednoduché přidání uzlu do stromu, při kterém vznikne jedna nová hrana. Ne každá `UPDATE` klauzule nad tabulkou uzlů musí nutně měnit i hierarchii. Pokud dotaz mění pouze vnitřní data uzlů, tak pro nás není ničím zajímavý.

Vazby v hierarchii mění následující úpravy:

přidání a odebrání listu
: Toto je nejjednodušší úprava. V hierarchii se společně s listem přidá nebo odebere jedna hrana.

[4]Tím se rozumí běžný tranzitivní uzávěr orientovaného grafu z grafové teorie. V případě hierarchie to také znamená tranzitivní uzávěr relace přímé nadřazenosti (nebo podřízenosti).

[5]Získání tranzitivního uzávěru pomocí různých rozšíření SQL se věnuje odstavec 3.1.4.

[6]Kdybychom to věděli tak vhodným vnořením SELECT klauzulí a použitím množinových operací můžeme získat výsledek jediným příkazem, byť složitým.

smazání celého podstromu
: Tato operace může potencionálně smazat velkou část uzlů. Jinak je to přímočará operace a není nijak problematická.

vložení vnitřního uzlu
: V grafových termínech se tato operace nazývá dělení hrany. Nový uzel je vložen ve stromu mezi otce a syna. Původní otec se stane otcem nového uzlu a nový uzel je otcem původního syna. Původní hrana je odstraněna a místo ní vzniknou dvě nové. Původní přímá hierarchická vazba se tak stane vazbou nepřímou.

smazání vnitřního uzlu
: Při odstranění vnitřního uzlu musíme vyřešit jak „zaplnit uvolněné místo“. To znamená, že musíme vyřešit, co uděláme se syny odstraněného uzlu a jejich podstromy.

  - Můžeme je také smazat, pak to není smazání vnitřního uzlu, ale smazání celého podstromu.
  - Další neobvyklé řešení je nedělat nic a nechat synovské podstromy oddělené. Pokud byla hierarchie souvislá, stane se nesouvislou.
  - Obvyklé řešení je jednoho ze synů podřídit otci smazaného uzlu a ostatní sourozence podřídit povýšenému synovi. To je typické řešení například pro hierarchii zaměstnanců, kdy je jeden z podřízených povýšen na uvolněnou pozici.
  - Poslední možností je rovnocenné podřízení všech synů přímo otci smazaného uzlu.

přesun podstromu
: Kořen podstomu získá jiného nadřízeného. Jedna přímá vazba tak zanikne a vznikne místo ní jiná. I když přímé vazby uvnitř podstromu zůstávají nezměněny, každému uzlu podstromu se zmení množina nepřímých nadřízených.

Jiné úpravy jsou jenom kombinací výše uvedených.

Jednou z vlastností jednotlivých metod uložení, kterou lze zkoumat a porovnávat, je efektivita aktualizací stromové struktury. V aplikacích, kde úpravy hierarchie probíhají zřídkakdy nebo se provádí dávkově mimo provozní špičku, není dopad aktualizací významný. Naopak pokud aktualizace probíhají často, je složitost aktualizací vlastnost významná pro výběr vhodné metody uložení.

Pro dvě metody uložení mohou být stejné aktualizace různě složité. Složitost aktualizace závisí na podobě indexační struktury dané metody. Množství

upravených záznamů se odvíjí od závislosti pomocných hodnot na vazbách hierarchie. I když úprava sama o sobě modifikuje jediný uzel, hodnoty pomocných sloupců mohou být stále ovlivněny pro velkou část záznamů. V extrémním případě je nutné při přesunu nebo smazání jediného uzlu přepočítání pomocných hodnot na většině záznamů. Jiným indexačním strukturám stačí pouze úprava pomocných hodnot v řádce příslušné editovanému uzlu.

Množství změněných řádek je tak jediná relevantní veličina, kterou můžeme u jednotlivých metod porovnávat. Konkrétní aktualizaci na konkrétní metodě uložení lze podle složitosti zařadit do jedné ze čtyř kategorií:

lokální změna
: Žádné další modifikace pomocných dat a tedy minimální náročnost aktualizačních operací. Do této skupiny patří především modifikace triviálního uložení stromových dat.

  Prakticky použitelné metody pro ukládání stromů mají bohužel složitost aktualizačních operací obvykle větší.

úprava podstromu
: Během aktualizace se musí upravit pomocné hodnoty u všech uzlů ležících pod místem změny. Pro listy je to stejné jako lokální změna. Při změně vazeb v blízkosti kořene se musí upravit téměř všechny řádky v tabulce.

úprava nadřazených uzlů
: Po přidání nebo odebrání uzlu se musí aktualizovat všechny řádky uzlů ležících na cestě ke kořeni. Zejména pak řádka příslušná samotnému kořeni, která se tak musí aktualizovat při každé změně struktury.

  Každá aktualizace tak musí získat zámek pro zápis na této řádce. Z toho plyne, že změny nelze provádět paralelně. Proto i když počet uzlů přímo ovlivněných aktualizací je průměrně menší než v předchozím případě, je tato metoda méně efektivní.

  Tato kategie složitosti je uvedena jen pro úplnost. I když lze teoreticky navrhnout metody spadající do této kategorie, žádná prakticky použitelná není známá. Proto ani tato práce podobnou metodu neobsahuje.

globální změna
: Změna struktury stromu způsobí modifikaci všech uzlů tabulky nebo alespoň její významné části. Situace je obdobná jako v předchozím případě, obzvláště pokud je nutné ve většině případů upravovat stále stejnou množinu řádek. Potom si transakce konkurují při získávání zámků pro zápis. To znemožňuje použít aktualizace z této kategorie pro často

měněné kolekce. Pro vytížené databáze to může znamenat nemožnost provádět takové aktualizace úplně. Metody, jejichž aktualizace spadají do této kategorie, jsou vhodné zejména pro statické kolekce nebo kolekce upravované pouze dávkově ve vyhrazené době.

Pro metody popsané dále v práci jsou obvyklé dvě závislosti pomocných dat. Prvním případem je závislost na všech nadřazených uzlech. To znamená úpravu pomocných dat při změně libovolného nadřízeného. Každá aktualizace stromu pak způsobí úpravu podstromu. Jelikož u přidání a odebrání listu neexistuje žádný podstrom, jsou tyto speciální případy pouze lokální změnou. Druhým obvyklým případem je závislost pomocných hodnot na celém stromě. To má za následek globální změnu při každé aktualizaci.

Klasifikaci úprav do tříd podle náročnosti lze použít i na obecné hierarchie, jen je třeba zobecnit některé termíny používané pro stromy. Například tam kde se mluví o podstromu, to pro obecnou hierarchii znamená všechny podřízené uzly. Obdobně, list je pro nestromové hierarchie takový uzel, který nemá žádné podřízené apod.

2.3 Dotazování

2.3.1 Obvyklé dotazy

Pro relevantní porovnání jednotlivých metod potřebujeme znát typické dotazy na stromovou strukturu. Obvyklé dotazy budou pro každou aplikaci trochu jiné, avšak obecně lze očekávat následující požadavky:

- Pro daný uzel najít otce a syny.
- Najít všechny podřízené uzly v libovolné hloubce, tedy získat celý podstrom pod daným uzlem.
- Od uzlu zjistit cestu ke kořeni stromu.
- Získat všechny listy stromu.
- Pro dva různé uzly najít nejbližšího společného nadřízeného.

První tři typy dotazů využívají například webová rozhraní databází použitých v kapitole 4. Ta s výpisem atributů vybraného uzlu informují uživatele o jeho zařazení v hierarchii a některých jeho podřízených.

Další informace, které mohou být požadovány po databázi, jsou informace relevantní k hloubce uzlů:

- Zjistit celkovou, případně průměrnou hloubku celého stromu nebo vybraného podstromu.
- Zjistit vzdálenost uzlu od kořene.
- Najít podřízené uzly do předem dané hloubky.

2.3.2 Konvence pro SQL příklady

Při ukládání záznamů do tabulek bude primárním klíčem vždy atribut *id*, kladné celé číslo nezávislé na datech. Vlastní data příslušná jednomu uzlu mohou být v praxi libovolně rozsáhlá, s vazbami na další tabulky. V našem případě budou reprezentována pouze atributem *název* obsahujícím jméno uloženého objektu.

Některé dotazy vyžadují použití konstrukcí, které jsou sice široce podporovány, ale nejsou standardizovány normou. Vzhledem k použití PostgreSQL při porovnávání metod v kapitole 4, budou i v této kapitole všechny ukázky SQL kódu uváděny ve formě kompatibilní s databází PostgreSQL. Důležité je to zejména tam, kde není norma jednoznačná nebo se implementace různých databázových systému odlišují. To nebude rozhodně častý případ, většina dotazů si vystačí s přenositelným základem SQL. Odchylky se týkají především odlišných názvů funkcí, například pro práci s řetězci. Tato práce vychází z verze aktuální v době jejího vzniku, což je konkrétně PostgreSQL 8.2.

Proměnné v zápisu dotazu

Dotazy uvedené v příkladech obvykle získávají data závislá na jednom uzlu. Skutečný dotaz pak obsahuje atributy takto vztaženého uzlu. Když se například ptáme na podstrom konkrétního uzlu, potom dotaz musí obsahovat hodnoty některých sloupců tohoto uzlu. Tyto proměnné hodnoty musíme vhodným způsobem v příkladech zapsat. K tomu budeme užívat znak $, za kterým bude následovat jméno příslušného sloupce. Ve skutečném dotazu pak bude proměnná nahrazena hodnotou sloupce vztaženého uzlu. V nejednoznačných případech, například když je proměnná součástí textového řetězce, bude používán formát se složenými závorkami ${*proměnná*}.

Výsledek dotazu

Pro jednoduchost budeme při dotazech na uzly vracet ve výsledku všechny sloupce tabulky. To znamená, že budeme používat `*` v `SELECT` klauzuli. V praxi můžeme chtít pouze vybrané sloupce nebo pouze jediný sloupec,

což je ale jen drobná modifikace dotazu. Jelikož každá řádka tabulky uzlů reprezentuje jeden konkrétní uzel, můžeme o řádkách mluvit jako o samotných uzlech. Běžně budeme říkat, že dotaz vrací seznam uzlů i když technický jsou to řádky které uzly reprezentují v databázi.

V dotazech, kde nechceme získat jenom samotné uzly, ale potřebujeme nějakou další informaci, případně se ptáme na skalární hodnoty, musí být jednotlivé sloupce výsledné relace explicitně uvedeny. V takovém případě budou pojmenovány výstižným názvem.

2.3.3 Řazení výsledku dotazu

Databáze vrací výsledné řádky uspořádané nějakým způsobem, který lze ovlivnit klauzulí `ORDER BY`. Situacemi, kdy je seřazení výsledných uzlů podstatné, se budeme zabývat v této sekci.

U některých dotazů existuje pouze jedna smysluplná možnost jak může být výsledek seřazen. Například cesta od uzlu ke kořeni je daná posloupnost uzlů. Pokud se na ní dotážeme, potom očekáváme setřídění směrem od kořene k nejhlubšímu uzlu a nebo obráceně (což je jen změna směru řazení v dotazu).

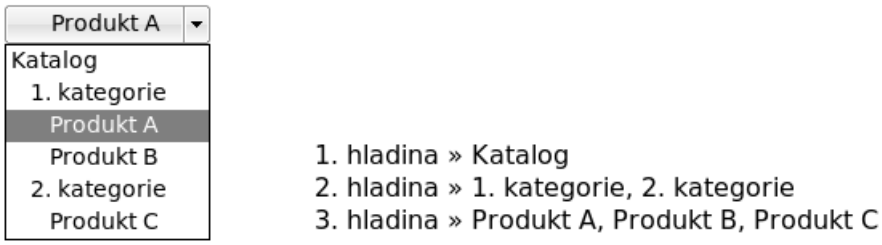
U jiných dotazů není uspořádání striktně dáno a existuje více možností. Tradiční algoritmy procházení stromu rozlišují:

- průchod do šířky,
- průchod do hloubky – zde se navíc rozlišuje, zda-li je vnitřní uzel zařazen do výsledku před průchodem podstromu a nebo až po něm[7].

Stejnými způsoby mohou být řazeny i uzly ve výsledku některých dotazů. Nejtypičtější dotaz, který má smysl takto řadit, je dotaz na celý podstrom. Ten vlastně odpovídá skutečnému průchodu podstromem s počátkem v daném uzlu. Více možností řazení existuje u všech dotazů, kde je výsledek nějakou vybranou částí podstromu. Příkladem je dotaz na všechny listy.

Při praktickém použití je procházení do hloubky výhodné při výpisu stromu ve stylu unixového příkazu `tree` (co řádek to jeden uzel s patřičným odsazením). Takový výpis lze například vidět na webu tam, kde se v rozbalovacím menu nabízí hierarchické kategorie (viz. obr. 2.4). Procházení do šířky je vhodné pro tradiční vodorovné zobrazení s kořenem nahoře a s uzly ve stejné hloubce zarovnanými do jedné řádky. Také je jednodušší omezit výsledek jen na uzly do určité hloubky.

[7] V grafových algoritmech jsou různé průchody stromem do hloubky běžně označovány názvy *preorder*, *inorder* a *postorder*.



Obrázek 2.4: Praktické použití odlišných způsobů řazení

V reálné aplikaci tedy požadujeme jeden konkrétní způsob řazení v závislosti na dalším využití dat. Problém je, že metody pro ukládání stromů většinou umožňují pouze jediný způsob, jak výsledek seřadit. To je dáno zejména charakterem pomocných sloupců, které konkrétní způsob řazení vynucují. V případě, že metoda podporuje více způsobů, bývá i tak zpracování jednoho z nich efektivnější.

Dotazování často probíhá z aplikace napsané v nějakém programovacím jazyce, která data prezentuje uživateli. Taková aplikace je schopná výsledek ještě před prezentací seřadit jiným způsobem, než jakým byl vrácen z databáze. To však v takové aplikaci vyžaduje načtení všech řádek do paměti a jejich netriviální zpracování. Pokud je výsledek již ve správném pořadí, lze data číst proudově (to znamená postupně odebírat řádky výsledku), bez nutnosti načtení všech najednou. Seřazení dat již na úrovni databáze je v každém případě výhodnější, pokud ovšem lze provést. Proto také možnosti řazení výsledku mohou přispět k výběru vhodné metody pro uložení stromu. Pro jednotlivé metody uložení stromových dat tak budeme zkoumat tuto vlastnost u dotazů, pro které to má smysl.

Kapitola 3

Metody uložení

Ve této kapitoly jsou popsány jednotlivé metody pro uložení hierarchických dat. Pro stromové hierarchie je popsáno šest způsobů:

- triviální uložení,
- metoda ukládání cesty ke kořeni,
- cesta ke kořeni jako složený typ,
- tabulka vazeb,
- vnořené množiny,
- uložený průchod do hloubky.

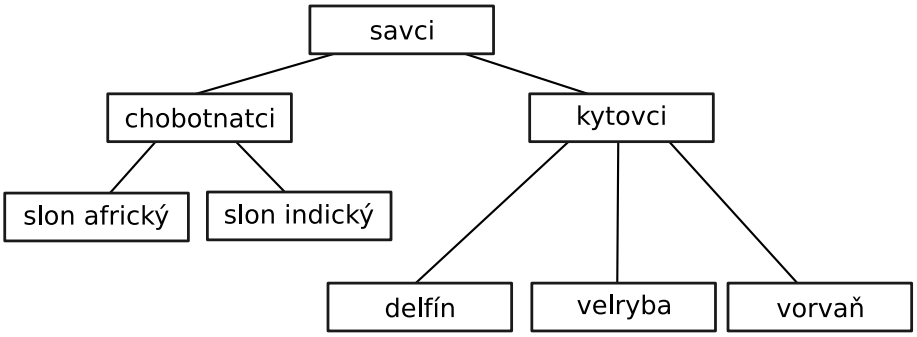
V sekci 3.7 je potom rozebráno ukládání obecných hierarchií. V obecném případě již není možné vybírat z širokého spektra různých metod.

Ukázková data

Na obrázku 3.1 je jednoduchý strom, jenž bude u všech popsaných stromových metod sloužit jako příklad. Příklad je inspirován sadou testovacích dat NCBI, která obsahuje taxonomii známých živočišných druhů.

3.1 Triviální uložení

Triviální uložení neukládá do relační databáze žádná redundantní pomocná data. Každá řádka obsahuje pouze odkaz na nadřazený záznam, což plně zachycuje stromovou strukturu.



Obrázek 3.1: Jednoduchá taxonomie – ukázková data

V praxi je tato metoda použitelná pouze pro malé kolekce dat nebo v případě, že nepotřebujeme provádět složité stromové dotazy. Metoda je uvedena ze dvou důvodů.

Prvním je, že stejné základní sloupce budou obsahovat všechny dalších metody. Teoreticky to není nutné, protože v pomocných sloupcích ostatních metod je stromová struktura také plně popsána a obešli bychom se tak u nich bez sloupce s odkazem na nadřazený záznam. Avšak v praktických aplikacích taková redundance zjednodušuje práci s daty, neboť odkaz na nadřazený uzel je často využívaný atribut. To také znamená, že všechny popsané vlastnosti týkající se sloupců této tabulky budou platit i pro příslušné sloupce všech ostatních metod.

Druhý důvodem je využití v kapitole 4, kde budeme reálná data z jejich nativních formátů převádět vždy nejdříve na toto jednoduché uložení. Následné transformační skripty na ostatní způsoby uložení budou očekávat data v tabulce, která má sloupce triviálního uložení. To mimo jiné umožňuje převádět kterékoliv uložení na libovolné jiné přímo, neboť tabulky ostatních metod základní sloupce také obsahují.

3.1.1 Struktura tabulky

```
CREATE TABLE triviální (
  id INTEGER NOT NULL PRIMARY KEY,
  nadřazený INTEGER NULL,
  název VARCHAR(100)
);
```

Co se indexů týče, sloupec *název* nesouvisí se stromovou strukturou, takže zbývá jediný kandidát *nadřazený*. Index na tomto sloupci je nutný kvůli ci-

zímu klíči, který definujeme v odstavci 3.1.2. Navíc index pomůže při dotazování na podřízené uzly, což je operace velmi častá. Index vypadá následovně:

```
CREATE INDEX idx_triviální_nadřazený
  ON triviální(nadřazený)
```

Pro ukázkový příklad bude obsah tabulky vypadat takto:

id	nadřazený	název
1		savci
2	1	chobotnatci
3	2	slon africký
4	2	slon indický
5	1	kytovci
6	5	delfín
7	5	velryba
8	5	vorvaň

3.1.2 Referenční integrita

Tabulka může definovat pouze integritní omezení pro vazby na jiné tabulky nebo omezit hodnoty jednotlivých sloupců na vhodnou doménu. U stromových dat nesmí v odkazech na nadřazený uzel vzniknout cyklus, takže už samotný požadavek na to, aby data byla stromem, nelze obecně prostředky SQL zajistit. Můžeme alespoň zajistit, aby hodnota ve sloupci *nadřazený* odkazovala na existující uzel. V případě importu celého stromu do tabulky v rámci jedné transakce, může být užitečné nastavit cizímu klíči vlastnost `DEFERRABLE`. Ta umožňuje odložit kontrolu integrity až do okamžiku potvrzení transakce. To je užitečné v situaci, kdy jednotlivé záznamy čteme z externího zdroje, ve kterém se některé uzly objevují dříve než jejich nadřízené.

```
  CONSTRAINT fk_triviální_nadřazený FOREIGN KEY (nadřazený)
    REFERENCES triviální(id) DEFERRABLE
```

Stejné omezení budeme předpokládat i u dalších metod kde bude uváděno jednodušší formou již v definici tabulky. Cizí klíč samotný nijak nepomůže při indexování hierarchické struktury. Jeho účel je pouze zamezit některým chybným hodnotám a neplatným datům.

Narušení integrity

Ještě rozebereme, kdy může dojít k narušení integrity dat i přes existenci cizího klíče. Jak už bylo zmíněno, jediný problém představuje cyklus v odkazech na nadřazený uzel, kterému nelze integritním omezením zabránit. V takovém případě by data přestalala být stromem (a ani by nesplňovala podmínky kladené na obecnou hierarchii). Při vkládání nových řádek do tabulky problém nastat nemůže, protože na nové řádky ještě nemůže nic odkazovat. Ani s mazáním řádek není žádná potíž, tím cyklus také nevznikne. Problematická operace je pouze modifikace již existující řádky, při které se změní odkaz na nadřazený uzel. A to pouze modifikace uzlu na který odkazují jiné řádky. Pokud se na záznam budeme dívat jako na uzel a ne jako na řádku tabulky, tak problematický je přesun vnitřního uzlu, konkrétně pak přesun uzlu do vlastního podstromu. Referenční integritu tedy stačí ověřovat pouze při přesunu podstromu. V takovém případě musíme zkontrolovat cíl přesunu, během jiných operací je integritní omezení zajištěno databází.

Integrita pomocných dat

Jiné metody než triviální uložení používají další pomocné hodnoty pro indexaci záznamů. U každé metody jsou omezení pro pomocná data jiná. Referenční integritou těchto pomocných dat se ale nemusíme zabývat. Úpravy pomocných hodnot jsou prováděny aplikací, která záznamy vkládá a modifikuje, případně přímo triggery v databázi. Proto zde nehrozí vložení neplatných dat z vnějšího zdroje. Navíc u všech metod lze vyrobit index nad již vytvořeným stromem, proto také lze nekonzistentní hodnoty stejným způsobem znovu rekonstruovat, až už k nekonzistenci došlo jakýmkoliv způsobem.

3.1.3 Dotazování

Všechny dotazy nutně fungují i pro ostatní metody uložení, takže pokud dále popsané metody neumožňují efektivnější dotaz nebudou u nich stejné dotazy opakovány.

Jednoduchým dotazem bez vnořených `SELECT` klauzulí toho nad triviální uložením příliš mnoho zjistit nelze. Jednou z výjimek je dotaz pro získání přímo podřízených uzlů. Podle zavedené konvence *$id* v dotazu reprezentuje *id* uzlu ke kterému chceme získat jeho syny.

```
SELECT * FROM triviální WHERE nadřazený = $id
```

Získání kořene stromu[1] je také přímočaré, ale není to příliš běžný dotaz.

[1] V případě nesouvislé hierarchie (data tvoří les) získání všech kořenů.

```
SELECT * FROM triviální WHERE nadřazený IS NULL
```

Dotaz pro získání všech listů je už složitější:

```
SELECT * FROM triviální AS t1 WHERE NOT EXISTS
  (SELECT * FROM triviální AS t2 WHERE t1.id = t2.nadřazený)
```

Pomocí příkazu `EXPLAIN` lze nahlédnout, že vnitřní klauzule se bude provádět pro každou řádku samostatně. To vede ke složitosti dotazu $O(n^2)$, kde n je počet záznamů. Pro rozsáhlejší data je taková složitost prakticky nepoužitelná, alespoň pro dotazy které se mají provádět častěji než zřídkakdy.

3.1.4 Rekurzivní dotazy

V případě triviálního uložení nelze prostředky SQL-92 vyjádřit složitější dotazy na tranzitivní uzávěr hierarchie. Takové dotazy jsou však pro hierarchické struktury typické. Příkladem je dotaz na všechny uzly vybraného podstromu. Takové dotazy jsou také nazývány rekurzivními, protože jejich výsledek lze získat rekurzivním prováděním a zpracováním příkazu `SELECT` ve vyšší aplikační vrstvě mimo databázi.

Zmíněný dotaz na podstrom rekurzivně provedeme následovně. Začneme v kořeni podstromu, ke kterému zjistíme jeho přímé podřízené. K takto získaným uzlům opět získáme podřízené a celý postup se rekurzivně opakuje. Záleží jakým způsobem budeme dotazy konstruovat, ale bude potřeba minimálně stejně příkazů `SELECT` jako je hloubka podstromu. Počet nutných příkazů `SELECT` závisí na hloubce a tvaru konkrétního stromu. To je velmi špatná vlastnost a také hlavní důvod pro použití jiných metod uložení v reálné aplikaci.

Norma SQL:1999 zavádí pro rekurzivní dotazy konstrukci `WITH ... UNION ALL`, ale její podpora je zatím v podstatě nulová.

Proprietární rozšíření SQL

Některé databáze implementují vlastní syntaxi pro rekurzivní dotazování. Například Oracle podporuje konstrukci `CONNECT BY`. V takových případech je metoda již použitelná lépe. Odpadá hlavní nevýhoda v nemožnosti získat celý podstrom najednou. Jak již bylo zmíněno v úvodu, zabývat se podrobně rozšířeními pro rekurzivních dotazy není náplní této práce.

3.1.5 Aktualizace hierarchie

Jednou z výhod této metody je snadná aktualizace. Vložení i smazání listu je triviální.

Úprava vnitřního uzlu

Při vložení vnitřního uzlu je třeba pouze úprava jednoho záznamu navíc.

U odebrání vnitřního uzlu je nutné pouze řešit, jak přípojit zpět podstromy. To bylo již řešeno v obecné části o aktualizacích v sekci 2.2.1 a ať už vybereme libovolný způsob, jeho provedení je v triviálním uložení snadné. U jiných metod složité operace jsou tady pouze lokální změnou.

Smazání podstromu

Komplikovanější operací je odebrání celého podstromu. Už při dotazování jsme narazili na problém získání všech uzlů, které obsahuje. Zde potřebujeme jejich identifikátory k tomu, abychom je mohli všechny smazat. U ostatních metod, kde lze uzly podstromu získat jedním dotazem, je tato operace relativně jednoduchá. V této metodě musíme operaci provést jinak.

Jednodušší řešení pro triviální uložení je nechat práci na databázovém systému. K tomu stačí aktivace kaskádového mazání na cizím klíčí pro sloupec *nadřazený*. Při odstranění řádky příslušné uzlu, provede databáze sama odstranění synů a mazání se dále šíří až k listům. Upravená definice cizího klíče v definici tabulky vypadá následovně

```
CONSTRAINT fk_triviální_nadřazený FOREIGN KEY (nadřazený)
  REFERENCES triviální(id) ON DELETE CASCADE DEFERRABLE
```

Pokud není automatické kaskádové mazání žádoucí, nezbývá než strom rekurzivně projít a smazat všechny uzly postupně. V takovém případě ani nevíme kolik dotazů bude na smazání podstromu potřeba. V případě mazání kaskády musí databáze provést stejný postup, takže rychlost obou postupů je obdobná[2], pouze v prvním případě je celá operace provedena automaticky.

Přesun podstromu

Při přesunu celého podstromu stačí změnit odkaz na nadřazený uzel. Na rozdíl od pokročilejších metod tu není nutné aktualizovat žádné pomocné sloupce. Jak bylo zmíněno v 3.1.2, toto je jediný případ kdy je třeba dát pozor na validitu aktualizace. Pokud je zaručena korektnost přesunu, tak je to jednoduchá lokální změna.

Ve skutečnosti jsou však běžně identifikátory modifikovaných uzlů získány od uživatele, či jiného nedůvěryhodného zdroje, a mohlo by se stát, že bez ověření nebudou po modifikaci data už stromem ani hierarchií. Musíme

[2]Při rekurzivním mazání opakovanými dotazy je nutný čas navíc pro posílání dotazů databázi a jejich zpracování.

ověřit, zda-li neleží cíl přesunu ve vlastním podstromu. To lze implementovat jako trigger spouštěný při `UPDATE` nebo musí být korektnost zajištěna mimo databázi.

U této konkrétní metody je ověření komplikované, protože nám opět chybí dotaz na získání podstromu, takže je nutné rekurzivní dotazování. Tím se stává přesun s ověřením korektnosti velmi neefektivní operací. U dále popsaných metod, kde potřebný dotaz existuje, je ověření již relativně rychlé a snadné.

3.1.6 Pomocné atributy

Tabulku lze ještě rozšířit o další pomocné atributy, které nejsou běžně potřebné, ale ve specifických případech mohou být užitečné. Nenesou žádnou informaci navíc, která by už v databázi nebyla uložena a slouží pouze ke zjednodušení a zrychlení vybraných dotazů. Cenou za to je nutnost udržovat tyto pomocné informace aktuální. Také to znamená uložení o něco většího množství dat v tabulce, což ale v dnešní době není příliš podstatné.

je_list
: Příznak určující zda-li je uzel listem, tedy hodnota typu `BOOLEAN`. Může pomoci v případě častého vyhledávání a práce s listovými uzly. Hodnotu je třeba aktualizovat pouze po přídání prvního a nebo odebrání posledního syna. Změna se nikam nepropaguje, takže režie na údržbu je relativně malá.

hloubka
: Celé číslo udávající vzdálenost uzlu od kořene. Údržba aktuálních dat je složitější, protože vložení nebo odstranění vnitřního uzlu ovlivňuje hloubku v celém podstromu.

Některé dále popsané pokročilejší metody umí uvedené informace snadno získat ze svých pomocných dat samy o sobě.

3.2 Metoda ukládání cesty ke kořeni

Tato metoda používá jeden pomocný sloupec, ve kterém udržuje cestu od uzlu ke koření. Sloupec je textový a obsahuje *id* nadřazených uzlů seřazené od kořene až po přímo nadřazený uzel. V jednom textovém sloupci je tak uložena strukturovaná informace, která je dále logický dělitelná na další hodnoty. To sice porušuje hned první normální formu, ale za to získáme jednodušší některé dotazy. Navíc ve sloupci nejsou skutečná data uloženého objektu, ale

pouze pomocná hodnota používaná ke specifickým účelům, takže to ani není významné provinění proti „dobrým mravům“.

Někdy je metoda nazývána také jako genealogický strom, protože z každého záznamu lze přímo vyčíst jeho předky.

3.2.1 Struktura tabulky

```
CREATE TABLE cesta_ke_kořeni (
  id INTEGER NOT NULL PRIMARY KEY,
  nadřazený INTEGER NULL REFERENCES cesta_ke_kořeni(id),
  cesta VARCHAR(500),
  název VARCHAR(100)
);
```

Jelikož podle pomocného sloupce budeme filtrovat záznamy, vytvoříme na něm index. Níže uvedená definice obsahuje za jménem sloupce ještě způsob použití indexu, což je konstrukce specifická pro PostgreSQL. Upřesnění `varchar_pattern_ops` říká, že se hodnoty sloupce mohou porovnávat po jednotlivých znacích bez ohledu na jazykové prostředí. Bez této volby by vyhledávání podle začátku řetězce v jiném, než základním jazykovém prostředí[3] degradovalo z vyhledávaní podle indexu na procházení celé tabulky. V jiných databázových systémech bude pravděpodobně postačovat obyčejná definice indexu. Kromě toho definujeme index na sloupci *nadřazený* stejně jako u triviálního uložení.

```
CREATE INDEX idx_cesta_ke_kořeni_nadřazený
  ON cesta_ke_kořeni(nadřazený);
CREATE INDEX idx_cesta_ke_kořeni_cesta
  ON cesta_ke_kořeni(cesta varchar_pattern_ops);
```

Nabízí se dva způsoby jak přesně ukládat *id* nadřazených uzlů do textového sloupce. Abychom rozlišili jednotlivé hodnoty, musíme je buď oddělit speciálním znakem[4] a nebo identifikátory doplnit na fixní šířku[5] a zapisovat je přímo za sebe. Ve druhém případě musíme znát maximální *id* uzlu, ale zase již z délky cesty poznáme v jaké hloubce stromu uzel leží. Z technických důvodů je vhodné oddělovač přidávat i na konec řetězce za poslední záznam. Lépe se pak s hodnotou pracuje. Můžeme se totiž spolehnout, že za každým *id* je oddělovač a není třeba speciálně ošetřovat poslední prvek v různých podmínkách. Počet oddělovačů také přesně odpovídá hloubce uzlu.

[3] Základní jazykové prostředí je „C locale“, tj. porovnávání podle písmen anglické abecedy.

[4] Budeme používat tečku, ale může to být libovolný znak který není obsažen v *id*.

[5] Číselné identifikátory se na fixní šířku přirozeně doplní zleva znakem 0.

Délka hodnoty cesta

Při vytváření tabulky je nutné definovat maximální délku sloupce *cesta*. Hloubka stromu je v obvyklém[6] případě $O(\log n)$ Počet cifer *id* je také $O(\log n)$. Pro každou hladinu se ve sloupci objeví jedno *id*, případně ještě oddělovač. Celkově je odhad nutné délky pole $O(\log^2 n)$. To je sice příznivý odhad, ale o konkrétní hodnotě velikosti pole nic neříká. Přesnou velikost musíme zvolit podle charakteristiky konkrétních dat. Pokud by během provozu délka pole nedostačovala, lze stále v již existující tabulce maximální délku snadno zvětšit.

Jako příklad použijeme testovací sadu dat NCBI z druhé části práce. Ta obsahuje méně než 350 000 uzlů. To je 6 znaků pro *id* plus jeden znak pro tečku. Maximální hloubka stromu je 41, takže v tomto konkrétním případě je dostatečná délka pole $41 \cdot 7 = 287$ znaků.

Ukázková data

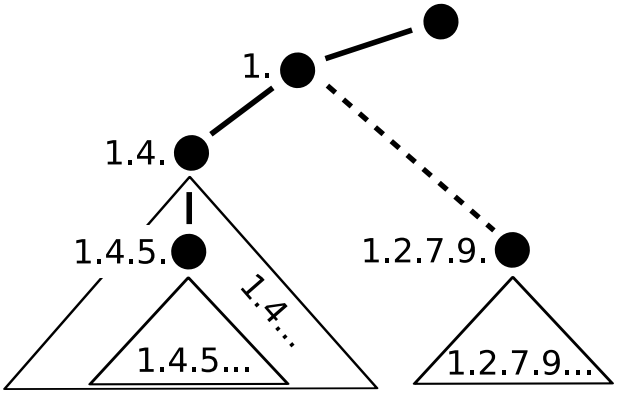
Pro naše ukázková data vypadá první způsob oddělování *id* následovně:

id	**nadřazený**	**cesta**	**název**
1			savci
2	1	1.	chobotnatci
3	2	1.2.	slon africký
4	2	1.2.	slon indický
5	1	1.	kytovci
6	5	1.5.	delfín
7	5	1.5.	velryba
8	5	1.5.	vorvaň

A druhý způsob, pří zarovnání na fixní šířku dvou znaků:

id	**nadřazený**	**cesta**	**název**
1			savci
2	1	01	chobotnatci
3	2	0102	slon africký
4	2	0102	slon indický
5	1	01	kytovci
6	5	0105	delfín
7	5	0105	velryba
8	5	0105	vorvaň

[6]Nebudeme uvažovat degenerované stromy, které mají charakter spojového seznamu. Takovou strukturu reálné stromy nemají.



Obrázek 3.2: Společný prefix podstromu

3.2.2 Dotazování

Podřízené uzly

Pro atribut *cesta* platí, že všechny uzly v jednom podstromu mají společný prefix, stejný jako atribut *cesta* nejhlubšího společného předka (viz. obr. 3.2). Dotazy pro vyhledávání v podstromu se díky této vlastnosti převedou na porovnávání řetězců.

Všechny uzly ležící v podstromu jiného uzlu získáme následovně:

```
SELECT * FROM cesta_ke_kořeni
  WHERE cesta LIKE '${cesta}${id}.%'
```

Zajímavou vlastností metody je možnost seřazení výsledku podle procházení do hloubky i do šířky. Řazení podle sloupce *cesta* způsobí uspořádání jako při procházení do hloubky. Pro výsledek jako při procházení do šířky poslouží drobný trik, a to řazení podle délky sloupce *cesta*, neboť nadřazený uzel má vždy kratší cestu ke kořeni než potomek.

Hloubka uzlů

Sloupec *cesta* také určuje v jaké hloubce leží daný uzel. V případě použití oddělovače je to počet oddělovačů které obsahuje. Pro zapsání dotazu v SQL je třeba využít řetězcových funkcí, které jsou závislé na konkrétním databázovém řešení.

V případě PostgreSQL nejdříve z řetězce odstraníme znaky odlišné od oddělovače a poté zjistíme jeho délku, čímž dostaneme hledaný počet oddělovačů. Dotaz pro zjištění hloubky všech uzlů vypadá následovně:

```
SELECT id, název,
  char_length(regexp_replace(cesta,'[0-9]','','g'))
  AS hloubka_uzlu FROM cesta_ke_kořeni
```

Poslední parametr funkce `regexp_replace` obsahuje příznaky, kde `g` znamená nahrazení všech výskytů.

Při použití druhého způsobu uložení s fixní délkou jednoho prvku je dotaz jednodušší. Stačí celkovou délku dotazu vydělit délkou jednoho prvku. Proměnná `FIXNÍ_ŠIŘKA` v dotazu je počet znaků, na které je zarovnáno *id* ve sloupci *cesta*.

```
SELECT id, název, char_length(cesta) / $FIXNÍ_ŠÍŘKA
  AS hloubka_uzlu FROM cesta_ke_kořeni
```

Nadřazené uzly

Identifikátory uzlů ležících na cestě ke kořeni jsou uvedeny ve sloupci *cesta*. Jejich získání je jednoduché pokud máme možnost provést parsování textové hodnoty mimo databázi. Podle způsobu uložení potřebujeme rozdělit text podle oddělovače nebo ho rozsekat na úseky o fixní šířce[7]. Dotaz by pak obsahoval jednoduchou podmínku `IN` pro *id*.

Dotaz využívající pouze SQL je také možný, ale je velmi neefektivní. Místo hledání podle primárního klíče se provádí porovnávání řetězců. Operátor `||` slouží ke spojení řetězců.

```
SELECT * FROM cesta_ke_kořeni WHERE
  '.' || $cesta LIKE '%.' || id || '.%'
  ORDER BY char_length(cesta)
```

Před cestu dotazovaného uzlu přidáme tečku, aby byl každý identifikátor oddělen tečkou z obou stran. Jinak by podmínka nebyla splněna pro kořen stromu.

3.2.3 Aktualizace hierarchie

Přidání nového listu je snadné. Jeho cestu ke kořeni získáme spojením cesty ke kořeni nadřazeného uzlu a *id* nadřazeného uzlu. Pokud používáme oddělovač, připojíme ho ještě nakonec za otcovo *id*.

Je vidět, že sloupec *cesta* je závislý na cestě ke kořeni nadřazeného uzlu. Proto při změně struktury se musí jeho hodnota změnit v celém podstromu pod místem změny. Smazání listu ani odstranění celého podstromu tak už

[7] Ve většině programovacích jazyků k takovému účelu slouží funkce pojmenovaná *split*.

další záznamy neovlivní. Problém nastává pouze při vkládání nebo mazání vnitřního uzlu. Uzly takto ovlivněného podstromu mají hodnotu sloupce *cesta* složenou ze dvou částí. První část je společná všem ovlivněným uzlům a reprezentuje cestu od kořene podstromu ke kořeni celého stromu. Druhá část je pro každý uzel unikátní a obsahuje cestu ke kořeni podstromu. Vnitřní struktura podstromu zůstává stejná, takže stačí modifikovat pouze společný prefix. Toho lze dosáhnout pomocí jediného příkazu `UPDATE`.

```
UPDATE cesta_ke_kořeni SET
  cesta = replace(cesta,$STARÝ_PREFIX,$NOVÝ_PREFIX)
  WHERE cesta LIKE '%${STARÝ_PREFIX}'
```

Prefix jednoznačně identifikuje daný podstrom, takže ostatní uzly ve stromu nemůže úprava ovlivnit. Stejným způsobem lze také celé podstromy přesouvat.

3.3 Cesta ke kořeni jako složený typ

Některé databázové systémy podporují složené typy. V předchozí metodě jsme potřebovali uložit ke každému uzlu posloupnost čísel a k tomu jsme využili základní typ `VARCHAR`. Pokud databázový systém obsahuje datový typ pole, či nějakou jeho obdobu, potom se nabízí využít ke stejnému účelu ten. Tím odpadne nutnost oddělování uzlů v řetězci a následné rozebíraní textové hodnoty na jednotlivé prvky. Také bude možné konstruovat dotazy pouze v SQL, bez nutnosti jejich předzpracování mimo databázi. Bohužel podpora složených typů se v různých implementacích databázových systémů značně liší, některé obsahují vlastní rozšíření nekompatibilní s normou SQL a v jiných není podpora vůbec žádná. Nejdříve se tedy podíváme, co o složených typech říká samotná SQL norma.

3.3.1 Složené typy podle normy SQL

První verze složených typů byla do normy zavedena až s verzí SQL:1999. Tím se tato metoda uložení liší od ostatních, pro jejichž použití postačuje starší a široce podporovaná verze SQL-92. To z části odporuje jednomu z cílů této práce, zabývat se především metodami, kterým stačí základní vlastnosti SQL a jsou díky tomu přenositelné a tedy použitelné na širokém spektru databázových systémů. Na druhou stranou je tato metoda zajímavým rozšířením předchozí. Co se týče dotazování, tak je elegantnější a přehlednější.

Složené typy v SQL:1999

Norma SQL:1999 zavádí pouze jediný složený typ `ARRAY` pro pole, tedy uspořádanou kolekci prvků jednoho typu. Pole jsou indexována od 1 a musí mít již v definici tabulky deklarovanou maximální možnou velikost. Pole mohou být pouze jednorozměrná, nelze tedy vytvořit pole prvků typu `ARRAY` nebo jiného složeného typu, který typ `ARRAY` obsahuje. V definici tabulky se pak klíčové slovo připojuje za základní typ sloupce spolu s maximální možnou velikostí.

```
pole_celých_čísel INTEGER ARRAY[10] NOT NULL
```

Pro účely níže popsané indexační metody potřebujeme zjišťovat přítomnost prvku v poli a získat jejich počet. Počet prvků pole vrací funkce `CARDINALITY`.

K testu na přítomnost prvku lze využít operátor `UNNEST`, který pole převádí na tabulku, kde každý prvek je reprezentován jednou řádkou. U pole základních typů bude mít výsledek odhnízdění obvykle jeden sloupec (pouze pokud bude prvek pole typu `ROW` může být sloupců i více). Pro zápis pole v dotazech se používá konstrukce `ARRAY(`*seznam prvků*`)`. Přistupovat lze pouze k jednotlivým prvkům pomocí indexu. Pokud bychom chtěli pracovat pouze s vybraným rozsahem, musíme použít `UNNEST` a vhodnou podmínkou prvky omezit. Díky možnosti převodu na běžnou relaci je možné prvky filtrovat pomocí `WHERE` a používat vestavěné funkce a predikáty. Z pole je tak možné vybrat nejen souvislý rozsah.

Složené typy v SQL:2003

Ve verzi SQL:2003 není již třeba deklarovat maximální velikost pole, které navíc může být i vícerozměrné. Novinkou je také druhý typ kolekce multimnožina, označená klíčovým slovem `MULTISET`. Ta se od pole liší především tím, že prvky nemají určené pořadí.

Práce s multimnožinou je obdobná jako s polem, navíc je možné použít další predikáty a operace. Pro nás je významný predikát `MEMBER OF` (lze vynechat `OF`) pro zjištění přítomnosti prvku v multimnožině. Tento predikát nelze aplikovat na pole ani v SQL:2003. Na multimnožinu lze také aplikovat množinové operace `UNION`, `INTERSECT` a `EXCEPT`. Obdobně jako u pole se pro zápis kolekce v dotazech používá konstrukce `MULTISET(`*seznam prvků*`)`.

Využití pro uložení cesty ke kořeni

Uzly na cestě ke kořeni leží v určitém pořadí, takže se nabízí spíše použití typu `ARRAY`. Typ `MULTISET` může obsahovat i data typu `ROW`, tedy hodnotu

složenou z více sloupců stejně jako řádka tabulky. Tímto způsobem lze použít i multimnožinu a ukládat řádku složenou z pořadí a samotné hodnoty uzlu. V SQL:2003 by definice sloupce pak vypadala takto:

```
cesta ROW(INTEGER, INTEGER) MULTISET
```

3.3.2 Složené typy v PostgreSQL

Norma a skutečná podpora jsou v praxi dvě odlišné věci. Nyní rozebereme jak vypadá situace v PostgreSQL. Současná implementace[8] obsahuje některé rysy obou standardů SQL:1999 i SQL:2003, ale rozhodně je neimplementuje plně. Navíc obsahuje množství vlastních rozšiřujících vlastností.

Je možné použít pouze jediný složený typ `ARRAY`. V podstatě je shodný s typem `ARRAY` podle normy SQL:2003. Tedy vícerozměrné pole, indexované od 1 s neomezenou velikostí. PostgreSQL ovšem používá vlastní syntaxi pro definici takového sloupce. Zápis vychází ze syntaxe programovacích jazyků, kdy za typ prvku je připojeno `[]`, případně je možné určit maximální velikost. Příklady použití:

```
pole VARCHAR(5)[]
pole_s_omezenou_velikostí INTEGER[10]
vícerozměrné_pole INTEGER[][]
```

Konstrukci podle normy je také možné použít, ale má zásadní nevýhodu, a to nutnost zadat pevnou maximální velikost pole. Pro zápis pole lze kromě standardní konstrukce `ARRAY(`*seznam prvků*`)` využít i zkrácenou variantu `{`*seznam prvků*`}`. Tuto formu také databáze používá pro zobrazování hodnot sloupců.

Přistupovat lze standardně k jednotlivým prvkům nebo uvést místo indexu rozsah hodnot, potom výsledkem výrazu je zase pole. Výběr rozsahu z pole zapisujeme jako *pole*`[`*dolní_mez*`:`*horní_mez*`]`.

Funkce a operátory

Funkcí pro práci s poli existuje několik, nás však zajímá především `array_upper`, která vrací horní mez jedné dimenze pole a lze ji tak použít jako náhradu za neexistující standardní funkci `CARDINALITY`. Jelikož naše pole má pouze jednu dimenzi a je indexované od 1, mají obě funkce v tomto případě stejný význam.

[8] verze PostgreSQL 8.2

Hledání prvků v poli je přímočaré. PostgreSQL poskytuje operátory `ANY` a `ALL` pro test, zda-li nějaký nebo všechny prvky pole odpovídají zadanému. Syntaxe je následující:

```
prvek operátor ANY/ALL (pole)
```

Lze tedy psát například

```
5 = ANY(pole)
4 > ALL(pole)
```

PostgreSQL poskytuje pro práci s poli další funkce a jazykové konstrukce, které lze nalézt v dokumentaci. Pro účely zde popisované indexační metody však nejsou potřebné.

Indexy

Na sloupci typu `ARRAY` lze vytvořit index, který umožňuje efektivně provádět porovnávání a řazení polí, ale vyhledávání konkrétního prvku v poli nijak neusnadní[9]. Výsledkem porovnání dvou polí je výsledek porovnání prvků na nejnižším indexu, kde se od sebe obě pole liší.

3.3.3 Struktura tabulky

V PostgreSQL bude tabulka se složeným typem vypadat následovně:

```
CREATE TABLE cesta_ke_kořeni_jako_pole (
  id INTEGER NOT NULL PRIMARY KEY,
  nadřazený INTEGER,
  cesta INTEGER[],
  název VARCHAR(100)
);
```

Aby nebylo nutné zadávat maximální velikost pole, použijeme v definici formát odlišný od normy preferovaný v PostgreSQL. Indexy zůstanou stejné jako v předchozí metodě:

```
CREATE INDEX idx_cesta_ke_kořeni_jako_pole_nadřazený
  ON cesta_ke_kořeni(nadřazený);
CREATE INDEX idx_cesta_ke_kořeni_jako_pole_cesta
  ON cesta_ke_kořeni(cesta);
```

[9]Index nad sloupcem typu pole je běžný index implementovaný pomocí B-stromu.

Textový výpis dat vypadá také obdobně, ale vnitřní struktura dat je diametrálně odlišná.

id	nadřazený	cesta	název
1		{}	savci
2	1	{1}	chobotnatci
3	2	{1,2}	slon africký
4	2	{1,2}	slon indický
5	1	{1}	kytovci
6	5	{1,5}	delfín
7	5	{1,5}	velryba
8	5	{1,5}	vorvaň

3.3.4 Dotazování

Místo textových porovnání budou dotazy využívat operátor `ANY` pro test přítomnosti prvku v poli.

Podřízené uzly

Dotaz na uzly podstromu, procházené do hloubky

```
SELECT * FROM cesta_ke_kořeni_jako_pole WHERE $id = ANY(cesta)
  ORDER BY cesta
```

Stejně jako v původní metodě lze procházet uzly i do šířky a použít k tomu stejný postup, při kterém výsledek necháme seřadit podle počtu prvků pole.

```
SELECT * FROM cesta_ke_kořeni_jako_pole WHERE $id = ANY(cesta)
  ORDER BY COALESCE(array_upper(cesta,1),0)
```

Druhý parametr funkce `array_upper` určuje dimenzi pole, jejíž velikost má být vrácena. U našeho jednorozměrného pole to bude vždy 1. V PostgreSQL vrací funkce `array_upper` pro prázdné pole `NULL` a ne 0. Pro korektní seřazení je proto třeba na výsledek `array_upper` aplikovat ještě funkci `COALESCE`.

Dotaz pro získání podstromu lze zapsat i jiným způsobem pomocí logického operátoru pro pole `<@`, jehož výsledek je pravdivý pokud prvky prvního operandu jsou obsaženy v druhém operandu. Využijeme opět toho, že prvky pole *cesta* v podstromu mají společnou počáteční sekvenci prvků, která se u jiných uzlů nemůže vyskytovat.

```
SELECT * FROM cesta_ke_kořeni_jako_pole
  WHERE array_append($cesta,$id) <@ cesta
```

Obě varianty dotazu na podstrom obsahují elegantní zápis podmínky, bohužel však nejsou optimální. V obou případech je při vyhodnocování dotazu nutný sekvenční průchod celou tabulkou, protože není využito toho, že hledáme pouze v prefixu celého pole. Abychom získali použitelnou alternativu původní metody, je potřeba vyhledávat v indexu. Toho lze dosáhnout použitím operace porovnání polí. Efektivní dotaz vypadá takto:

```
SELECT * FROM cesta_ke_kořeni_jako_pole WHERE cesta
  BETWEEN array_append($cesta,$id)
  AND array_append(array_append($cesta,$id),$MAX_INT)
```

Proměnná `MAX_INT` reprezentuje číslo větší než libovolný identifikátor. Horní mez `BETWEEN` jinými slovy říká, že chceme všechny uzly s prefixem menším než `cesta || (id+1)`. Jelikož `BETWEEN` zahrnuje i uzly, u kterých je hodnota rovná mezím (což je výhodné u dolní meze), tak je nutné podmínku zapsat ve formě uvedené v dotazu. Řazení výsledku je možné ovlivnit stejně jako v předešlých neefektivních variantách.

Při praktickém použití je nejvhodnější obě meze předpřipravit mimo databázi a do dotazu dosadit jen dvě hodnoty. Spojování polí pomocí `array_append` provedení dotazu zbytečně prodlužuje. Zde se vyskytuje pouze kvůli tomu, aby mohl být dotaz zapsán pomocí proměnných odkazujících se na hodnoty sloupců.

Hloubka uzlů

Zjištění hloubky uzlu je už podstatně přehlednější než u původní metody. Hloubka odpovídá počtu nadřazených uzlů, tedy také počtu prvků pole *cesta*.

```
SELECT id, název, COALESCE(array_upper(cesta,1),0)
  AS hloubka_uzlu FROM cesta_ke_kořeni_jako_pole
```

Aby byl výsledek dotazu shodný s původní metodou, použijeme funkci `COALESCE` stejně jako při řazení.

Nadřazené uzly

Dotaz na nadřazené uzly je přímočarý.

```
SELECT * FROM cesta_ke_kořeni_jako_pole WHERE id = ANY($cesta)
  ORDER BY COALESCE(array_upper(cesta,1),0) DESC
```

Identifikátor společného nadřízeného dvou uzlů musí být obsažen v obou cestách. Chceme nejhlubší takový uzel, takže musíme řadit dle hloubky a vrátit první řádek výsledku.

```
SELECT * FROM cesta_ke_kořeni_jako_pole
  WHERE id = ANY($A.cesta) AND id = ANY($B.cesta)
  ORDER BY array_upper(cesta,1) DESC LIMIT 1
```

3.3.5 Aktualizace hierarchie

Aktualizacemi se tato rozšířená metoda, co týče ovlivněných řádků, neliší od původní. Stejně tak při vkládání listu dostaneme hodnotu sloupce *cesta* přidáním *id* nadřazeného záznamu na konec pole s cestou ke kořeni nadřazeného záznamu. K tomu lze využít operátor `||` pro spojení dvou polí nebo připojení skalární hodnoty za poslední prvek. Stejný efekt má také funkce `array_append`.

Úprava vnitřního uzlu

Složitější situace nastává v případě změny hodnoty *cesta* v celém podstromu. V původní metodě jsme mohli výhodně využít textové nahrazení části řetězce. V této metodě chceme nahradit počáteční část pole za jinou. V případě přídání vnitřního uzlu musíme provést jen vložení prvku do vnitřku pole. Při mazání vnitřního uzlu pak potřebujeme naopak vnitřní prvek pole vypustit. Obě operace stále pokrývá obecný případ nahrazení počáteční sekvence jinou. Konkrétní provedení opět značně závisí na databázovém systému, v PostgreSQL to lze provést dvěma způsoby. V jiném systému může být dostupný pouze jeden z nich nebo vlastní úplně odlišný způsob.

První způsob vypadá takto:

```
UPDATE cesta_ke_kořeni SET cesta =
  array_cat($cesta,cesta[${DÉLKA_PREFIXU+1} : $MAX_INT ]
  WHERE $id = ANY(cesta)
```

Dotaz vypadá komplikovaně, ale je to především kvůli použití proměnných, které budou ve skutečnosti nahrazeny konkrétními čísly. Celý je vztažen k uzlu, jehož podstrom je ovlivněn změnou. To je buď nově přidaný vnitřní uzel nebo nadřízený smazaného uzlu, případně kořen přesouvaného podstromu. Potom nová *cesta* všech aktualizovaných uzlů vznikne spojením nového prefixu, který přesně odpovídá hodnotě sloupce *cesta* kořene podstromu a neměněné části, která reprezentuje cestu v rámci podstromu. Proměnná `$DELKA_PREFIXU+1` značí index, na kterém začíná nezměněný zbytek sekvence a `$MAX_INT` je pak libovolné číslo větší než délka každého pole v tabulce. Tento technický trik musíme použít, protože nevíme jak je nejdelší pole dlouhé a pro každou řádku chceme získat zbytek pole od známého indexu až po poslední prvek. Pokud použijeme v rozsahu číslo větší než je délka pole,

databázi to nijak nevadí a použije jednoduše sekvenci až do jeho konce tak, jak potřebujeme.

Druhým způsobem jak zařídit aktualizaci, je převést problém na textovou náhradu a použít dotaz uvedený u původní metody. Využít můžeme párové metody `array_to_string` a `sring_to_array` pro převod pole na řetězec a zpět. Tím dostaneme řetězec, kde jsou jednotlivá *id* oddělena vybraným znakem nebo řetězcem, což je přesně předchozí metoda ve variantě s oddělovači.

3.4 Tabulka vazeb

Jak už název napovídá, tato metoda, stručně popsána v článku [3], nepřidává žádné pomocné atributy do původní tabulky uzlů, ale používá druhou pomocnou tabulku pro uložení vzájemných vazeb v hierarchii. Ta obsahuje všechny hierarchické vztahy nadřazený a podřízený uzel. A to nejen vztahy přímé nadřazenosti, ale i nepřímou nadřazenost přes libovolný počet dalších uzlů[10]. Kromě samotné dvojice uzlů se v každé řádce tabulky zaznamenává ještě délka cesty mezi oběma uzly, která slouží především ke snadnému řazení výsledku.

Vzhledem k tomu, že každý uzel má nad sebou průměrně $O(\log n)$ uzlů, velikost vazební tabulky bude $O(n \cdot \log n)$. Odkaz na nadřazený uzel je v hlavní tabulce redundantní, stejnou informaci obsahuje i tabulka vazeb, stačí nalézt přímého nadřízeného se vzdáleností jedna. I kdybychom nezavedli konvenci, že každá tabulka bude obsahovat všechny sloupce triviálního uložení, stejně by se redundantní sloupec vyplatil kvůli výkonnosti. Hlavní tabulka obsahuje podstatně méně záznamů a informace o přímo nadřazeném je v praxi často využívaná.

3.4.1 Struktura tabulek

Základní tabulka je stejná jako u triviální metody. Vazební tabulka vypadá následovně:

```
CREATE TABLE vazby (
  nadřazený INTEGER NOT NULL REFERENCES triviální(id),
  podřízený INTEGER NOT NULL REFERENCES triviální(id),
  vzdálenost INTEGER NOT NULL,
  PRIMARY KEY (nadřazený,podřízený)
);
```

[10] Jsou zde všechny dvojice které jsou v relaci uspořádání příslušné k dané hierarchii.

Dotazování nad tabulkou vazeb bude především vyhledávat hodnoty ve sloupcích *nadřazený* a *podřízený*. K vyhledávání podle nadřazených uzlů poslouží implicitní index vytvořený k primárnímu klíči. Hodnota podřízeného uzlu je v tomto složeném indexu až na druhém místě takže k filtrování pouze této hodnoty nelze použít. Proto vytvoříme ještě samostatný index:

```
CREATE INDEX idx_vazby_podřízený
  ON vazby(podřízený)
```

Obsah tabulky vazeb pro ukázková data vypadá takto:

nadřazený	**podřízený**	**vzdálenost**
1	2	1
1	5	1
1	3	2
1	4	2
1	6	2
1	7	2
1	8	2
2	3	1
2	4	1
5	6	1
5	7	1
5	8	1

Ohodnocení hran

Výhodou vlastní tabulky pro vazby je možnost jejího rozšíření o další sloupce. V teorii grafů bychom takové atributy nazvali ohodnocením hrany. U ostatních metod, kde neexistují řádky odpovídající přímo hranám, lze ohodnocení hrany uložit k podřízenému uzlu, protože každý uzel figuruje jako podřízeny pouze v jedné hraně. To je jistě také funkční řešení, ale je méně praktické a přehledné.

3.4.2 Dotazování

Nevýhodou metody, vyplývající z použití druhé tabulky, je nutnost operace spojení (`JOIN`) pro získání vlastních dat uzlů u všech dotazů používajících pomocnou tabulku. Místo spojení může dotaz obsahovat predikát `EXIST` apod., ale na složitosti dotazů a nutnosti procházet dvě tabulky to nic nemění. To je rozdíl od předchozích metod, které používaly tabulku jedinou a dotazy tak mohly být jednodušší. To se může projevit na efektivitě zejména pokud

je stromový dotaz pouze součástí složitějšího a výsledek je dále spojován s jinými tabulkami.

Pokud nám stačí samotné identifikátory uzlů, můžeme se omezit pouze na jednoduchý dotaz. V praxi jsou ovšem obvykle potřeba i další sloupce tabulky. Vlastní data uzlů se například prezentují uživateli nebo se řádky tabulky mapují na objektové typy jiného programovacího jazyka v obslužné aplikaci. Dotazy na celé řádky budou navíc konzistentní s dotazy uvedenými u předchozích metod. Také během měření efektivity v kapitole 4 budou použity rozšířené dotazy se všemi sloupci, aby bylo porovnání relevantní.

Nadřazené uzly

Pro získání cesty ke kořeni využijeme toho, že všechny nadřazené uzly jsou v tabulce vazeb. Nadřazenější uzly jsou více vzdálené, proto seřadíme výsledek sestupně podle vzdálenosti.

```
SELECT uzel.* FROM triviální AS uzel
  JOIN vazby AS v ON (uzel.id = v.nadřazený)
  WHERE v.podřízený = $id ORDER BY vzdálenost DESC
```

Podřízené uzly

Získání celého podstromu je obdobná situace jako získání cesty ke kořeni. Pouze v dotazu vyměníme sloupec *podřízený* za *nadřazený*. Opět řadíme podle vzdálenosti, takže výsledek odpovídá průchodu do šířky.

```
SELECT uzel.* FROM triviální AS uzel
  JOIN vazby AS v ON (uzel.id = v.podřízený)
  WHERE v.nadřazený = $id ORDER BY vzdálenost
```

Společný předek

V tabulce vazeb hledáme nejhlubší společný nadřazený záznam. Dotaz pro získání společného předka dvou uzlů:

```
SELECT * FROM triviální WHERE id = (
  SELECT parent FROM vazby AS v1
  JOIN vazby AS v2 ON (v1.nadřazený = v2.nadřazený)
  WHERE v1.podřízený = ${A.id} AND v2.podřízený = ${B.id}
  ORDER BY vzdálenost ASC LIMIT 1
)
```

Výsledek dotazu je vždy uzel odlišný od obou zadaných, i když je jeden z nich nadřazený druhému. V takové situaci bychom někdy chtěli už samotný nadřazený uzel. Nejjednodušší je toto vyřešit samostatným dotazem.

Teoreticky je možné pro získání společného předka využít dotazu na cestu ke kořeni. Ten aplikovat na oba uzly samostatně, na výsledcích provést průnik a z něj vrátit nejhlubší uzel. Problém je se získáním nejhlubšího uzlu z průniku. Ve výsledku množinové operace `INTERSECT` jsou pouze řádky shodné ve všech sloupcích. Dva uzly pro které hledáme předka ale obecně neleží ve stejné hladině a jsou od společného předka různě vzdálené. Proto musíme průnik hledat na výsledku bez sloupce *vzdálenost*. K průniku lze pak operací spojení opět jednu z hloubek pro účely seřazení dodat, ale dotaz se tak zbytečně komplikuje. Původní varianta je přehlednější a efektivnější.

Hloubka uzlů

Sloupec *vzdálenost* lze kromě řazení použít ještě k nalezení hloubky podstromu. Jednoduše je to maximum přes všechny vazby k podřízeným dotazovaného uzlu. Hloubka celého stromu je pak zřejmě hloubka podstromu kořene.

```
SELECT MAX(vzdálenost) FROM vazby WHERE vazby.nadřazený = $id
```

Z tabulky vazeb lze také ověřit, zda-li je uzel listem. Nesmí pro něj existovat záznam ve kterém figuruje jako *nadřazený*. Vzhledem k tomu, že obdobný dotaz lze provést i nad samotnou tabulkou uzlů, která obsahuje mnohem menší počet řádek, postrádá takový dotaz většinou smysl.

3.4.3 Aktualizace hierarchie

Po přidání nového listu je třeba tabulku vazeb aktualizovat vložením jednoho řádku pro každý nadřazený uzel. To se dá snadno zařídit pomocí konstrukce `INSERT SELECT`. Nový list má vazby ke stejným uzlům jako jeho otec, pouze vzdálenost je o jedna větší. Nakonec musíme přídat ještě úplně novou vazbu se vzdáleností jedna mezi listem a jeho otcem.

Pokud vkládáme nelistový uzel, musíme navíc vytvořit i vazby k všem uzlům ležícím v podstromu. Na rozdíl od jiných metod, kde vložení vnitřního listu způsobuje také aktualizaci pomocných hodnot v celém podstromu, v této metodě zůstávají vlastní záznamy nezměněné a úprava probíhá jen na tabulce vazeb. To je o trochu lepší situace, protože hlavní tabulka je vždy k dispozici pro čtení a též pro modifikace vlastních dat uzlu[11].

Mazání uzlu je jednoduché, odstraníme všechny vazby ve kterých figuruje.

[11] To znamená úprava dat, kdy se nemění hierarchie.

```
DELETE FROM vazby WHERE vazby.nadřazený = $id
  OR vazby.podřízený = $id
```

Všechny dotazy, které udržují tabulku vazeb konzistentní se dají snadno implementovat jako trigger.

3.5 Vnořené množiny

Tuto oblíbenou metodu pro ukládání stromů lze najít pod různými názvy. Kromě vnořených množin se lze setkat například s vnořenými intervaly, zejména pak u zobecněné varianty. I v českých textech se často používá původní nepřeložený název *Nested Sets*[12].

Princip metody spočívá v přiřazení dvou čísel každému záznamu při procházení stromu do hloubky. Hodnoty vyjadřují pořadí v jakém do uzlu vstupujeme, respektive se přes něj vracíme během procházení do hloubky. Sloupce tabulky s těmito hodnotami pojmenujeme:

- *levá* – pro vstupní hodnotu,
- *pravá* – pro výstupní hodnotu.

Názvy vychází z grafického znázornění (viz. obr 3.3) a v dotazech jsou tyto názvy pro čestinu přehlednější než názvy *vstupní* a *výstupní*, které na první pohled vypadají stejně[13]. Konkrétní hodnoty sloupců pro existující strom získáme tak, že na počátku nastavíme čítač na 1 a procházíme strom od kořene do hloubky. Při vstupu do uzlu přiřadíme hodnotu čítače do *levá* a k čítači příčteme jedničku. Po zpracování všech podřízených uzlů a opouštění uzlu přířadíme hodnotu čítače do *pravá*.

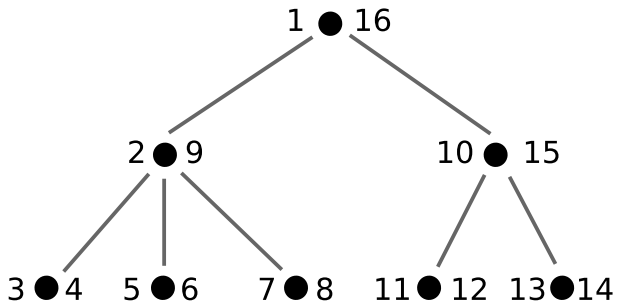
Pro souvislou hierarchii platí, že *levá* hodnota kořene je rovná jedné a *pravá* hodnota kořene je rovná dvojnásobku počtu uzlů stromu.

Znázornění pomocí intervalů

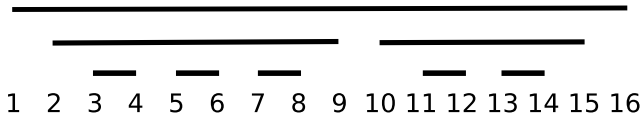
Název vnořené množiny, případně vnořené intervaly, pochází z jiného pohledu na očíslování. Každému uzlu přísluší jeden celočíselný interval, kde hodnoty *levá* a *pravá* jsou jeho meze. Listy odpovídají nejkratším intervalům délky 1. Interval podřízeného je vždy vnořen do intervalů svých nadřízených. Všechny ostatní intervaly jsou tak vnořeny do nejdelšího intervalu odpovídajícímu kořeni stromu. Očíslovaný strom z minulého příkladu (obr. 3.3) je znázorněn pomocí intervalů na obrázku 3.4.

[12]Další anglický název je *Modified Preorder Tree Traversal*.

[13]V anglických názvech je i označení *in* a *out* přehledné, takže je možné potkat obě varianty.



Obrázek 3.3: Očíslování stromu



Obrázek 3.4: Vnořené intervaly (množiny)

Vlastnosti očíslování

Můžeme si všimnout několika vlastností očíslování, které využijeme při dotazování:

- Všechny podřízené uzly mají levou hodnotu větší než otec (nebo libovolný další nadřízený) a pravou hodnotu naopak menší.
- Pro listy platí *levá* + 1 = *pravá*.
- Všechny hodnoty *levá* a *pravá* jsou navzájem[14] unikátní.

Uspořádání sourozenců

Oproti předchozím metodám mají vnořené množiny jednu velice užitečnou vlastnost. Pomocné hodnoty, nejen že určují stromovou strukturu, ale také přesně určují vzájemné pořadí sourozenců[15].

3.5.1 Struktura tabulky

```
CREATE TABLE vnořené_množiny (
```

[14] Žádná *levá* hodnota uzlu se nerovná *levé* ani *pravé* hodnotě jiného uzlu. To samé platí i pro *pravou* hodnotu.

[15] To lze u ostatních metod zajistit přidáním jednoho sloupce, ale tady je pořadí dáno implicitně.

```
  id INTEGER NOT NULL PRIMARY KEY,
  nadřazený INTEGER NULL REFERENCES vnořené_množiny(id),
  levá INTEGER NOT NULL,
  pravá INTEGER NOT NULL,
  název VARCHAR(100)
);
```

Pomocné sloupce *levá* a *pravá* by mohly vybízet k vytvoření unikátního indexu, který ovšem není vhodný. Takový index stejně nezaručí unikátní hodnoty mezi oběma sloupci, způsobí zpomalení aktualizací a především způsobí problém při hromadné změně hodnoty.

Příkladem může být přidání vnitřního uzlu, popsané dále v odstavci 3.5.3, při kterém se může část pravých hodnot zvětšit o 2. Některé z těchto hodnot se ve stromě mohou lišit právě o inkrement 2. Databáze provádí úpravy na řádkách postupně a dočasně tak vznikne duplicitní hodnota v rámci jednoho `UPDATE` příkazu. Úprava tak při existenci unikátního indexu selže, i když by po její dokončení bylo vše v pořádku.

Na pomocných hodnotách je nejvýhodnější složený index nad oběma hodnotami, nejprve řazený podle levé hodnoty. Dále pak tradiční index na sloupci *nadřazený*

```
CREATE INDEX idx_vnořené_množiny_levá_pravá
  ON vnořené_množiny(levá, pravá);
CREATE INDEX idx_vnořené_množiny_nadřazený
  ON vnořené_množiny(nadřazený);
```

Pro úplnost je ještě uveden textový výpis ukázkových dat pro vnořené množiny.

id	nadřazený	levá	pravá	název
1		1	16	savci
2	1	2	7	chobotnatci
3	2	3	4	slon africký
4	2	5	6	slon indický
5	1	8	15	kytovci
6	5	9	10	delfín
7	5	11	12	velryba
8	5	13	14	vorvaň

3.5.2 Dotazování

Invarianty pro levé a pravé hodnoty umožňují jednoduché konstrukce pro většinu obvyklých dotazů. Využije se zejména „vnořenosti“ podřízených.

Nadřazené uzly

```
SELECT * FROM vnořené_množiny WHERE
  levá < $levá AND pravá > $pravá ORDER BY levá
```

K řazení lze v principu použít sloupec *levá* i *pravá*, ale vzhledem k vytvořeným indexům je vhodnější řadit podle levé hodnoty.

Podřízené uzly

Otočením nerovností dostaneme dotaz pro získání uzlů v podstromu. Místo obalujících intervalů chceme v tomto případě vnořené.

```
SELECT * FROM vnořené_množiny WHERE
  levá > $levá AND pravá < $pravá ORDER BY levá
```

Takový dotaz ale není optimální. Levé hodnoty v podstromu jsou také omezené pravou hodnotou nadřízených. Stejný výsledek tak můžeme dostat pouze podmínkou pro levou hodnotu.

```
SELECT * FROM vnořené_množiny WHERE
  levá > $levá AND levá < $pravá ORDER BY levá
```

Obě dvě varianty použijí filtrování podle indexu. V první variantě se ale na hodnotách, které splňují podmínku pro levou hodnotu musí kontrolovat platnost podmínky pro pravou hodnotu v druhé úrovni indexu. Takových řádek je mnohem více než jich bude ve výsledku. Kdežto druhá varianta, která používá pouze filtrování na levé hodnotě, může obě podmínky splnit vybráním příslušného rozsahu v první úrovni indexu. Není potřeba testovat žádné řádky navíc. Dopad na čas provedení dotazu je zásadní.

O kolik je druhý dotaz rychlejší nelze obecně určit. Závisí to na množství dat, tvaru celého stromu i dotazovaného podstromu. Obecně čím rozsáhlejších data a čím více falešných zásahů vybere první podmínka horšího dotazu, tím bude rozdíl v efektivitě větší. Lze si všimnout, že falešnými zásahy jsou všechny uzly na pravé straně od podstromu, který je výsledkem. Proto první dotaz se chová relativně dobře pro vysoké pomocné hodnoty, kdežto selhává při dotazech nad uzly s nízkými hodnotami (tj. když ptáme se na podstrom v levé části stromu).

Pro ilustraci můžeme použít sadu dat NCBI z kapitoly 4, na které je druhý dotaz průměrně 90× rychlejší[16]. Už vznik očíslování pomocných hodnot průchodem stromu do hloubky naznačuje, že jediné možné řazení výsledku bude stejné jako při tomto průchodu.

[16]Měření bylo provedeno stejně jako praktické testy v kapitole 4, která také pojednává o metodice testování.

Nahrazením ostrých nerovností za neostré snadno přidáme do výsledku i uzel na jehož nadřazené nebo podřízené se ptáme, což bývá někdy užitečné. V dotazu s neostrými nerovnostmi by bylo hezčí využít operátor `BETWEEN`, který lze samozřejmě využít i v uvedených dotazech s ostrými nerovnostmi, ale je pak potřeba hodnoty upravit o 1.

Společný předek

Zjištění společného předka zajistí také jednoduchý dotaz. Stačí si uvědomit, že interval předka musí obalovat intervaly obou podřízených uzlů. To znamená, že levá hodnota je menší než obě levé hodnoty a pravá je vetší než obě pravé.

```
SELECT * FROM vnořené_množiny WHERE
  levá < ${A.levá} AND levá < ${B.levá}
  AND pravá > ${A.pravá} AND pravá > ${B.pravá}
```

Alternativně lze dotaz zapsat s pomocí funkcí `LEAST` a `GREATEST`

```
SELECT * FROM vnořené_množiny WHERE
  levá < LEAST(${A.levá},${B.levá})
  AND pravá > GREATEST(${A.pravá},${B.pravá})
```

Další dotazy

Rozdíl mezi levou a pravou hodnotou uzlu snížený o jedna je dvojnásobkem počtu uzlů v jeho podstromu. Taková informace sice není od databáze často požadována, ale lze ji získat pouze z dat jediného uzlu bez dalšího dotazování.

Jelikož se u listů liší levá a pravá hodnota o jedničku, lze z tabulky touto podmínkou vybrat všechny listy.

```
SELECT * FROM vnořené_množiny WHERE levá + 1 = pravá
```

Pro takový dotaz nelze použít index[17], takže provedení vyžaduje sekvenční procházení všech řádek, ale i tak je efektvnější než dotaz pro hledání listů uvedený u triviálního uložení.

3.5.3 Aktualizace dat

Aktualizace struktury je největší nevýhodou jinak rychlé metody. Očíslování každého uzlu závisí na struktuře celé hierarchie, tudíž i nejjednodušší možná úprava přidání jediného listu způsobí nutnost přečíslovat velké množství záznamů.

[17]Můžeme samozřejmě vytvořit speciální index na výrazu *pravá - levá* jen pro tento jediný dotaz.

Přidání uzlu

Pro přidání uzlu musíme v očíslování nejdříve vyrobit „mezeru“ pro nový uzel. To znamená zvětšit o 2 všechny levé a pravé hodnoty větší než je hodnota v místě vložení. V nejhorším případě tak musíme zaktualizovat všechny uzly, a to když vkládáme list na pozici ve stromu nejvíce vlevo.

Pokud bychom netrvali na tom, aby byl kořen číslován od 1, lze místo přičtení 2 k větším hodnotám odečíst 2 od všech menších hodnot. Potom lze vybrat pro úpravu menší část stromu a nejhorší případ nastane pokud jsou obě části přibližně stejné. Úprava tak modifikuje až polovinu řádek tabulky. Levá hodnota v kořeni odlišná od 1 nám chování a vlastnosti metody nijak nezhorší.

Bohužel ani tato drobná úprava aktualizacím příliš nepomůže. Můžeme si všimnout, že při přičtení 2 jsou nutně modifikována data uzlů ležící nejvíce napravo v každé hladině. Obdobně při odečtení 2 jsou upraveny všechny uzly na levé straně stromu. Speciálně pak kořen, který je ve své hladině sám a je tak zároveň nejlevějším i nejpravějším uzlem hladiny, bude upravován pokaždé. Při libovolné aktualizaci hierarchie to pak znamená zamčení záznamu pro zápis. Úpravy tak nemohou nikdy probíhat paralelně. Krom toho není během aktualizací možné provádět dotazy jejichž výsledek obsahuje kořen, což jsou například časté dotazy na zařazení v hierarchii respektive na cestu ke kořeni.

Popsaný problém se netýká jen samotného kořene, nejvíce pravé a nejvíce levé uzly v hladinách mají pravděpodobnost úpravy 1/2 a i mnoho dalších vnitřních uzlů může být zamknuto pro zápis. Výsledkem je nepoužitelnost této metody pro databáze které se často mění. Nepoužitelná je ovšem také pro kolekce které se mění jen příležitostně, ale jsou stále čteny. Vytížená databáze nemůže pozdržet čtení a čekat na uvolnění zámků pro zápis. Taková situace může vést až zahlcení databázového systému. Konkrétní chování bude záviset na způsobu zamykání v databázovém systému a izolační úrovni transakcí. Obecně lze počítat s tím, že se tato medoda bude během aktualizace jistě chovat hůře než všechny předchozí.

Ostatní aktualizace

Další aktualizace jsou pak stejně složité jako přídání uzlu, které způsobuje globální změnu. V podstatě při každé úpravě je potřeba v očíslování udělat místo na nový interval a nebo naopak vzniklou mezeru zrušit. Provedení a z toho plynoucí následky jsou proto úplně stejné jako v situaci popsané výše.

Nesouvislé hierarchie

V případě nesouvislé hierarchie číslujeme jednotlivé stromy lesa postupně. Očíslování následujícího pokračuje tam, kde skončilo u předchozího stromu.

Pro nesouvislé hierarchie a předchozí metody uložení platilo, že modifikace jednoho stromu nijak neovlivňovala ostatní stromy. To je pěkná vlastnost, která u této metody neplatí. Důvod je zřejmý, předchozí metody při aktualizaci struktury upravovaly pouze záznamy v podstromech, kdežto tato metoda upravuje část uzlů globálně.

Pokud je pro reálné použití taková vlastnost nevhodná, lze zavést malou modifikaci pomocných dat. Každý strom v hierarchii číslujeme levými a pravými hodnotami zvlášť a do tabulky přidáme sloupec navíc, kterým rozlišíme jakého stromu se číslování týká. Dotazy pak rozšíříme o podmínku, která omezí výsledek na konkrétní strom (případně stromy).

3.5.4 Vylepšení metody pro aktualizace

Joe Celko ve své publikaci [1] nabízí několik způsobů, jak špatné chování při aktualizacích vylepšit.

Vnořené množiny s mezerami

První cestou ke zlepšení je zachovat pravidlo pro vnoření intervalu (množin), ale nepožadovat těsné vnoření. Mezi pravou a levou hodnotou sousedních uzlů tak může být mezera větší než 1. Obdobně může být větší interval mezi hodnotami otce a jeho synů. Autor takto rozšířenou metodu místo vnořených množin označuje jako vnořené intervaly. Mezery umožňují nechat si určitou rezervu pro vkládání dalších uzlů bez nutnosti měnit pomocné hodnoty dalších záznamů. Abychom měli rezervu co největší, tak kořen stromu obsadí celý interval možných hodnot (tj. levá hodnota je rovná minimu a pravá maximu integerové hodnoty).

Distribuce rodičovského intervalu

Už při přidávání prvního podřízeného uzlu narazíme na problém, protože nevíme jak velkou část intervalu rodiče máme synovi přiřadit. Kořenový interval musíme rozdělit na dvě části, jednu rezervovat pro podřízené nového uzlu a zbytek nechat pro další sourozence a jejich podstromy. Můžeme například vyhradit vždy 1/8 zbývajícího rodičovského intervalu pro nový uzel. Zbylý interval hodnot je pak k dispozici později přidaným sourozencům (každý takový si také vezme osminu zbytku).

Kapacita upravené metody

Zvolený poměr má vliv na to, jaký tvar stromu bude tabulka schopná pojmout. V našem případě (poměr 1 : 8) pojme strom maximálně uzel v 10. hladině, pokud je celočíselný typ pro pomocné hodnoty 32-bitový [18]. Nemůže to však být uzel umístěný ve stromě libovolně. Toto platí pouze pro cestu, která obsahuje jen syny přidané jako první. Pouze ti mají z původního rodičovského intervalu vyhrazenou celou 1/8. Další sourozenci „ukusují“ jen osminu ze zbytku, takže čím později vložený uzel, tím i méně prostoru pro vlastní podřízené. To je další zřejmá nevýhoda, metoda se nechová ke všem uzlům rovnocenně. Dříve přidaní synové mají k dispozici větší množství hodnot, i když ve skutečném stromě může být potřeba přidělovat rozsahy hodnot přesně opačná.

Obdobným způsobem mohou dojít hodnoty i pro sourozence. Pro 32-bitovou hodnotu lze přidat kořeni kolem 150 synů, kde navíc poslední nemají téměř žádný prostor pro vlastní podřízené. Pro nižší hladiny bude maximální množství synů ještě menší.

Podívejme se teď na charakteristiku dat NCBI (popsaných v 4.1.1):

Hloubka stromu	41
Max. počet synů	5 600

Je jasné, že podle výše popsaného systému se celá data stromu do tabulky uložit nedají. Pro zmíněná data ani jiné nastavení parametrů metody nepomůže. Nelze zabránit tomu, aby interval pro přidání byl příliš krátký. Budujeme-li strom prostupným vkládáním uzlů, v jednu chvíli se nutně stane, že rozdíl mezi levou a pravou hodnotou rodiče je menší než 3 a nový uzel není kam vložit[19]. Pokud taková situace nastane, lze ji stále ještě řešit obdobně jako v základní metodě posunutím pomocných hodnot již vložených řádek. Problém je, že v upravené metodě je tato operace komplikovanější. Volné místo není na krajích kořenového intervalu, ale je rozprostřené v celém stromě. Proto je nutné nejdříve najít vhodnou mezeru a z ní část přesunout na vkládané místo.

Uspořádání sourozenců

Ještě zmiňme jednu nevýhodu. V původním uložení pomocné hodnoty určovaly také vzájemné pořadí jednotlivých sourozenců. O tuto vlastnost přijdeme, protože volný interval pro přidávání nových uzlů je pouze za posledním synem. Nechávat mezery pro vkládání i jinde je nemožné, tím bychom

[18] S 64-bitovými celými čísly lze uložit uzel ve dvojnásobné, tj. 20. hladině.

[19] Potřebujeme vložit 2 nové hodnoty mezi meze rodičovského intervalu.

vyčerpali intervaly s rezervou pro další vkládání mnohem rychleji. Fakticky je pořadí určeno stejně jako v základní metodě, ale nelze si vybrat na kterou pozici nový uzel vložíme bez rozsáhlého přečíslování uzlů, kterému se snažíme vyhnout. Stejně tak přijdeme o pěkné vlastnosti vyplývající z neexistence mezer. V této metodě už nepoznáme listy ze samotné levé a pravé hodnoty. Ani neumíme jednoduše najít posledního syna podle pravé hodnoty, která byla v původní metodě o 1 menší než vlastní pravá hodnota apod.

Shrnutí varianty s mezerami

Výsledek našeho snažení není nijak uspokojivý, nevhodným úpravám velké části podstromu jsme nezabránili, jen jsme zmenšili jejich četnost za cenu ztráty jiných výhodných vlastností. Naopak pokud k takové operaci dojde, je její provedení náročnější a je potřeba netriviální algoritmus, který zajistí vhodnou distribuci volných intervalů. Bez něj by metoda byla pravděpodobně ještě horší než původní varianta. Jak by takový algoritmus ideálně vypadal není zřejmé a ani není předmětem této práce ho hledat.

Vnořené množiny s racionálními intervaly

Autor vnořených intervalů nabízí ještě jedno vylepšení původní metody. Jediný problém minulého vylepšení je další nedělitelnost intervalu v okamžiku, kdy má délku menší než 3. Problém bychom eliminovali, kdybychom mohli pro očíslování použili racionální nebo reálná čísla. Desetinné datové typy mají v SQL také omezenou přesnost, navíc potřebujeme hodnoty přesně porovnávat, takže přímé použití typu `DECIMAL` nic nevyřeší. V [1] je popsána pěkná teoretická konstrukce, jak pro racionální intervaly používat dvě celá čísla, s tím že lze snadno jejich hodnoty porovnávat, přepočítat na hodnoty nadřazených uzlů a provádět podobné operace, jako umožňují původní vnořené množiny. Navíc hodnoty pro jednou vložený uzel, respektive pozici ve stromě, se již nemění, stejně jako kdyby intervaly byly opravdu racionální.

Pro praktické použití ale opět platí to, co pro předchozí verzi. Přepočítaná celá čísla nejsou nijak omezena a pro větší stromy se také nevejdou do integerové hodnoty. V tomto případě ani není cesty jak situaci řešit, proto se touto metodu podrobně nebudeme zabývat vůbec.

Závěrem k vnořeným množinám, pro reálné použití je patrně nejvhodnější základní verze metody s vědomím jejich slabých stránek. Rozšíření přinášejí spíše více nevýhod než užitku, a tam kde není vhodná původní metoda bude pravděpodobně lepší pro ukládání stromů zvolit úplně jinou metodu .

3.6 Uložený průchod do hloubky

Stručný popis poslední metody lze najít ve článku [4], kde se používá pro ukládání malých stromů. Krátký a ustálený český název patrně neexistuje[20], takže budeme metodu označovat jako „uložený průchod do hloubky“ podle způsobu, jakým se konstruují pomocná data.

Strukturou pomocných dat je tato metoda podobná vnořeným množinám a je také založena na číslování uzlů při procházení stromu do hloubky. Místo číslování dvěma hodnotami budeme ukládat pouze jedno číslo určující pořadí vstupu do uzlu. Je to vlastně obdoba levé hodnoty z vnořených množin. Na rozdíl od vnořených množin, kde levé a pravé hodnoty jsou samy o sobě nic neříkající hodnotou, ukládá tato metoda do druhého pomocného sloupce hloubku uzlu. Ta je skutečným atributem a má smysl i mimo vlastní metodu. Může tak být dále využívána v jiných dotazech nebo prezentována přímo uživateli.

3.6.1 Struktura tabulky

```
CREATE TABLE uložený_průchod
  id INTEGER NOT NULL PRIMARY KEY,
  nadřazený INTEGER NULL REFERENCES uložený_průchod(id),
  pořadí INTEGER NOT NULL,
  hloubka INTEGER NOT NULL,
  název VARCHAR(100)
);
```

S indexem na sloupci *pořadí* a odkazem na *nadřazený* záznam.

```
CREATE INDEX idx_uložený_průchod_nadřazený
  ON uložený_průchod(nadřazený);
CREATE INDEX idx_uložený_průchod_pořadí
  ON uložený_průchod(pořadí);
```

Pokud bychom vyhledávali podle hloubky uzlů, mohl by být užitečný ještě samostatný index na sloupci *hloubka*. Pro dotazy uvedené v této práci však není potřeba.

V ukázkovém případě je *id* stejné jako *pořadí*. To obecně neplatí, *id* nemá k vlastním datům žádný vztah a záleží jen na tom jakém pořadí byly uzly do tabulky vloženy.

[20] V původním anglickém článku je metoda pojmenována jako Flat Table Model.

id	nadřazený	pořadí	hloubka	název
1		1	0	savci
2	1	2	1	chobotnatci
3	2	3	2	slon africký
4	2	4	2	slon indický
5	1	5	1	kytovci
6	5	6	2	delfín
7	5	7	2	velryba
8	5	8	2	vorvaň

3.6.2 Dotazování

Metoda je původně určená především pro ukládání malých stromů. Například pro stromové kategorie, kde si uživatel vybírá ze všech záznamů a je nutné celý obsah tabulky pouze vhodně naformátovat. K vytvoření takového výběru stačí jednoduchý dotaz na záznamy seřazené podle *pořadí*. V originálním článku se dokonce místo *hloubky* používá pro název sloupce *odsazení*, protože hodnota sloupce se primárně používá k tisku daného počtu mezer při výpisu tabulky. Takový výpis jsme už mohli vidět dříve v příkladu na obrázku 2.4.

U malých stromů je důležitá především jednoduchost použití a příliš nezáleží na rychlosti dotazů, protože provedení se liší jen o milisekundy. Nad malým počtem uzlů jsou rychlé v podstatě libovolné dotazy. Přestože je původní účel metody jiný, lze ji použít i pro ukládání rozsáhlých stromových struktur. Některé dotazy nejsou tak přímočaré, ale stále metoda disponuje několika dobrými vlastnostmi. Například máme stále výhodu snadného procházení do hloubky i do šířky. Metoda má stejnou vlastnost jako vnořené množiny, očíslování definuje pořadí uzlů ve stejné hladině, respektive pořadí synů.

Podřízené uzly

Můžeme si všimnout, že podřízené uzly mají pořadí větší než daný uzel a menší než jeho další sourozenec. Ještě zbývá vyřešit jeden technický problém. Co když je uzel posledním sourozencem? Pokud existuje uzel s větším pořadím ve stejné nebo menší hloubce, tak se nic neděje a díky omezení na hloubku dostaneme pořád správný výsledek. Pokud je však uzel posledním uzlem ve své hladině, vnořený `SELECT` vrátí `NULL`. V takovém případě hledáme záznamy s pořadím vyšším než zadaný uzel, takže stačí funkcí `COALESCE` nahradit `NULL` z poddotazu hodnotou vyšší než všechna pořadí. Zapsáno v SQL to vypadá takto:

```
SELECT * FROM uložený_průchod WHERE
```

```
  pořadí > $pořadí AND
  pořadí < COALESCE(
    (SELECT min(pořadí) FROM uložený_průchod WHERE
      hloubka <= $hloubka AND pořadí > $pořadí
    ),$MAX_INT)
ORDER BY pořadí
```

Pokud dotaz porovnáme s příslušným dotazem u vnořených množin, zjistíme že jsou si podobné. Pravou hodnotu z vnořených množin tu zastupuje pořadí následujícího sourozence. Na rozdíl od vnořených množin máme v tomto případě možnost procházet uzly i do šířky prostým řazením podle sloupce *hloubka*.

Nadřazené uzly

Uzly na cestě ke kořeni mají *pořadí* menší než vybraný uzel. Z takto vybraných uzlů leží na cestě ke kořeni ty uzly, které mají největší *pořadí* ve svojí hloubce.

```
SELECT cesta.* FROM uložený_průchod AS cesta NATURAL JOIN
  (SELECT max(pořadí) AS pořadí FROM uložený_průchod WHERE
    pořadí < $pořadí AND hloubka < $hloubka GROUP BY hloubka)
  AS hodnoty ORDER BY pořadí
```

Je vidět, že dotaz je relativně složitý a je nutné používat klauzuli `GROUP BY`. Dotazy na nadřazené uzly nejsou silnou stránkou této metody. Kdybychom neměli na tabulce složený index pro sloupce *hloubka* a *pořadí*, byl by pro provedení dotazu nutný sekvenční průchod tabulkou. Pro nalezení společného předka dvou uzlů nezbývá než průnik jejich cest ke kořeni, který je obecně použitelný pro všechny metody.

3.6.3 Aktualizace dat

Pro aktualizace platí téměř to samé co pro vnořené množiny. Číslování ve sloupci *pořadí* je globální pro celý strom a i jednoduchá změna struktury způsobí nutnost aktualizace velkého počtu záznamů. Stejně jako u vnořených množin je důležité pouze uspořádání na sloupci s pořadím, takže lze množství náročných aktualizací zmenšit očíslováním uzlů mezerami pro pozdější vkládání. Zamezit jim úplně však nelze.

3.7 Ukládání obecných hierarchií

Dosud jsme se zaměřovali pouze na ukládání stromů. Nyní probereme obecné hierarchie, kde může existovat více přímých nadřízených.

3.7.1 Obecné hierarchie s preferovaným rodičem

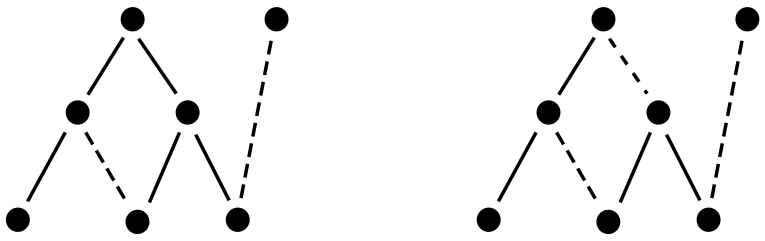
Nejprve se zaměříme na speciální případ obecných hierarchií, kdy pro každý uzel má jeden z rodičů výjimečné postavení. Takového rodiče nazveme hlavním, ostatní budeme nazývat vedlejšími rodiči (obdobně budeme mluvit o hlavních a vedlejších vazbách) Úvodní definice hierarchie s ničím takovým nepočítá. Formálně bychom ji tak měli rozšířit o hranovou funkci, která bude typ hrany rozlišovat. Takto definovanou hierarchii můžeme nazvat hierarchií s preferovaným rodičem. Sémantický význam je pak takový, že uzel je primárně podřízen hlavnímu rodiči a až v druhé řadě ostatním nadřízeným.

Příklady hierarchií s preferovaným rodičem

Jako příklad uveďme opět webový katalog zboží se stromovými kategoriemi. Každé zboží je zařazeno do jedné kategorie. Ta určuje zařazení, které se také vypisuje při zobrazení detailu produktu (ve stromové hierarchii by to byla cesta ke kořeni). Výrobek ovšem logicky patří i do jiné kategorie, v jejímž seznamu produktů by se měl také zobrazovat.

Konkrétní příklad je pak zboží „inkoust do tiskárny“ s primárním zařazením „Spotřební materiál / Inkousty “, které se také zobrazuje v „Tiskárny / Doplňky“.

Druhý konkrétní příklad z katalogu: Primárním zařazením produktu je tradiční stromová kategorie. Navíc existují kategorie do kterých jsou produkty zařazovány podle barvy. Kromě tradičního zařazení tak má uzel ještě například rodiče „Zelené produkty“.



Obrázek 3.5: Vedlejší vazby pouze u listů a obecný případ

Struktura tubulek

Hierarchii s preferovaným rodičem bychom místo obecné hierarchie možná měli spíše považovat za strom s dalšími informacemi. Takovým způsobem ji také lze nejlépe uložit. K libovolné stromové metodě přidáme tabulku s dalšími rodiči.

```
CREATE TABLE další_rodiče {
  nadřazený INTEGER NOT NULL,
  podřízený INTEGER NOT NULL,
  PRIMARY KEY (nadřazený,podřízený)
}
```

Definici tabulky by ještě bylo vhodné doplnit o cizí klíče. Oba sloupce odkazují na uzly v tabulce s uloženým stromem, jenž je určen hlavními vazbami.

Vedlejší vazby pouze u listů

Než se podíváme jak se na záznamy dotazovat, rozebereme, kdy se takové tabulky v praxi opravdu používají. Ve výše uvedeném případě jsme situaci zjednodušili, a tím paradoxně zobecnili problém dotazování. V našem příkladu jsou produkty listovými uzly jedné hierarchie a nepřímé vazby mohou existovat mezi libovolnými uzly (s výjimkou omezení danými samotnou definicí hierarchie).

V reálném katalogu s kategoriemi a produkty mají obvykle oba typy v databázi vlastní tabulku. Uzlem hierarchie je pouze kategorie a produkt se na své zařazení pouze odkazuje. První tabulka ukládá strom kategorií a produkty se na ně odkazují přes cizí klíč (to je hlavní rodič). Třetí tabulka je pomocná tabulka dalších rodičů, která k sobě váže vždy produkt s jeho vedlejší kategorií.

V obecném případě může mít více rodičů libovolný uzel (z pohledu katalogu tedy i kategorie). Ve speciálním a obvyklejším případě s rozděleným katalogem mají více rodičů pouze produkty. Převedeno zpět do našeho hierarchického pohledu, kde je vše v jednom stromě, pak více rodičů mají pouze listy. Příklad obou situací je na obrázků 3.5, vedlejší vazby jsou znázorněny přerušovanou čarou.

Dotazování

Výše uvedené omezení má zásadní dopad na konstrukci dotazů. Pro získání všech podřízených lze ve speciálním případě ze stromu vybrat hlavní podstrom (to je obyčejný stromový dotaz) a k výsledku připojit uzly vázané

vedlejší vazbou. Máme zaručeno, že všechny další podřízené uzly jsou dosažitelné přes nepřímou vazbu z nějakého uzlu hlavního podstromu (jelikož jsou to listy, tak pod nimi další uzly nemůžou být).

V obecném případě toto neplatí a podřízené uzly mohou být dosažitelné přes libovolný větší počet vedlejších vazeb. Potom nezbývá než rekurzívní dotazování. Iterací je stejný počet, jako maximum počtu nepřímých vazeb na cestě od kořene hledaného podstromu k podřízeným uzlům.

Při zobrazování zařazení uzlu v hierarchii se využívá cesta ke kořeni po hlavních vazbách. Pro nadřazené uzly tak stačí běžný dotaz nad stromem. Vedlejší vazby se pro „procházení nahoru“ takřka nepoužívají. Ono také hlavní a vedlejší vazby rozlišujeme zejména proto, abychom mohli ze všech cest přes nadřízené vybrat nějakou jednoznačně.

Shrnutí

Tato metoda na pomezí stromu a obecné hierarchie je vhodná zejména ve specifických případech, kdy je zaručena zmíněná dosažitelnost podřízených přes maximálně jednu nepřímou vazbu. Na příkladech jsme ukázali, že to není přehnaně silný požadavek a pro obvyklé situace není omezující.

Není nezbytně nutné, aby byla hierarchie rozdělena do dvou tabulek, tak jak provedeno v popisu výše. Pro dané omezení je však takové rozdělení praktické a přirozené. V obecném případě je vhodnější použít skutečnou metodu pro obecné hierarchie s ohodnocením hran.

3.7.2 Rozšíření stromových metod

Nejjednodušší způsob uložení obecných hierarchií je tabulka uzlů s tabulkou hran grafu. Je to v podstatě zobecněné triviální uložení se stejnými problémy a nutností rekurzivních dotazů. Navíc se situace komplikuje použitím dvou tabulek a nutností jejich spojování. Jako další varianta se nabízí použití jedné tabulky uzlů a složeného typu pro zaznamenání všech rodičů. Taková metoda se původnímu triviálnímu uložení podobá možná ještě více. Zde je problém se složenými typy, jejichž problematiku jsme diskutovali v sekci 3.3.

Zobecněná metoda uložení s tabulkou vazeb

Od uložení hran a uzlů do dvou tabulek je jen krůček k zobecnění stromové metody s tabulkou vazeb, ve které ukládáme i nepřímé vazby včetně vzdálenosti koncových uzlů. To je také jediná metoda, která se dá použít pro efektivní ukládání obecných hierarchií (tj. situace kdy mají uzly více rovnocenných nadřízených). Všechny ostatní popsané pokročilejší metody zásadním

způsobem využívají stromových vlastností a nelze je pro obecné hierarchie zobecnit.

V případě metody s tabulkou vazeb pro obecné hierarchie zřejmě z definice hlavní tabulky zmizí sloupec s odkazem na nadřízeného. Všechna data týkající se struktury tak zůstanou pouze v pomocné tabulce. Dotazování je obdobné jako v původní variantě. Neexistuje nic jako cesta ke kořeni, ale lze vybrat „nadstrom“ k danému uzlu, který obsahuje všechny nadřazené uzly. Tímto způsobem lze též snadno uložit hierarchii s hlavními a vedlejšími vazbami. Stačí rozšířit tabulku vazeb o sloupec s příznakem, který vazby rozliší a doplnit ji o vhodný index.

3.8 Shrnutí

3.8.1 Indexování stromů

Když pomineme triviální uložení, které fakticky hierarchii nijak speciálně neindexuje, tak se u popsaných metod objevují dva přístupy, jak stromová data efektivně indexovat.

Nadřazené uzly

První možností je zaznamenání nadřazených uzlů vybraným způsobem. Pokud máme ke každému uzlu jeho nadřízené, umíme většinou z dat vyčíst ke každému uzlu i jeho podstrom.

Do této kategorie patří metody cesta ke kořeni, cesta ke kořeni jako složený typ a uložení pomocí tabulky vazeb. Liší se jenom v tom, jak informaci o nadřazených uzlech ukládají. Tabulka vazeb používá, co se týče návrhu databáze a dobrých zvyklostí, nejčistší řešení. Naopak metoda s cestou ke kořeni zhušťuje všechnu informaci do jednoho textového pole a snaží se o co nejméně pomocných dat a jednoduché dotazy. Cesta ke kořeni jako složený typ je pak kompromisem mezi oběma řešeními, ale naráží na závislost na nových vlastnostech SQL standardu, respektive nutnosti používat proprietární rozšíření jednotlivých databázových systémů.

Průchod stromem

Druhým přístupem je zaznamenání průchodu stromem vhodnými hodnotami. Očíslování uzlů nesmí pouze zachytit strukturu stromu, ale musí také umožnit efektivní dotazování. Do této kategorie patří zbylé dvě metody. Vnořené množiny (a jejich varianty) a uložení průchodu do hloubky. Tyto metody typicky umožňují krátké a jednoduché dotazy nad hierarchickou strukturou.

Nevýhodou je globální očíslování závislé na všech uzlech a z toho plynoucí problémy při aktualizacích. Použití těchto metod je výhodné zejména u statických kolekcí.

3.8.2 Přehled vlastností

Na závěr kapitoly je uveden přehled všech metod s jejich hlavními rysy a odlišnostmi. U každé metody je zmíněn způsob procházení podstromu, respektive řazení výsledku. Tato vlastnost může být významná při výběru vhodné metody pro konkrétní použití. Dále je u každé metody uvedena složitost aktualizací stromové struktury podle klasifikace uvedené v úvodu práce. Aktualizace stačí rozdělit na dva typy. Jednoduchou aktualizací je myšleno vložení a smazání listu. Pod složitou pak spadá vložení a smazání vnitřního uzlu nebo přesun a smazání podstromu.

triviální
: průchod podstromu pouze rekurzivními dotazy
: lokální aktualizace
: pro práci s hierarchiemi nevhodné

cesta ke kořeni
: průchod do šířky i do hloubky
: jednoduchá aktualizace: lokální změna
: složitá aktualizace: úprava podstromu
: hledání pomocí textových porovnání
: parsování cesty lepší provádět mimo databázi

cesta ke kořeni jako složený typ
: průchod do šířky i do hloubky
: jednoduchá aktualizace: lokální změna
: složitá aktualizace: úprava podstromu
: nutná podpora složených typů databázovým systémem

tabulka vazeb
: 2 tabulky
: průchod do hloubky
: jednoduchá aktualizace: lokální změna
: složitá aktualizace: úprava podstromu (ale jen v 2. tabulce)
: rychlé dotazy na hloubku a vzdálenosti uzlů

vnořené množiny
: průchod do hloubky
: všechny aktualizace: globální změna
: definuje pořadí synů
: jednoduché dotazy

uložený průchod do hloubky
: průchod do hloubky i do šířky
: všechny aktualizace: globální změna definuje pořadí synů
: neefektivní dotazy na předky

Kapitola 4

Porovnání metod

V této kapitole budou metody představené v předchozím textu vzájemně porovnány z praktického hlediska.

Nad jednotlivými způsoby uložení dat budou spuštěny vybrané dotazy z předchozí kapitoly a měřena doba jejich provádění. Naší snahou bude simulovat typické dotazy nad skutečnou hierarchickou databází. Před samotným testováním musí být reálná biologická data transformována z poskytovaných nativních formátů do tabulek v databázi.

Pro transformaci dat a samotné měření je využíván skriptovací jazyk Python. Jeho předností je úsporný a přesto přehledný zápis. Pro připojení k databázi je použita standardní knihovna PyGreSQL.

Pro ukládání obecných hierarchií byla popsána jediná metoda, proto se tato kapitola zabývá výhradně metodami pro ukládání stromových hierarchií.

4.1 Zdroje dat

Zdrojem testovacích dat jsou veřejné biologické databáze. To však neznamená, že by výsledky nebyly relevantní i pro hierarchická data z jiných oborů.

Vhodná sada dat by měla obsahovat dostatečný počet uzlů pro zajištění relevance měření. Také je užitečné, pokud je možné poskytovaný export dat snadno zpracovat. Jelikož další zpracování je obvykle primárním účelem exportu, není tento požadavek problematický.

Pro účely měření byly vybrány následující tři kolekce dat:

4.1.1 Taxonomie

Jako první zdroj dat poslouží klasifikace živých organizmů. Data jsou veřejně přístupná na webových stránkách *National Center for Biotechnology*

Information (dále bude zkracováno na NCBI). Tato instituce spravuje více biologických databází, z nichž nás bude zajímat pouze *Taxonomy*.

Taxonomický systém zařazuje všechny organizmy do hierarchických skupin. V českém názvosloví jsou označovány jako čeledi, řády, třídy apod. Základní skupiny jsou členěny ještě do menších částí, takže hloubka tohoto stromu je relativně velká. Databáze obsahuje klasifikaci živočichů, hub, rostlin, virů a bakterií.

Klasifikaci lze procházet přímo přes webové rozhraní na adrese:
`http://www.ncbi.nlm.nih.gov/Taxonomy/`

4.1.2 Klasifikace struktury proteinů

Druhým zdrojem dat je klasifikace struktury proteinů. Zkráceně SCOP, z anglického *Structural Classification of Proteins*. Data lze získat i procházet na webové adrese: `http://scop.mrc-lmb.cam.ac.uk/scop/`

4.1.3 Genová ontologie

Poslední sada dat obsahuje stavební části živých organizmů (od buňky až k nukleotidům) a buněčné a molekulární procesy.

Genová ontologie je od obou předchozích sad dat odlišná. Obsahuje hrany dvou typů. První vazba *is_a* říká, že potomek je podtypem rodiče. Druhý typ vazby je *part_of*, která říká že potomek je částí celku[1]. Genová ontologie je navíc obecnou hierarchií. To znamená, že může existovat více rodičovských uzlů. Není nijak omezen ani počet rodičů konkrétního typu. Jeden objekt může být specializací i částí více dalších objektů.

Pro účely našeho porovnávání potřebujeme strom. Omezíme se proto pouze na vazbu *is_a*, která reprezentuje dědičnost objektů. Většina uzlů má pouze jednoho rodiče tohoto typu. Abychom dostali skutečně strom, musíme ještě vyřešit zbývající případy, kdy je rodičů více. Ve stromě pro účely měření bude takový uzel synem pouze jednoho z nich. Každému ze zbývajících rodičů přidáme nového syna, který bude kopií původního uzlu. Kopie budou vždy listy, i kdyby originální uzel měl další podřízené. To znamená, že kopie funguje pouze jako reference na skutečný uzel. V naší transformované hierarchii tedy bude o něco více uzlů než v původních datech.

Původní hierarchie je nesouvislá a skládá se ze tří komponent souvislosti[2]. Poslední technickou záležitostí je tak vytvoření jednoho kořene, pod

[1] Význam vazeb je stejný jako princip dědičnosti a kompozice v objektově orientovaném návrhu a programování.

[2] Každá komponenta souvislosti obsahuje jeden typ dat. Konkrétně komponenty představují již zmíněné části živých organizmů, buněčné procesy a molekulární procesy.

který zavěsíme všechny tři komponenty transformované podle výše uvedeného postupu. Tím jsme z původní nesouvislé obecné hierarchie vytvořili strom, který stále obsahuje smysluplná data. Všechny informace obsažené v původní struktuře lze vyčíst i z našeho stromu.

Motivací k použití této sady dat je ukázka reálné obecné hierarchie a také jednoduchý standardizovaný formát exportu. Celé uložení testovacích dat do tabulky relační databáze proto není složitější než u předchozích dvou sad, i když je na začátku nutná transformace na strom.

4.2 Statistické údaje

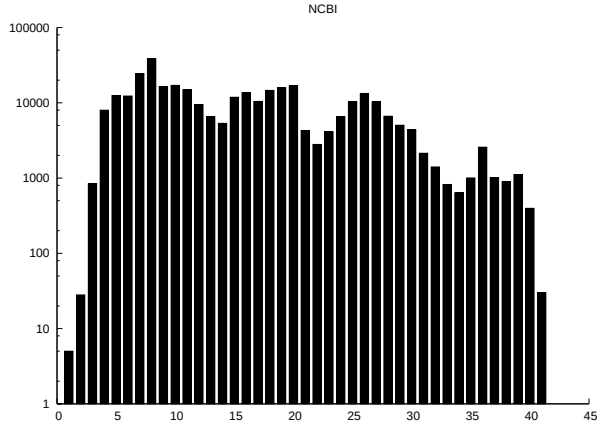
Charakteristika testovacích dat je důležitá ke správné interpretaci naměřených výsledků. Metody uložení neukládají všechny kolekce dat stejně dobře. Pokud budeme znát strukturu konkrétního případu, můžeme zkoumat důvody odlišného chování a odhadnout vlastnosti metod pro data s různou strukturou.

Například velikost pomocné tabulky u metody s tabulkou vazeb závisí na hloubce stromu. Pro strom hloubky jedna bude obsahovat $n - 1$ vazeb, kde n je počet uzlů. Pro opačný extrém, degenerovaný strom hloubky $n - 1$ bude obsahovat $\frac{n^2-n}{2}$ vazeb. Velikost tabulky, jenž zaručeně ovlivňuje rychlost dotazování, se tedy podle charakteru dat pohybuje mezi $O(n)$ a $O(n^2)$.

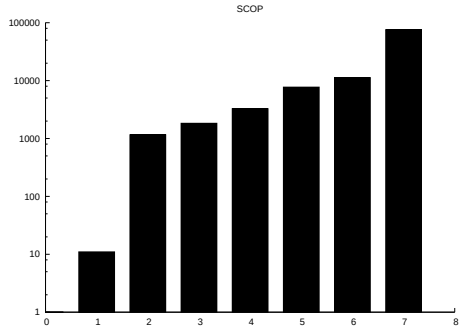
První ze zkoumaných vlastností je hloubka stromu a s ní související veličiny. Samotná hloubka je pouze nejdelší cestou ke kořeni a nás zajímá i celkové rozložení uzlů do jednodlivých hladin a průměrná hloubka. O tvaru stromu také poměrně dobře vypovídá průměrný počet synů na vnitřní uzel. Čím více se strom větví, tím menší má průměrnou hloubku při stejné velikosti.

Druhou důležitou veličinou je počet uzlů stromu. Při měření na různě velkých kolekcích máme možnost relativně spolehlivě aproximovat chování metody pro jiné počty uzlů.

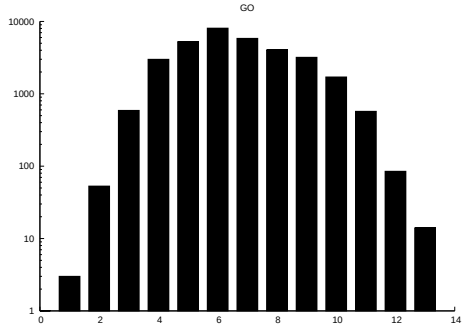
Následují statistická data testovacích kolekcí. Na straně 59 je pro každou sadu uveden histogram rozložení hloubek jednotlivých uzlů. Ve vyváženém stromě roste množství uzlů v hladinách exponenciálně. V diagramu je použita logaritmická stupnice, takže u vyváženého stromu by hodnoty ležely na přímce.



Obrázek 4.1: Histogram hloubky pro NCBI



Obrázek 4.2: Histogram hloubky pro SCOP



Obrázek 4.3: Histogram hloubky pro GO

4.2.1 Data NCBI

Počet uzlů	329 173
Počet listů	266 486 (81%)
Hloubka stromu	41
Průměr. hloubka stromu	15,33
Max. počet synů	5 600
Průměr. počet synů	5,25

NCBI data jsou největší kolekcí testovacích dat. Dané množství uzlů lze považovat za velmi rozsáhlou hierarchii. Stejně tak jsou relativně extrémní krajní případy pro hloubku a počet synů. Z histogramu (obr. 4.1) je vidět, že strom je velmi nevyvážený. Vyvážený strom se stejným počtem uzlů, kde každý uzel má 4 syny by měl hloubku 9. Pokud bychom povolili 5 synů, pak by hloubka vyváženého stromu byla pouze 7.

4.2.2 Data SCOP

Počet uzlů	101 184
Počet listů	75 930 (75%)
Hloubka stromu	7
Průměr. hloubka stromu	6,51
Max. počet synů	539
Průměr. počet synů	4

U reálných dat nelze očekávat absolutně vyvážený strom. Pohledem na histogram (obr. 4.2) zjistíme, že tato kolekce se vyvážené struktuře relativně hodně přibližuje.

4.2.3 Data GO

Počet uzlů	32 418
Počet listů	24 276 (75%)
Hloubka stromu	13
Průměr. hloubka stromu	6,63
Max. počet synů	386
Průměr. počet synů	3,98

Strom genové ontologie obsahuje ze všech tří kolekcí nejméně uzlů. Stále ještě ovšem obsahuje dost uzlů k tomu, aby mělo smysl se zabývat vhodnou reprezentací dat v relační databázi a rozdíly mezi jednotlivými metodami byly měřitelné.

Pro ještě menší kolekce o velikosti stovek, možná i několik tisíců uzlů je z hlediska doby trvání dotazů v podstatě jedno, jakou z metod použijeme. Více než rychlost dotazů, která bude pro všechny metody obdobná, nás budou zajímat ostatní vlastnosti metod.

I když GO data obsahují jen třetinu uzlů co SCOP data, hloubka stromu je přesto větší.

4.3 Konverze dat z nativních formátů

Každá databáze poskytuje data ve vlastním formátu. Pro porovnání metod musíme tato data nejprve převést do relační databáze. Tato část se zabývá parsováním nativních formátů a převodem do triviálního uložení. Z hlediska samotného porovnání je následující text irelevantní.

Exporty jednotlivých sad ve verzi použité pro měření jsou součástí přiloženého CD.

Formát NCBI

Každý uzel má přidělen jednoznačný číselný identifikátor. NCBI poskytuje export dat rozdělený do několika souborů, z nichž některé obsahují odkazy na nestromová data a další pro nás nepodstatné atributy. Pro import stromu s názvy uzlů jsou významné pouze dva soubory:

nodes.dmp
: Obsahuje identifikátor uzlu a všechny jeho atributy kromě názvu. Pro účely měření je podstatný jenom odkaz na nadřazený uzel.

names.dmp
: Obsahuje názvy jednotlivých uzlů. Každý uzel může mít přiřazeno několik názvů, které jsou v souboru rozlišeny svoji rolí. Role je například *Vědecké jméno*, *Synonymum* apod. Při převodu preferujeme vědecké jméno. Pokud není uvedeno tak je použito první uvedené.

Všechny soubory používají znak `|` jako oddělovač záznamů na jedné řádce. Detailní specifikace formátu je v souboru `readme.txt`, který je součásti exportu.

Parsování a uložení do databáze probíhá ve dvou krocích. Nejprve se podle souboru *nodes.dmp* vytvoří nepojmenované řádky, kterým se následovně při průchodu souboru *names.dmp* přiřazují jména. Příslušný skript lze nalézt v příloze B.1.1. Během plnění tabulky využijeme možnost odložení kontroly cizího klíče na sloupci *nadřazený*, protože data nejsou v exportu nijak řazena a některé synovské uzly jsou vloženy před svými rodiči.

Formát SCOP

Formát kolekce SCOP je velmi podobný předchozímu. Export se také skládá z několika souborů, z nichž opět využijeme pouze dva. Jednotlivé záznamy na řádce jsou v tomto případě odděleny tabelátorem. Soubor `dir.hie` obsahuje stromovou strukturu, tedy identifikátor, nadřazený uzel a navíc i seznam podřízených (ten na nic nepotřebujeme). Soubor `dir.des` pak mimo jiné obsahuje názvy uzlů. Uložení do databáze (skript viz. B.1.2) probíhá stejně jako u NCBI, dokonce je ještě jednodušší, neboť ke každému uzlu přísluší pouze jeden název a není třeba vybírat z více variant.

Formát genové ontologie

Data GO jsou uložena ve standardním OBO formátu pro ukládání ontologií. Všechny informace jsou v jednom souboru, který je rozdělen na mnoho sekcí. Nás zajímají pouze sekce `[Term]`, protože každá z nich reprezentuje jeden uzel. Sekce obsahuje pojmenované atributy (co řádka to jeden atribut) ve formátu `jméno:hodnota`. Pro naše účely jsou důležité především atributy *id*, *name* a *is_a*. Jeden atribut může mít více hodnot, pak je uveden u jednoho uzlu několikrát. To je případ *is_a*, který obsahuje odkaz na nadřazený uzel. Skript B.1.3 je relativně jednoduchý.

Musíme pouze vyřešit několik technických záležitostí. Jednak je třeba převést GO identifikátor na celé číslo, což se provede prostým odseknutím textového prefixu. Dále z obecné hierarchie vytváříme strom podle popisu v 4.1.3. Tedy pro uzly s více rodiči vložíme do tabulky ještě příslušné kopie záznamu s unikátním *id*. Také ignorujeme neplatné uzly označené atributem *is_obsolete*. Poslední technická záležitost je pak spojení tří samostatných stromů do jednoho. Zavěšíme původní kořeny pod nový uzel, který se tak stane kořenem celého GO stromu.

4.4 Transformace dat

Testovací kolekce již máme převedené do relační databáze v podobě triviálního uložení. Abychom mohli začít s porovnáním metod, zbývá tuto tabulku transformovat do všech ostatních způsobů uložení. K tomu opět poslouží jednoduché skripty v jazyce Python, které se nachází v příloze B.2. Pro každou metodu existuje jeden převodní skript.

Provádění skriptů je celkem přímočaré. Při plnění tabulek obou metod s cestou ke kořeni se načtou všechny záznamy z triviální tabulky. V pomocném poli se uchovává cesta už uložených uzlů. Cesta ke kořeni každého uzlu vznikne složením cesty rodiče s rodičovským identifikátorem. Načtený záznam

tak lze uložit do cílové tabulky, pokud už byl zpracován jeho rodič nebo je to přímo kořen. Uzly, které uložit zatím nelze, skript odkládá do dalšího pomocného pole a zpracovává nakonec. To by teoreticky mohlo vést až ke kvadratické časové složitosti vzhledem k počtu uzlů. Data v tabulce triviálního uložení pochází z exportů, která jsou seřazena relativně příhodně a počet uzlů pro závěrečné zpracování je tak minimální. Doba provádění skriptů navíc není příliš zajímavá, protože konverze je pro každou kolekci dat prováděna pouze jednou.

Ostatní metody prochází strom do hloubky. Metody, které používají očíslování tak přímo získají potřebné hodnoty pomocných dat. Skript pro převod na metodu s tabulkou vazeb si navíc ukládá seznam nadřízených, takže může každému uzlu vytvořit odpovídající záznamy do pomocné tabulky.

4.5 Porovnání metod

Porovnávány budou dva základní dotazy nad stromovou strukturou. Získání nadřízených uzlů (cesta ke kořeni) a podřízených uzlů (uzly v podstromu). Složitější stromové dotazy jsou jen obdoby těchto dvou a i jejich časová náročnost pro konkretní metodu uložení bude korespondovat se základním dotazem.

Požadované uzly se dají u každé metody získat jedním dotazem, který byl popsán v předchozí kapitole.

4.5.1 Metodika měření

Jednotlivé testovací sady jsou testovány zvlášť. Každý test začíná náhodným výběrem 2 000 uzlů. Pro každou metodu je pak nad všemi vybranými uzly postupně proveden odpovídající dotaz. Měřena je doba provedení dotazu a doba potřebná k získání dat z databáze[3]. I když doba pro načtení dat tvoří pouze menší část celkové doby zpracování, pro relativně malé kolekce (jako je genová ontologie) je to podíl významný. Doba pro získání dat je pro jednotlivé metody rozdílná, takže má smysl ji také sledovat.

Všechny výsledné časy jsou v tabulkách uvedeny v sekundách.

Pro účely testování je využíván modul uvedený v příloze B.3. Ten poskytuje metody pro náhodné generování identifikátorů uzlů, testování a prezentaci výsledků. Samotný test pak zajišťuje metoda `benchmark`, která má několik povinných parametrů:

benchIds – Seznam *id* uzlů pro které se provede požadovaný dotaz.

[3]V angličtině a programovacích jazycích je tato operace nazvána jednoduše – fetch.

table – Jméno tabulky s uzly. Ta slouží k získání seznamu sloupců pro parametrizování dotazů.

query – Samotný dotaz. Proměnné jsou v něm označeny stejným způsobem jako v použité standardní databázové knihovně PyGreSQL znakem % a typem parametru.

params – Jména sloupců, jejichž hodnota bude dosazena za proměnné v dotazu.

Zbývající parametry jsou nepovinné:

skip – Počet záznamů, pro které bude proveden dotaz, ale čas dotazů se nebude započítávat do výsledku.

debug – Tiskne řádky vrácené ve výsledcích dotazů.

customQuery – Testovacímu prostředí lze předat funkci, která předzpracuje dotaz na databázi pro uzel. Používá se tam, kde jsou parametry dosazované do dotazu složitější než jen prostá hodnota sloupce.

Všechna měření probíhají na čerstvě spuštěné instanci PostgreSQL. Test je zahájen provedením dotazu pro 50 náhodných uzlů, jejichž doba se nezapočítává do výsledku. První dotazy slouží k simulaci reálného provozu. Tím se například naplní cache pro často čtené datové stránky a také cache pro samotný dotaz s plánem provedení. Skutečné měření času pak probíhá pro 2 000 náhodných uzlů. Náhodné identifikátory jsou vygenerovány na počátku testu a všechny metody jsou testovány nad stejnou množinou dat.

Funkce pro předzpracování dotazu se používá i tam, kde lze dotaz zapsat prostředky SQL s parametry odkazujícími na sloupce, ale provedení by bylo podstatně méně efektivní než zpracování v Pythonu. Při reálném použití lze řetězcové operace na atributech uzlů provést mimo SQL a dosadit již předpřipravené hodnoty. Pro zachování objektivity je v takovém případě čas nutný na provedení pomocné funkce přičten k celkovému času, i když z výsledků měření je vidět, že doba pro přípravu předzpracování dotazu je minimální. Naopak formátování parametrů do správného tvaru v SQL může zvednout čas potřebný na provedení dotazu až o polovinu. Při porovnávání je kladen důraz na praktické hledisko, takže nevyužití skriptování by výsledek zkreslilo, neboť by neodpovídal reálné použitelnosti jednotlivých metod. Předzpracování parametrů se týká zejména metody ukládání cesty ke kořeni.

Při testování se z vrácených řádek spočítá hash hodnota, aby se dalo ověřit, že dotazy nad všemi metodami vrací stejný výsledek a porovnáváme tedy korespondující dotazy.

4.5.2 Parametry testovacího prostředí

Parametry PC na kterém jsou všechny testy změřeny jsou: AMD Athlon 3000+, 1GB RAM se systémem Kubuntu Linux 7.04 64-bit.

Pro testování byla použita databáze PostgreSQL ve verzi 8.2 s defaultním nastavením. U starších verzí bývaly reálné parametry výchozího nastavení spíše podhodnoceny. To se se současnou verzí změnilo a není nutné ho pro naše účely měnit.

PostgreSQL používá pro plánování dotazů statistiky výskytu jednotlivých hodnot v indexu. Jestliže statistika neodpovídá datům, nemusí být dotazy provedeny optimálně. Před testy byl na každou tabulku aplikován příkaz `VACUUM ANANALYZE`, který mimo jiné provede aktualizaci statistiky. Dále příkaz optimalizuje tabulku, což by v našem případě nemělo mít žádný efekt. Podrobnější informace o příkazu `VACUUM` lze nalézt například v [6] nebo v online dokumentaci k PostgreSQL.

Skriptovací jazyk Python byl použit ve verzi 2.5.1.

4.5.3 Cesta ke kořeni

První testovanou operací je získání nadřízených. Takový dotaz je potřeba všude tam, kde se prezentuje zařazení uzlu v hierarchii. Konkrétním příkladem je webové procházení kolekcí NCBI a SCOP, kde je jsou na stránce s detailem uzlu vypsáni synové[4] a cesta ke kořeni, o kterou v tomto testu jde. Výsledné uzly se přirozeně vypisují seřazené od kořene k hlubším uzlů, proto i testované dotazy obsahují řazení podle hloubky uzlu.

Metoda nazvaná uložení cesty ke kořeni má identifikátory požadovaných uzlů už obsaženy v jednom sloupci oddělené speciálním znakem. Varianta dotazu, která provádí extrahování pomocí SQL, je velmi neefektivní. Jestliže má test odrážet reálné použití je potřeba měřit variantu, kde se hodnota pole rozdělí na id v testovacím skriptu a v dotazu se uzly hledají podle id[5].

Jak už bylo zmíněno v odstavci 4.5.1, doba pro provedení pomocné metody je přičtena k celkovému času. Proto u této metody není celková doba rovná součtu doby pro provedení dotazu a času pro získání dat. Doba pro předzpracování tvoří v tomto případě méně než 1% celkového času.

Pro genovou ontologii jsou časy jednotlivých metod následující:

[4] Dotaz na syny je triviální a u všech metod stejný. Tím se nezabýváme.

[5] Ve skutečnosti stačí ve sloupci s cestou ke kořeni nahradit oddělovač čárkou a výsledný řetězec použít v IN podmínce

metoda	dotaz	data	celkem
cesta ke kořeni	1,293	0,628	1,938
cesta jako složený typ	1,536	0,934	2,470
tabulka vazeb	1,881	0,578	2,459
vnořené množiny	17,186	0,944	18,130
uložený průchod	17,310	0,942	18,252

Výsledky pro data SCOP:

metoda	dotaz	data	celkem
cesta ke kořeni	1,352	0,613	1,982
cesta jako složený typ	1,590	0,911	2,501
tabulka vazeb	4,949	0,565	5,514
vnořené množiny	29,658	0,914	30,572
uložený průchod	28,144	0,898	29,042

A pro data NCBI:

metoda	dotaz	data	celkem
cesta ke kořeni	9,021	2,349	11,392
cesta jako složený typ	11,508	3,572	15,080
tabulka vazeb	34,830	2,159	36,989
vnořené množiny	61,459	2,056	63,515
uložený průchod	105,820	2,030	107,850

Metody s uloženou cestou ke kořeni

Z výsledků je vidět, že nejlépe dopadla metoda „cesta ke kořeni“. To je očekávatelný výsledek, protože tato metoda ve svých pomocných datech již obsahuje odkazy na výsledné uzly a stačí tak pouze načíst správné řádky podle primárního klíče. Uložení cesty ke kořeni jako složený typ je podobné předchozímu případu, jen operace s polem jsou o něco pomalejší. Nevýhodou je také nejvyšší čas nutný pro načtení dat. Důvod není úplně zřejmý, ale mohlo by to být způsobeno převodem složeného typu na textovou reprezentaci při vracení výsledku z databáze.

Metody s tabulkou vazeb

Měli bychom si povšimnout nevyrovnaného výsledku metody s tabulkou vazeb. V teoretické části bylo zmíněno, že velikost pomocné tabulky roste asymptoticky rychleji než počet uzlů. Počet uložených vazeb také nepříznivě ovlivňuje hloubka stromu. Pro první nejmenší kolekci, která je relativně malá a mělká, je tato metoda ještě v podstatě stejně rychlá jako obě ukládání

cesty ke kořeni. Pro poslední kolekci NCBI, která je třikrát větší než kolekce SCOP je výsledek podstatně horší. To je navíc umocněno extrémní hloubkou v některých částech stromu, který není vůbec vyvážený.

Z tabulky časů není vidět ještě jedna skutečnost, která se týká metody s tabulkou vazeb. Všechny ostatní metody vykazují při opakovaných testech[6] stabilní výsledky. Jinak je tomu v případě uložení s tabulkou vazeb, které je pravděpodobně velmi závislé na dotazovaných uzlech a ani množství 2 000 testovaných uzlů nezajistí stabilní průměrný výsledek. Například pro data SCOP se obvyklý naměřený výsledek pro tabulky vazeb pohybuje mezi 3 a 10 sekundami, kdežto výsledky ostatních metod pro různá vstupní data se typicky liší o 100 milisekund.

Metody s očíslováním uzlů

Výsledek vnořených množin také není příliš dobrý. Způsobeno je to nemožností efektivně provést druhou část podmínky pro pravé hodnoty. Uzly splňující podmínku pro levé hodnoty lze vybrat z indexu, ale podmínka sama o sobě není příliš selektivní a druhá část se už nutně musí provádět sekvenčním průchodem.

Metoda s uloženým průchodem dopadala nejhůře, ale pro menší kolekce je na tom v podstatě stejně jako vnořené množiny. Její špatný výsledek jsme také předpokládali, protože dotaz obsahuje konstrukci `GROUP BY` a má nejsložitější plán provedení.

4.5.4 Dotaz na podstrom

Druhou testovanou operací je získání všech podřízených daného uzlu. Takový dotaz se běžně nepoužívá přímo pro výpis všech uzlů ve výsledku (takových uzlů je obvykle velké množství), ale pro další zpracování. Výsledná sada se dále filtruje nebo je na ni aplikována agregační funkce.

Pomocná metoda pro předpřipravení parametrů je použita tentokrát u obou metod ukládání cesty ke kořeni. Jejich podobu a samotné dotazy lze nalézt v příloze B.3.4.

V případě podřízených má smysl porovnávat i různé způsoby řazení. Nejdříve se podíváme jak proběhl test bez jakéhokoliv řazení. Takový dotaz má smysl například pokud používáme nějakou hodnotu uzlu v agregační funkci, kde na pořadí nezáleží. Pro genovou ontologii jsou výsledky následující:

[6] To znamená, že jsou prováděny pro jiná náhodná vstupní data

metoda	dotaz	data	celkem
cesta ke kořeni	1,397	0,337	1,753
cesta jako složený typ	1,515	0,494	2,034
tabulka vazeb	2,506	0,312	2,818
vnořené množiny	1,066	0,489	1,555
uložený průchod	1,495	0,485	1,980

Test nad SCOP daty:

metoda	dotaz	data	celkem
cesta ke kořeni	1,435	0,489	1,944
cesta jako složený typ	1,961	0,716	2,699
tabulka vazeb	3,982	0,436	4,418
vnořené množiny	1,182	0,695	1,877
uložený průchod	1,540	0,710	2,250

A test nad NCBI daty:

metoda	dotaz	data	celkem
cesta ke kořeni	13,348	1,957	15,334
cesta jako složený typ	14,563	2,961	17,566
tabulka vazeb	39,675	1,001	40,676
vnořené množiny	2,644	2,374	5,018
uložený průchod	3,495	2,640	6,135

Stejně jako u dotazu na nadřízené platí, že čím více je dat, tím dává metoda s tabulkou vazeb horší výsledky. V dotazech na podstrom mají navrch metody, které přiřazují uzlům očíslování (vnořené množiny a uložený průchod). Pro menší kolekce dat se jim metody s ukládáním cesty ke kořeni mohou rovnat, ale u rozsáhlých dat je rozdíl již markantní.

4.5.5 Dotaz na podstrom s řazením

Test s řazením uzlů podle průchodu do hloubky a do šířky byl proveden pouze na SCOP datech. Jelikož se nám jedná o poměr k době provedení dotazu bez řazení, měl by test pro zbylé kolekce dat dopadnout obdobně. Aby se rozdíly mezi dotazy projevily, bylo tentokrát použito pro testování 4 000 uzlů. Tabulka obsahuje pouze celkové časy.

metoda	bez řazení	do hloubky	do šířky
cesta ke kořeni	5,567	6,514	6,843
cesta jako složený typ	9,476	9,716	9,978
tabulka vazeb	7,072	-	7,444
vnořené množiny	7,901	7,759	-
uložený průchod	9,231	10,059	9,557

V tabulce je vidět, že dotazy s řazením jsou o něco náročnější. U vnořených množin je čas bez řazení o něco vyšší, ale ten bude způsoben chybou měření.

Pokud se zaměříme na porovnání času řazení podle průchodu do hlouby a do šířky u metod, pro které obě varianty řazení existují, tak rozdíl mezi časy je minimální a pohybuje se na hranici chyby měření. Obě varianty řazení se liší pouze způsobem, jakým se získává hodnota, podle které se třídí. Samotný běh třídícího algoritmu je pak vždy stejný. Z toho také plyne, že ani doby potřebné na provedení obou dotazů by se neměly příliš lišit, což výsledek testu potvrzuje. Závěr je tedy takový, že z hlediska času je jedno jakým způsobem výsledek řadíme. Důležitější je pro nás fakt zda-li je požadovaný způsob vůbec možný.

4.6 Interpretace výsledků měření

Porovnání metod v různých situacích neodhalilo žádnou metodu, která by byla lepší než ostatní. Stejně tak mezi testovanými metodami není žádná, která ve všech ohledech za ostatními zaostává. Každá z pěti testovaných metod je použitelná v jiné situaci. Nelze dát jednoduchou odpověď na to, kdy kterou metodu použít.

Výběr vhodné metody závisí především na velikosti uložené hierarchie, na dotazech, které budou nad daty prováděny a na četnosti aktualizací. Je také dobré si uvědomit, že pro testování byly použity rozsáhlé kolekce dat. Mnoho reálných stromových struktur bude menších než nejmenší testovaná sada. U takových kolekcí je výkonnostní rozdíl méně patrný a důraz bude kladen spíše na snadnost použití a podporu netypických, ale v daném případě užitečných stromových dotazů.

Obecně lze říci, že metody s cestou ke kořeni umějí rychle získat nadřazené uzly, ale při hledání podřízených jsou pomalejší. U metod s očíslováním je to naopak, jsou rychlé pro hledání podřízených a pomalé pro hledání nadřazených uzlů. Metoda s uložením vazeb do vlastní tabulky není nejrychlejší pro ani jednu skupinu dotazů, avšak pro menší kolekce není v obou skupinách ani nejpomalejší. Co testy spolehlivě vyloučily, je její použití pro rozsáhlé kolekce. V některých případech může být také na obtíž nemožnost procházet

uzly do hloubky. Výhodou je pak například možnost snadno přidávat k vazbám vlastní atributy a „čistota“ databázového návrhu. Vlastnosti metody s uloženou tabulkou vazeb ji i přes její nevýhody staví do role nejuniverzálnější metody z pěti testovaných.

Vzájemné srovnání dvou metod s ukládáním cesty ke kořeni nemá jednoznačně lepší metodu. Tradiční varianta je o něco efektivnější, ale postrádá možnost jednoduše pracovat s jednotlivými prvky cesty ke kořeni přímo prostředky SQL. Nevýhodou varianty se složeným typem je zase komplikovaná přenositelnost na jiné systémy. Při volbě mezi oběma metodami tak pravděpodobně častěji zvítězí tradiční varianta.

Ani mezi metodami s očíslováním nelze vybrat obecně lepší z obou metod. Vnořené množiny jsou sice ve všech testech o něco výkonnější, ale neobsahují informaci o hloubce uzlu. Očíslování uzlů použité v metodě s uloženým průchodem je o něco jednodušší a rozšířená varianta s mezerami v očíslování se lépe udržuje.

V kapitole 3 byly u každé metody řešeny možnosti aktualizací hierarchické struktury, což je další parametr, který ovlivňuje výběr vhodné metody. Časté aktualizace většinou diskvalifikují použití metod s očíslováním uzlů.

Kapitola 5

Závěr

Cílem práce byla analýza problému ukládání hierarchických biologických dat do relačni databáze. Celá práce se omezuje pouze na hierarchie, jejichž mapovaní na databázové tabulky je díky vlastnostem jazyka SQL netriviální.

Problém uložení dat lze rozdělit na dvě hlavní oblasti – teoretické rozebrání problému a praktickou implementaci se srovnáním různých metod uložení nad reálnými daty.

Teoretická část této práce se nejprve věnuje popisu základních pojmů, vlastnostem hierarchických struktur a problémům s jejich ukládáním do relační databáze. Následuje přehled všech známých metod pro ukládání hierarchií. V závěru teoretické časti jsou rozebrány způsoby uložení různých speciálních případů hierarchických struktur, které se ve skutečném světě vyskytují a je nutné se s nimi vypořádat. Hlavním přínosem této části by měl být ucelený český přehled včetně rozboru opomíjených vlastností, jejichž znalost je užitečná při reálném použití.

Popis jednotlivých metod je platný pro všechny hierarchické struktury a je tak použitelný i pro jiné obory než je bioinformatika.

Praktická část práce se pak zaměřuje konkrétně na biologická data. Pro měření a porovnání jednotlivých způsobů uložení bylo nutné nejprve najít vhodné zdroje dat. Následovala implementace nástroju pro převedení dat do databáze a implementace vhodného prostředí pro samotné měření.

To, že výsledky měření neurčily jednoznačně nejlepší metodu, jenom ilustruje složitost problematiky. Interpretace výsledku se tak omezuje na určení případů, pro které je každá metoda vhodná. Teoretické popisy jednotlivých metod a výsledky testů by měly minimálně poskytnout vodítko pro volbu nejvhodnějšího způsobu reprezentace hierarchických dat v konkrétní situaci.

Možné rozšíření práce

Nad rámec stanovených cílu by bylo možné práci rozšířit zejména o provedení dalších testů. Je možné zkoumat vliv různých nastavení databáze na efektvitu dotazování a nebo provést testy nad jinými databázovými implementacemi.

Také by bylo možné prakticky měřit možnosti paralelních aktualizací u jednotlivých metod. To je však relativně složitý problém, neboť paralelní přístup a způsob aktualizací může mít mnoho podob a vyžadoval by tak hlubší analýzu.

Práce se příliš nezaobírá proprietárními rozšířeními SQL pro rekurzivní dotazy. Ty by bylo možné prozkoumat hlouběji a prakticky porovnat jejich výkonnost s popsanými metodami.

Literatura

[1] CELKO, Joe. *Joe Celko's Trees And Hierarchies In SQL For Smarties.* San Fransisco: Morgan-Kaufmann, 2004

[2] MATOUŠEK, Jiří a NEŠETŘIL Jaroslav. *Kapitoly z diskrétní matematiky.* Praha: Karolinum, 2000

[3] MACKEY, Aaron. *Relational Modeling of Biological Data: Trees and Graphs* [cit. 1. 8. 2007] [www dokument]
Dostupný z:
`http://www.oreillynet.com/pub/a/network/2002/11/27/bioconf.html`

[4] FLING, Kirby. *Four ways to work with hierarchical data* [cit. 1. 8. 2007] [www dokument]
Dostupný z:
`http://www.evolt.org/node/4047`

[5] ČERMÁK Martin, DVOŘÁK Tomáš a RYBIČKOVÁ Alena. *Rekurzivní dotazy v SQL*, 2004 [cit. 1. 8. 2007] [dokument ve formátu ppt]
Dostupný z:
`http://www.ms.mff.cuni.cz/~jhum8111/DOTAZOVACI_JAZYKY/rekurze.ppt`

[6] DOUGAS, Kerry a DOUGLAS Susan. *PostgreSQL.* Indianapolis: Developer's Library, 2003

[7] POKORNÝ Jaroslav. *Dotazovací jazyky.* Praha: Karolinum, 2002

Příloha A

Obsah CD

Přiložený CD disk obsahuje všechny nástroje potřebné k zopakování testů. Na CD je také uložen text práce. Struktura adresářů je následující:

- `/text` – Obsahuje text práce ve formátu `pdf` a `ps`.
- `/sql` – Zde jsou SQL skripty, které vytvářejí tabulky k jednotlivým metodám uložení. Názvy sloupců tabulek jsou anglické, což odpovídá obvyklým konvencím.
- `/python` – Adresář obsahuje všechny skripty v jazyce Python. Přímo v samotném adresáři je pouze jednoduchý skript s konfigurací připojení k databázi. Další skripty jsou podle funkce členěny do podadresářů.
- `/python/parse` – Převod exportovaných dat z vlastních formátů do triviálního uložení
- `/python/convert` – Převod tabulky triviálního uložení do ostatních uložení.
- `/python/benchmark` – Nástroje pro porovníní metod.
- `/dat/export` – Testovací data ve vlastních formátech ve verzi použité pro testování.
- `/dat/sql` – Testovací data převedená skriptem do databázové tabulky a vyexportovaná jako seznam příkazů `INSERT`.

Příloha B

Skripty v jazyce Python

Všechny skripty využívají jednoduchý modul `dbconfig`, který inicializuje přípojení k databázi. Do kódu je nutné doplnit skutečné parametry připojení.

```
import pgdb
db = pgdb.connect(database='jméno databáze', host='localhost', user='uživatel')
```

Pro připojení k PostgreSQL je využívána standardní knihovna PyGreSQL.

Parametry připojení a adresáře s exportem dat jsou pro jednoduchost ve skriptech zadány jako konstanty, které je nutné měnit přímo ve zdrojovém kódu. Vzhledem k jednorázovému použití skriptů je zbytečné programovat složitější způsob zadávání parametrů.

B.1 Skripty pro parsování nativních formátů

Všechny tři skripty pro převod z nativního formátu do trviálního uložení začínají shodným kódem, který je z důvodu úspory místa ve výpisu vynechán. Kód inicializuje připojení k databázi a vyprázdní cílovou tabulku.

```
#!/usr/bin/env python

DUMP_DIR = '--CESTA-K-ADRESARI-S-EXPORTEM-';

import sys
sys.path = ['..'] + sys.path
from dbconfig import db
c = db.cursor()
c.execute('SET CONSTRAINTS trivial_parent_id_fkey deferred')
c.execute('TRUNCATE trivial')
```

Obdobně je v následujících skriptech vynecháno potvrzení transakce a uzavření spojení s databází.

```
db.commit()
db.close()
```

B.1.1 NCBI

```
infile = open(DUMP_DIR + 'nodes.dmp', 'r')

for line in infile:
    id, parent, _ = line.split('|',2)
    values = {'id': int(id), 'parent': int(parent)}
    if values['id'] == values['parent']:
        c.execute("INSERT INTO trivial (id, parent_id, name) values \
(%(id)d,NULL,NULL)",values)
    else:
        c.execute("INSERT INTO trivial (id, parent_id, name) values \
(%(id)d,%(parent)d,NULL)",values)

infile.close()

def assignName(id, name) :
    if id == None:
        return
    c.execute("UPDATE trivial SET name = %s WHERE id = %d", [ name, id ])

infile = open(DUMP_DIR + 'names.dmp', 'r')

lastId = None
bestName = None

for line in infile:
    id, name, _ , nameClass, _ = line.split('|',4)
    id = int(id)
    name = name.strip()
    nameClass = nameClass.strip()
    if id != lastId:
        assignName(lastId,bestName)
        lastId = id
        bestName = None
    if bestName == None or nameClass == 'scientific name':
        bestName = name

assignName(lastId,bestName)
infile.close()
```

B.1.2 SCOP

```
infile = open(DUMP_DIR + 'dir.hie.scop.txt_1.71', 'r')

for line in infile:
    if line[0] == '#':
        continue
    id, parent, _ = line.split()
    id = int(id)
    if id == 0:
        c.execute("INSERT INTO trivial (id, parent_id, name) values (1,NULL,'Root')")
    else:
        parent = int(parent)
        if (parent == 0):
            parent = 1
        c.execute("INSERT INTO trivial (id, parent_id, name) values (%d,%d,NULL)",[id, parent])

infile.close()
```

```
def assignName(id, name) :
    if id == None:
        return
    c.execute("UPDATE trivial SET name = %s WHERE id = %d", [ name, id ])


infile = open(DUMP_DIR + 'dir.des.scop.txt_1.71', 'r')

lastId = None
bestName = None

for line in infile:
    if line[0] == '#':
        continue
    tokens = line.split(None,4)
    id = int(tokens[0])
    name = tokens[4].strip()
    if len(name) > 100:
        name = name[0:97] + '...'
    assignName(id,name)

infile.close()
```

B.1.3 GO

```
infile = open(DUMP_DIR + 'gene_ontology.obo', 'r')

def insertTerm(term):
    if term == None or 'is_obsolete' in term:
        return
    #print term['id'],' is_a ',term['is_a']
    if len(term['is_a']) == 0:
        c.execute("INSERT INTO trivial (id, parent_id, name) values \
  (%d,1000000,%s)",[term['id'],term['name']])
    else:
        id = term['id']
        name = term['name']
        if len(name) > 94:
            name = name[0:94] + '...'
        for i in range(len(term['is_a'])):
            if (i > 0):
                copyid = i * 1000000 + id;
                fullname = name + ' ' + str(i)
            else:
                copyid = id
                fullname = name
            c.execute("INSERT INTO trivial (id, parent_id, name) values \
(%d,%d,%s)",[copyid, term['is_a'][i], fullname])

def toid(value):
    value = value.split('!',1)[0]
    return int(value.replace('GO:',''))

c.execute("INSERT INTO trivial (id, parent_id, name) values (1000000,NULL,'Root')")

term = None
for line in infile:
    line = line.strip()
    if len(line) == 0:
        continue
```

```
    if line[0] == '[':
        insertTerm(term)
        term = { 'is_a' : [] }
        if line != '[Term]':
            break; #typedef section reached
        continue
    if term == None:
        continue
    property, value = line.split(':',1)
    if (property == 'id'):
        term['id'] = toid(value)
    elif (property == 'name'):
        term['name'] = value.strip()
    elif (property == 'is_a'):
        term['is_a'].append(toid(value))
    elif (property == 'is_obsolete'):
        term['is_obsolete'] = True

infile.close()
```

B.2 Transformační skripty

Následující skripty převádějí triviální uložení na ostatní způsoby uložení. Stejně jako v předchozí části je u skriptů vynechána inicializační a ukončovací část. Ve zkrácených skriptech se předpokládá, že proměnné `c`, `cr` a `cw` jsou otevřené databázové kurzory. V ukončovací části se pak provádí potvrzení transakce a uzavření kurzorů. Skripty v kompletní podobě jsou k dispozici na CD disku přiloženém k práci.

B.2.1 Cesta ke kořeni

```
cw.execute('TRUNCATE path_enum')
cr.execute('SELECT * from trivial')

paths = { }
skipped = [ ]

while (1):
    row = cr.fetchone()
    if row == None:
        break
    id, parent, name = row
    if parent == None:
        paths[id] = ""
    else:
        if parent in paths:
            paths[id] = paths[parent] + str(parent) + '.'
        else:
            skipped.append(id)
            continue
    if parent == None:
        cw.execute('INSERT INTO path_enum (id, parent_id, name, path) VALUES \
            (%d, NULL, %s, %s)',[ int(id), name, paths[id] ] )
    else:
        cw.execute('INSERT INTO path_enum (id, parent_id, name, path) VALUES \
```

```
    (%d, %d, %s, %s)', [ int(id), int(parent), name, paths[id] ] )

def storeSkipped(id):
    cr.execute('SELECT * FROM trivial WHERE id = %d',[id])
    row = cr.fetchone()
    id, parent, name = row
    if parent not in paths:
        storeSkipped(parent)
        skipped.remove(parent)
    paths[id] = paths[parent] + str(parent) + '.'
    cw.execute('INSERT INTO path_enum (id, parent_id, name, path) VALUES \
       (%d, %d, %s, %s)', [ int(id), int(parent), name, paths[id] ] )

while (len(skipped) > 0):
    storeSkipped(skipped.pop(0))
```

B.2.2 Cesta ke kořeni jako složený typ

```
cw.execute('TRUNCATE path_array')
cr.execute('SELECT * from trivial')

paths = { }
skipped = [ ]

def arrString(id):
    return paths[id][:-1] + '}'

def storeSkipped(id):
    cr.execute('SELECT * FROM trivial WHERE id = %d',[id])
    row = cr.fetchone()
    id, parent, name = row
    if parent not in paths:
        storeSkipped(parent)
        skipped.remove(parent)
    paths[id] = paths[parent] + str(parent) + ','
    cw.execute('INSERT INTO path_array (id, parent_id, name, path) VALUES \
(%d, %d, %s, \''+ arrString(id)+ '\')', [ int(id), int(parent), name ] )

while (1):
    row = cr.fetchone()
    if row == None:
        break
    id, parent, name = row
    if parent == None:
        paths[id] = '{'
    else:
        if parent in paths:
            paths[id] = paths[parent] + str(parent) + ','
        else:
            skipped.append(id)
            continue
    if parent == None:

        cw.execute('INSERT INTO path_array (id, parent_id, name, path) VALUES \
          (%d, NULL, %s, \'{}\')', [ int(id), name ])
    else:
        cw.execute('INSERT INTO path_array (id, parent_id, name, path) VALUES \
          (%d, %d, %s, \''+ arrString(id)+ '\')', [ int(id), int(parent), name ])

while (len(skipped) > 0):
```

```
    storeSkipped(skipped.pop(0))
```

B.2.3 Tabulka vazeb

```
c.execute('TRUNCATE bindings')
c.execute('SELECT * from trivial where parent_id IS NULL')

dfs = []
dfs.append(c.fetchone()[0])

parents = []

while(len(dfs) > 0):
    id = dfs.pop()
    c.execute('SELECT parent_id from trivial where id = %d',[id])
    parent_id = c.fetchone()[0]
    while (len(parents) > 0 and parents[-1] != parent_id):
        del parents[-1]
    c.execute('SELECT * from trivial where parent_id = %d',[id])
    while (1):
        row = c.fetchone()
        if row == None:
            break
        dfs.append(row[0])
    depth = 1;
    for i in range(len(parents)-1,-1,-1):
        c.execute('INSERT INTO bindings (parent, child, depth) VALUES (%d,%d,%d)',
                  [ parents[i], id, depth ] )
        depth += 1
    parents.append(id)
```

B.2.4 Vnořené množiny

```
c.execute('TRUNCATE nested_sets')
c.execute('SELECT * from trivial where parent_id IS NULL')

open_nodes = {}
dfs = []
dfs.append(c.fetchone()[0])
ns_value = 1;

def insert(id):
    c.execute('SELECT * from trivial where id = %d',[id])
    row = c.fetchone()
    id, parent, name = row
    if parent == None:
        c.execute('INSERT INTO nested_sets (id, parent_id, name, ns_left, ns_right) \
  VALUES (%d, NULL, %s, %d, %d)', [ int(id), name, ns_value, 0 ] )
    else:
        c.execute('INSERT INTO nested_sets (id, parent_id, name, ns_left, ns_right) \
          VALUES (%d, %d, %s, %d, %d)',[ int(id), int(parent), name, ns_value, 0 ])

while(len(dfs) > 0):
    id = dfs[-1]
    if (id in open_nodes):
        del dfs[-1]
        del open_nodes[id]
        c.execute('UPDATE nested_sets SET ns_right = %d WHERE id = %d',[ns_value, id])
    else:
```

```
        insert(id)
        open_nodes[id] = True
        c.execute('SELECT * from trivial where parent_id = %d',[id])
        while (1):
            row = c.fetchone()
            if row == None:
                break
            dfs.append(row[0])
    ns_value += 1
```

B.2.5 Uložený průchod

```
c.execute('TRUNCATE flat')
c.execute('SELECT * from trivial where parent_id IS NULL')

dfs = []
dfs.append(c.fetchone()[0])

rank = 1;
depths = {}

def insert(id):
    c.execute('SELECT * from trivial where id = %d',[id])
    row = c.fetchone()
    id, parent, name = row
    if parent == None:
        c.execute('INSERT INTO flat (id, parent_id, name, rank, depth) VALUES \
  (%d, NULL, %s, %d, 0)', [ int(id), name, rank] )
        depths[id] = 0;
    else:
        depth = depths[parent] + 1;
        c.execute('INSERT INTO flat (id, parent_id, name, rank, depth) VALUES \
  (%d, %d, %s, %d, %d)', [ int(id), int(parent), name, rank, depth ])
        depths[id] = depth;

while(len(dfs) > 0):
    id = dfs.pop()
    insert(id)
    c.execute('SELECT * from trivial where parent_id = %d',[id])
    while (1):
        row = c.fetchone()
        if row == None:
            break
        dfs.append(row[0])
    rank += 1
```

B.3 Skripty pro porovnání metod

B.3.1 Testovací modul

```
#!/usr/bin/env python

import sys
sys.path = ['..'] + sys.path

from dbconfig import db
c = db.cursor()
```

```
import time
import random

def queryTime(query, params = [], debug = False):
    tBegin = time.time()
    c.execute(query, params)
    tRun = time.time()
    hashCode = 7
    if (debug):
        print ' result size ',c.rowcount
    while (1):
       row = c.fetchone()
       if row == None:
           hashCode += 1
           break
       #hashCode = hashCode * 23 + int(row[0])
       hashCode += int(row[0])
       if (debug):
           print '.... ',str(row)
    tRetrieve = time.time()
    return (tRun - tBegin, tRetrieve - tRun, hashCode % 10000)

def randomIdSeq(size):
    seq = []
    c.execute('select count(*) from trivial')
    rowCount = c.fetchone()[0]
    for i in range(size):
        offset = random.randint(0,rowCount-1)
        c.execute('select id from trivial limit 1 offset %d' % offset)
        seq.append(c.fetchone()[0])
    return seq;

#prevod jmen parametru na cislene indexy
def getParamIndexes(cursor, params):
    indexes = []
    for par in params:
        for i,desc in enumerate(cursor.description):
            if ('.' in par):
                #variant pro pair benchmark
                tok = par.split('.')
                if (desc[0] == tok[1]):
                    indexes.append([tok[0],i])
                    break
            if (desc[0] == par):
                indexes.append(i)
                break
    return indexes

def benchmark(benchIds, table, query, params, skip = 0, debug = False, customQuery = None):
    timeSum = [0,0,time.time(),0]
    hashCode = 3
    indexes = None
    preparedQuery = query
    for i,id in enumerate(benchIds):
        c.execute('select * from '+table+' where id = %d',[id])
        if (indexes == None):
            indexes = getParamIndexes(c,params)
        row = c.fetchone()
        if (debug):
            print '> ',str(row),
        if (customQuery):
```

```
            tBegin = time.time()
            preparedQuery = customQuery(query,row)
            tRun = time.time()
            timeSum[3] += tRun - tBegin
        queryParams = []
        for idx in indexes:
            queryParams.append(row[idx])
        t = queryTime(preparedQuery,queryParams,debug)
        hashCode = hashCode * 17 + t[2]
if i < skip:
  continue
        timeSum[0] = timeSum[0] + t[0]
        timeSum[1] = timeSum[1] + t[1]

    timeSum[2] = time.time() - timeSum[2]
    def itemround(item): return round(item,3)
    timeSum = map(itemround,timeSum)
    timeSum.append(hashCode % 10000)
    return timeSum

def printTimeSum(timeSum, header, query):
    (eTime, rTime, bTime, customTime, hashCode) = timeSum
    total = rTime + eTime
    print '-- table: ',header,' -------------'
    print 'query ',query
    print 'result hash ',hashCode
    print 'raw total time ',str(bTime)
    if (customTime > 0.0001):
        print 'custom function time ',str(customTime)
        total += customTime
    print 'retr time ',str(rTime)
    print 'exec time ',str(eTime)
    print 'total time ',str(total)
    print '---------------------------------------'
    return [eTime,rTime,total]


def benchPrint(benchIds, table, query, params, skip = 0, debug = False, customQuery = None):
    timeSum = benchmark(benchIds, table, query, params, skip,  debug, customQuery)
    return printTimeSum(timeSum, table, query)
```

B.3.2 Generátor identifikátorů

Tento skript se využívá při testování různých způsobů řazení. Aby všechny testy proběhly za stejných podmínek je nutné vygenerované identifikátory uložit do souboru. K tomu slouží standardní modul `pickle`.

```
#!/usr/bin/env python

from benchmark import *
from pickle import dump

skip = 100
ids = randomIdSeq(4100)
fout = open("ids.pickle",'w')
dump(skip,fout)
dump(ids,fout)
fout.close()
c.close()
```

B.3.3 Dotaz na nadřízené

```
from benchmark import *

skip = 50
ids = randomIdSeq(2050)

def pathEnumQuery(query,row):
    return 'SELECT * FROM path_enum WHERE id in (' + row[2].replace('.',',') + '0)'

benchPrint(ids,'path_enum',"?",[],skip,customQuery = pathEnumQuery)
benchPrint(ids,'path_array', \
  "SELECT * FROM path_array WHERE id = ANY(%s) ORDER BY \
  COALESCE(array_upper(path,1),0)", \
  ['path'],skip)
benchPrint(ids,'trivial', \
  "SELECT node.* FROM trivial AS node JOIN bindings AS b ON (node.id = b.parent) \
   WHERE b.child = %d ORDER BY depth DESC ", \
   ['id'],skip)
benchPrint(ids,'nested_sets', \
  "SELECT * FROM nested_sets WHERE ns_left < %d AND ns_right > %d ORDER BY ns_left", \
  ['ns_left','ns_right'],skip)
benchPrint(ids,'flat', \
  "SELECT f.* FROM flat AS f NATURAL JOIN \
    (SELECT max(rank) AS rank FROM flat WHERE rank < %d AND depth < %d GROUP BY depth) \
    AS values ORDER BY rank", \
    ['rank','depth'],skip)
```

B.3.4 Dotaz na podřízené

```
from benchmark import *

skip = 50
ids = randomIdSeq(2050)

def pathEnumQuery(query,row):
    return "SELECT * FROM path_enum WHERE path LIKE '" + \
      str(row[2]) + str(row[0]) + ".%%'"

def pathArrayQuery(query,row):
    pathid = row[2][0:-1] + ',' + str(row[0])
    return "SELECT * FROM path_array WHERE path BETWEEN '%s}' \
      AND '%s,99999999}'" % (pathid,pathid)

benchPrint(ids,'path_enum',"?",[],skip,customQuery = pathEnumQuery)
benchPrint(ids,'path_array',"?",[],skip,customQuery = pathArrayQuery)
benchPrint(ids,'trivial', \
  "SELECT node.* FROM trivial AS node JOIN bindings AS b ON \
  (node.id = b.child) WHERE b.parent = %d", \
  ['id'],skip)
benchPrint(ids,'nested_sets', \
  "SELECT * FROM nested_sets WHERE ns_left > %d AND ns_left < %d", \
  ['ns_left','ns_right'],skip)
benchPrint(ids,'flat', \
  "SELECT * FROM flat WHERE rank > %d AND rank < COALESCE( \
  (SELECT min(rank) FROM flat WHERE depth <= %d AND rank > %d) \
  ,9999999)", \
  ['rank','depth','rank'],skip)
```

B.3.5 Testování různých způsobů řazení

Použijí se tři sktipty velmi podobné B.3.4. Liší se pouze podmínkou pro řazení v dotazech a uvodním získáním náhodných identifikátorů. Ty jsou v úvodu načteny ze souboru do kterých byly uloženy pomocí B.3.2. Počátek všech tří skriptů tedy vypadá následovně:

```
from pickle import load

fin = open("ids.pickle",'r')
skip = load(fin)
ids = load(fin)
fin.close()
```