Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Zuzana Částková

Reprezentace XML dat pomocí C-store

Katedra softwarového inženýrství

Vedoucí diplomové práce: *prof. RNDr. Jaroslav Pokorný, CSc.*

Studijní program: *Informatika, I2-Sofwarové systémy, Databázové systémy*

Srdečné poděkování za pomoc v období vzniku práce patří:

- *prof. RNDr. Jaroslavu Pokornému, CSc.* za vedení diplomové práce a množství podnětných připomínek,
- *RNDr. Ireně Mlýnkové, Ph.D.* za poskytnutí sady reálných XML dat pro provedení experimentů,
- *Petru Částkovi* za propůjčení funkční implementace databáze C-store rovněž pro účely provedení experimentů.

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

V Praze dne *17. 04. 2009* *Zuzana Částková*

Obsah

Abstrakt

Název práce: *Reprezentace XML dat pomocí C-store*

Autor: *Zuzana Částková*

Katedra: *Katedra softwarového inženýrství*

Vedoucí diplomové práce: *prof. RNDr. Jaroslav Pokorný, CSc.*

e-mail vedoucího: *Jaroslav.Pokorny@mff.cuni.cz*

Abstrakt:

Prostředí C-store – tedy relační databáze ukládající záznamy na disku po sloupcích. Vhodná pro agregace, optimalizovaná pro čtení. Lze ji efektivně použít coby XML databázi? Práce zvažuje XML data se schématem i bez něj. Pro oba případy je definovaný databázový model vzniklý z XML souboru. Měřítkem vhodnosti modelu je možnost rychlého vyhledávání cest v XPath. Předek, potomek, mladší sourozenec – to jsou pouze některé z diskutovaných os XPath. Nízká systémová úroveň, umožňující odhadnout počet nutných skoků na disku, je pro algoritmy na jednotlivých cestách charakteristická. Výsledek? Schéma XML dat se neukazuje jako přínosné. Co se týká dat bez schématu, použití C-store se jeví jako optimální. Ale... Doplnění určitých funkčností systému by bylo i tady záhodno - přímý přístup k položkám, nečtení nutně celého sloupce. Práce rozšiřuje definici C-store o tyto principy a doplňuje to experimenty na reálných datech. V závěru je uvedena ukázka implementace vybraných algoritmů sloužící jako návod na další implementaci.

Klíčová slova: *XPath osy, vyhledávání cest v XML, databáze, ukládání C-store*

Abstract

Title: *Representing XML data by C-store*

Author: *Zuzana Částková*

Department: *Department of Software engineering*

Supervisor: *prof. RNDr. Jaroslav Pokorný, CSc.*

Supervisor's e-mail address: *Jaroslav.Pokorny@mff.cuni.cz*

Abstract:

C-store environment - relational database storing records on the disk by columns. Suitable for aggregations, optimalized for reading. Can it be effectively used as XML database? This thesis considers XML data with and without a schema. Database model is generated from a XML file for both cases. A measure of the model suitability is the possibility of making quick XPath queries. Ancestor, child, younger sibling - these are just some of the discussed XPath axes. Low level system enabling the estimation of number of jumps needed on drive is characteristic for algorithms for each axis. Result? Schema for XML data does not seem useful. Regarding the data without a schema, the use of C-store seems to be optimal. But ... Addition of certain functionality to the system would also be appropriate here - direct access to elements, possibility of not reading the whole column. The thesis extends the definition of C-store with these principles, and adds experiments with real data. The conclusion provides an example implementation of selected algorithms that might be used as guidance for further implementation.

Keywords: XPath Axes, querying XML, database, column-oriented storage

1 Úvod

Práce [1] popisuje architekturu relačního databázového systému, optimalizovaného pro čtení. Jedním z nejdůležitějších rozdílů navrhovaného systému oproti dnešním běžným SŘBD je ukládání dat do databáze po sloupcích místo po řádcích. Tento přístup umožní databázovému systému odpovídat na dotazy přečtením hodnot pouze v relevantních sloupcích, tedy ve sloupcích nutných pro zpracování dotazů. Systém tak není zatížen načítáním zbytečných dat do paměti.

Cílem této práce je ukázat, jak lze do takového úložiště uložit XML data s možností získávat rychle cesty v XML stromu. V práci jsou diskutované dva modely: model pro ukládání XML dat bez schématu a model pro data se schématem. Původní záměr byl neupřednostňovat žádný ze zmíněných modelů. V průběhu práce se ukázalo, že existence schématu není v tomto případě velkým přínosem. Navíc jsou XML data bez schématu mnohem častější. Proto byl v průběhu práce mírně změněn přístup. Diskutujeme sice oba modely, ale větší důraz je kladen na data bez schématu. Tento důraz se týká hlavně detailního rozpracování některých problémů a také praktických návodů na implementaci. V obou případech byl však zachován cíl ukázat nad zvoleným modelem algoritmy, které vyhledávají cesty v XML dokumentu a to podle jednotlivých os dokumentu.

Každý uvedený algoritmus je doplněný o výpočet jeho složitosti v kontextu určité abstrakce diskových operací. Za účelem vzájemného srovnání uvedených algoritmů jsou použity operace, jejichž reálná časová složitost je závislá jak na fyzických vlastnostech použitého disku, tak na samotné implementaci konkrétního databázového systému.

Organizace této práce je následující. Ve zbytku kapitoly0uvádíme základní principy úložiště C-store, objasnění důležitých pojmů a terminologickou domluvu, dále pak stručný úvod do XML a XPath dotazů. Kapitola 2 uvádí aspekty, ke kterým při srovnávání algoritmů a modelů budeme přihlížet. V této kapitole jsou rovněž uvedeny všechny abstrakce, které budeme používat při výpočtu časové složitosti algoritmů. V kapitolách 3 a 4 jsou popsány modely a algoritmy dvou základních skupin XML dat – data bez schématu (kapitola 3) a data se schématem (kapitola 4). Formulace algoritmů v kapitolách 3 a 4 je přizpůsobena účelu odhadu časové složitosti. Ukázky praktické implementace algoritmů s ohledem na spočítanou časovou složitost se nacházejí v kapitole 5 V této kapitole se rovněž nachází diskuse o případném rozšíření systému C-store za účelem zachování spočítané časové složitosti pro různé principy používané v uváděných algoritmech. Tato diskuse je nakonec podložena

provedením experimentů s náhodným dotazováním nad reálnými XML daty. V kapitole 6 jsou shrnuty výsledky práce společně s doporučeními pro další vývoj. Přílohou práce je CD se zprávou o provedených experimentech. Lze tam rovněž najít zdrojové a přeložené soubory s testy a sadu testovacích dat.

1.1 Terminologie a úvod do C-store

Většina dnešních relačních databázových systémů využívá model ukládání dat po řádcích. Tento přístup se ukazuje jako efektivní v databázových aplikacích typu OLTP, u nichž se počítá s častým zápisem. C-store (z angl. column oriented storage) je naopak úložiště dat ukládající data po sloupcích. Tato koncepce zaručuje efektivitu v aplikacích umožňujících časté dotazování nad velkým množstvím statických dat.

Představíme teď stručně v základních bodech architekturu sloupcově orientovaného úložiště dat C-store. Kromě ukládání dat po sloupcích místo po řádcích se C-store liší od běžných SŘBD také snahou co nejvíce používat kompresi a dekompresi provádět co nejpozději.

C-store, narozdíl od komerčních databázových systémů, nepoužívá indexy (a to včetně struktur optimalizovaných pro čtení, jako např. materializované pohledy nebo bitmapové indexy). Místo toho C-store ukládá kolekce sloupců setříděných podle některých atributů. Této kolekci se říká *projekce*. Jeden sloupec se může nacházet v několika projekcích (případně setříděných podle jiných atributů). Díky silně využívané kompresi duplicitní ukládání jednoho sloupce zaručí podporu pro setřídění dat podle různých atributů a to bez výrazného prostorového zatížení. Architektura C-store využívá redundanci uložených dat také pro zotavení.

Pro úplnost (i když mimo rozsah potřebný pro tuto práci) uvedeme ještě princip, jak se C-store vypořádává s praktickou potřebou transakčních aktualizací i v prostředí, kde je častější čtení. C-store implementuje dvě komponenty – malé úložiště umožňující rychlý zápis (WS) a mnohem větší úložiště optimalizované pro čtení (RS). Propojení těchto komponent zabezpečuje tzv. t*uple mover*, jehož úkolem je přesouvat záznamy v dávkách z WS do RS a efektivní metodou slévání tak vytvořit novou, aktualizovanou kopii RS. Takto navržená architektura by vyústila v mnoho konfliktů čtecích a zapisovacích zámků, kdyby je používala na zabezpečení transakcí. Proto C-store pracuje v tzv. *historickém* modu. V tomto modu jsou do odpovědi na dotaz zahrnuty pouze ty záznamy, jejichž časové razítko je starší než časové razítko naposledy provedené transakce. Takto systém zaručí správnost odpovědi v určitém

bodě historie.

Pro potřeby této práce můžeme vynechat popis implementačních detailů systému C-store. Důležitým však je popis datového modelu, nad kterým C-store pracuje.

Logický model – tabulky

C-store podporuje standardní logický model relačních databází, kde databáze sestává z množiny pojmenovaných *tabulek*. Definice každé tabulky je určena seznamem *atributů*, které mají své jméno a obor hodnot. Jeden atribut nebo množina atributů může tvořit *primární klíč* tabulky nebo *cizí klíč*, jakožto ukazatel na primární klíč jiné tabulky.

Fyzický model – projekce

C-store fyzicky neukládá přímo tabulky. Ukládá pouze *projekce*. Každá projekce je ukotvena k jedné logické tabulce T1. Projekce obsahuje alespoň jeden atribut tabulky T1. Navíc může projekce obsahovat libovolný počet atributů dalších tabulek. Jedinou podmínkou, aby se mohl atribut tabulky T2 nacházet v projekci ukotvené k tabulce T1, je vztah n:1 mezi entitami reprezentovanými řádky v tabulkách T1 a T2. Projekce má vždy stejný počet *záznamů* (řádků) jako tabulka, ke které je ukotvena.

Poznámka: C-store zavedl pojem projekce s mírně odlišným významem než je dnes běžně používaný. V této práci budeme standardně používat pojem projekce v konvenci C-store. V případech, kdy bude míněna projekce ve smyslu relační algebry, bude v textu použit výraz *operace projekce*.

Pro uložení projekce ukotvené k tabulce T aplikujeme *operaci projekce* nad touto tabulkou na odpovídající atributy, do výsledku zahrneme i duplicitní řádky a provedeme levé vnější spojení tabulky T s každou tabulkou, jejíž atributy se také mají nacházet v projekci, abychom získali hodnoty těchto atributů v každém řádku projekce.

Jako příklad uvedeme projekce nad dvěma tabulkami: Zaměstnanec (Jméno, Věk, Název_oddělení, Platová_třída) a Oddělení (Název, Patro). Možná množina projekcí v této databázi je například:

```
Zaměstnanec_1 (Jméno, Věk),
Zaměstnanec_2 (Název_oddělení, Věk, Oddělení.Patro),
Zaměstnanec_3 (Jméno, Platová_třída),
Oddělení (Název, Patro).
```

Příklad naplnění tabulek Zaměstnanec a Oddělení je vidět na obrázku 2. Na obrázku 1 je pak vidět odpovídající naplnění výše navržených projekcí.

Zaměstnanec

Jméno	Věk	Název_oddělení	Platová_třída
Marek	25	Účetní	A
Petr	31	Lidské zdroje	B
Karel	33	Účetní	B

Oddělení

Název	Patro
Účetní	1
Lidské zdroje	2

Obr. 2: Příklad naplnění logických tabulek

Zaměstnanec_1

Jméno	Věk
Marek	25
Petr	31
Karel	33

Zaměstnanec_2

Název_oddělení	Věk	Oddělení.Patro
Účetní	25	1
Lidské zdroje	31	2
Účetní	33	1

Zaměstnanec_3

Jméno	Platová třída
Marek	A
Petr	B
Karel	B

Oddělení

Název	Patro
Účetní	1
Lidské zdroje	2

Obr. 1 Příklad projekcí

Při sestavování fyzického datového modelu a při definici projekcí je zřejmě nutné pokrytí všech atributů všech tabulek.

Data každé projekce jsou fyzicky uložena po sloupcích. Jednu projekci si tedy můžeme představit jako *N* sloupců, kde *N* je počet atributů dané projekce. Každý z těchto sloupců má stejný počet řádků a je setříděn podle stejného klíče. Třídícím klíčem může být kterýkoliv sloupec v projekci. V definici projekce budeme pořadí jejich záznamů značit tak, že za seznamem atributů projekce následuje, oddělený svislou čarou, seznam klíčů (zleva doprava podle priority řazení). U výše definovaných projekcí je možné například toto řazení:

```
Zaměstnanec_1 (Jméno, Věk | Věk),
Zaměstnanec_2 (Název_oddělení, Věk,
Oddělení.Patro|Název_oddělení),
```

```
Zaměstnanec_3 (Jméno, Platová_třída | Platová třída, Jméno),
Oddělení (Název, Patro |Patro).
```

Pro názornost uvedeme pouze, jak by se změnil obsah projekce Zaměstnanec_3 (obrázek 3).

Zaměstnanec_3

Jméno	Platová třída
Marek	A
Karel	B
Petr	B

Obr. 3: Příklad řazení projekce

Fyzický model – mapovací tabulky

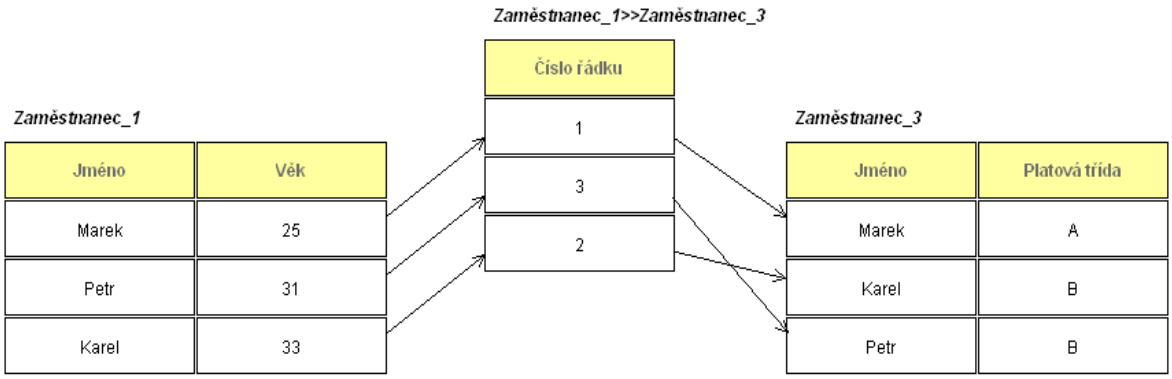
Aby bylo možné zodpovědět libovolný dotaz i bez fyzického uložení logických tabulek, musí systém C-store poskytnout nástroj pro rekonstrukci celých řádků každé tabulky na základě znalosti řádků jednotlivých projekcí. Tímto nástrojem jsou tzv. *mapovací tabulky* (angl. j*oin-indices*).

S mapovacími tabulkami úzce souvisí pojem *číslo řádku*. Data ve všech sloupcích každé projekce mají přiřazené *číslo řádku*, tedy pozici, na které se daná datová položka v rámci daného sloupce nachází. Data v různých sloupcích jedné projekce s odpovídajícím číslem řádku tedy vytváří logický řádek. Čísla řádku nejsou fyzicky v C-store uložena, jsou průběžně odvozována. Čísla řádku jsou pro každý sloupec přiřazována postupně od jedničky.

Nyní zavedeme mapovací tabulky. Nechť T1, T2 jsou dvě projekce ukotvené ke stejné tabulce T. Potom mapovací tabulkou T1>>T2 rozumíme tabulku: `T1>>T2 (číslo_řádku)`. Sémantika mapovací tabulky T1>>T2 je, že pro každý řádek *i* v projekci T1 je v *i-tém* řádku mapovací tabulky T1>>T2 uloženo číslo korespondujícího řádku *j* z projekce T2. Vzhledem k tomu, že projekce T1, T2 jsou ukotveny ke stejné tabulce, mapovací tabulka mapuje řádky projekce T1 a T2 ve vztahu 1:1. Jiný možný pohled na mapovací tabulky je ten, že mapovací tabulka T1>>T2 ukazuje, jak přeuspořádat řádky projekce T1 tak, aby měly stejné pořadí jako řádky projekce T2.

Jako příklad mapovací tabulky uvedeme mapovací tabulku **Zaměstnanec_1>>Zaměstnanec_3** (viz obrázek 4) mezi projekcemi:

```
Zaměstnanec_1 (Jméno, Věk | Věk),
Zaměstnanec_3 (Jméno, Platová_třída | Platová třída, Jméno).
```



Obr. 4: Ukázka mapovací tabulky

Mapovací tabulky sehrávají důležitou roli při rekonstrukci celých řádků logických tabulek, ale rovněž při optimalizaci některých dotazů.

V databázovém systému C-store je logický datový model shodný s modelem standardních relačních databází. Sestává z definicí tabulek a jejich atributů. Tyto tabulky však nejsou fyzicky uloženy v databázi. Databázové schéma je tvořeno pouze souhrnem definic projekcí (včetně jejich řazení) a mapovacích tabulek. Data jsou komprimovaná a ukládaná po sloupcích. Databázové schéma společně s daty tvoří kompletní databázi.

1.2 Kategorizace dotazů nad XML daty

Vzhledem k současné rozšířenosti problematiky XML technologií není třeba psát rozsáhlý úvod k tomuto tématu. Uvedeme pouze shrnutí těch faktů, které jsou klíčové pro tuto práci.

XML dokument můžeme chápat jako soubor dat uspořádaných do stromové struktury. Ve skutečnosti je stromová struktura podmínkou tzv. *dobře vytvořeného* XML dokumentu. V této práci budeme uvažovat pouze takto strukturovaná data. Jednotlivými uzly XML stromu jsou *elementy*, *atributy* a *znaková data*. Všechny ostatní prvky XML dokumentu (např. komentáře, entity, řídící instrukce) budeme pro účely této práce zanedbávat.

Důležitou vlastností XML dokumentu je (ne)existence schématu. XML data se schématem mají předem definovanou svoji strukturu, tedy vztahy mezi elementy, elementy a atributy a dále typová a integritní omezení. V této práci budeme zkoumat jak data bez schématu, tak i data se schématem. U definice schématu XML dat se však omezíme pouze na sílu jazyka

DTD, hlavně deklarace elementů a jejich podelementů a atributů.

Pro dotazování nad XML daty se využívají XPath dotazy. Ty jsou založeny na existenci různých relací mezi uzly XML dokumentu. Tyto relace se nazývají *osy* v XPath. Definujme *osu* σ jako množinu uspořádaných dvojic uzlů XML dokumentu. Nechť je dán uzel U. Říkáme, že uzel V leží na ose σ vzhledem k U právě když $\langle U,V \rangle \in \sigma$.

Bez újmy na obecnosti se dá nahlížet na vyhodnocení libovolného XPath dotazu jako na proces sestávající ze dvou kroků:

1) nalezení množiny *kandidátů* – tedy uzlů ležících na nějaké ose vzhledem k danému uzlu a
2) selekce z množiny *kandidátů* pouze na uzly splňující danou podmínku.

Časová složitost různých XPath dotazů tedy záleží na ose, na které se výsledek dotazu vzhledem k danému uzlu nachází. V této práci budeme pro jednotlivé osy zkoumat časovou složitost vyhledání množiny všech uzlů ležících na této ose vzhledem k danému uzlu. Budeme přitom diskutovat následující osy:

Předek – vzhledem k danému uzlu U na ní leží všechny uzly ležící na cestě od U (nepočítaje) ke kořenovému uzlu (včetně) v XML stromu.

Rodič – vzhledem k danému uzlu U na ní leží pouze první uzel ležící na cestě od U (nepočítaje) ke kořenovému uzlu (včetně) v XML stromu.

Potomek – vzhledem k danému uzlu U na ní leží všechny uzly, vzhledem ke kterým leží uzel U na ose předek.

Dítě – vzhledem k danému uzlu U na ní leží všechny uzly, vzhledem ke kterým leží uzel U na ose rodič.

Sourozenec – vzhledem k danému uzlu U na ní leží všechny uzly, které mají stejného rodiče jako U.

Mladší (starší) sourozenec – vzhledem k danému uzlu U na ní leží každý sourozenec, který předchází (následuje) uzel U při průchodu stromem doleva-do hloubky.

Předchůdce, následník – vzhledem k danému uzlu U na ní leží každý uzel, který předchází (následuje) uzel U při průchodu XML stromem doleva-do hloubky kromě jeho předků (potomků).

2 Kritéria pro srovnávaní algoritmů

Algoritmus mapování XML dokumentu na databázi typu C-store dostane na vstupu XML dokument (případně i jeho schéma) a jeho výstupem je kompletní databáze typu C-store, tedy databázové schéma společně s daty.

Pro účely této práce budeme předpokládat jednorázové vytvoření takové databáze a pak opakované vyhodnocování různých XPath dotazů. Základem pro ohodnocení mapovacího algoritmu tedy nebude jeho vlastní časová složitost, nýbrž časová složitost vyhodnocování jednotlivých XPath dotazů nad vytvořeným modelem.

Nejčastějším řešením srovnání různých modelů je specifikace, za jakých okolností je daný model vhodný a za jakých okolností vhodný není. V našem případě by to znamenalo odhad, jaké typy XPath dotazů budou nad daty nejčastěji vyhodnocovány. V databázi typu C-store však můžeme zkombinovat několik modelů tak, že výsledný model bude nabízet optimální vlastnosti pro efektivní vyhodnocení libovolného XPath dotazu, čímž získáme jeden univerzální model. Kombinace více modelů do jednoho má sice za následek redundantní uložení některých informací, nicméně koncepce databáze typu C-store je navržena tak, aby redundance dat neznamenala výraznou prostorovou expanzi.

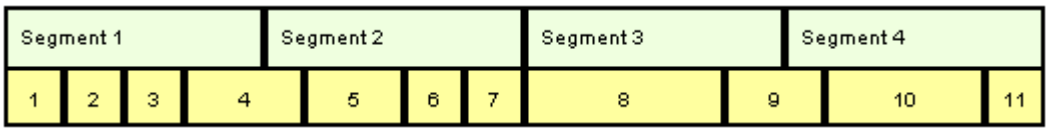
V této práci tedy uvedeme jeden model (optimální pro vyhodnocování všech typů XPath dotazů) pro XML data bez schématu a rovněž jeden model pro data se schématem. U obou modelů uvedeme diskusi jejích vlastností založenou na odhadech časové složitosti vyhodnocení dotazu pro jednotlivé typy XPath dotazů. Oba modely vzájemně porovnáme.

2.1 Abstrakce pro výpočet časové složitosti vyhodnocování dotazů

Detailní výpočet časové náročnosti jednotlivých algoritmů znamená analýzu až na úroveň diskových operací. Tento přístup vyžaduje znalost technických detailů implementace konkrétního úložiště dat. Pro potřeby srovnání algoritmů nezávisle na použité platformě je vhodnější abstrahovat od diskových operací a použít operátory, jejichž reálná časová náročnost je závislá na implementaci konkrétního úložiště dat a na fyzických vlastnostech použitého disku.

2.1.1 Linearizace disku a související operace

Použijme pro potřeby této práce představu disku jakožto lineární pásky sestávající z jednotlivých *pozic*. *Pozicí* budeme myslet adresu na disku, kde se nachází konkrétní (libovolně velká) datová položka. Kromě *pozic* budeme na pásce rozlišovat také *segmenty*. Ty jsou narozdíl od *pozic* pevné velikosti a jejich hranice nesouvisí s uloženými daty. V jednom *segmentu* může být uloženo více datových položek. Naopak, jedna datová položka může překračovat hranici *segmentu*. Příklad takové pásky je vidět na obrázku 5.

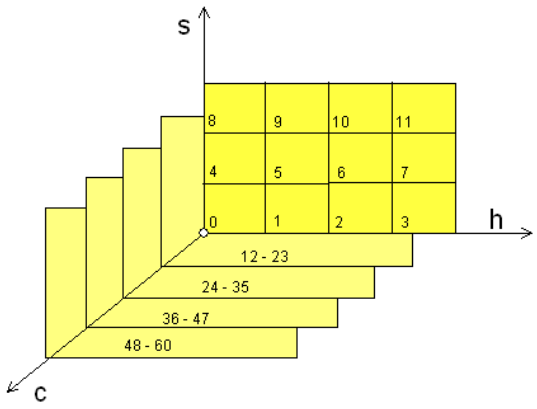


Obr. 5: Představa disku jako pásky sestávající z pozic (značeno žlutě) a segmentů (zeleně)

Ukážeme jeden ze způsobů, jakým lze vzájemně jednoznačně převést polohu na disku určenou cylindrem, hlavou a sektorem na *segment* na pásce. Nechť C je počet cylindrů disku, S je počet sektorů na jedné kružnici disku a H je počet hlav. Dále nechť $c \in \{0,\ldots,C-1\}$, $s \in \{0,\ldots,S-1\}$ a $h \in \{0,\ldots,H-1\}$. Uspořádaná trojice <c,s,h> tedy jednoznačně určuje polohu na disku. Každé poloze na disku přiřadíme její *segment* následujícím způsobem:

$$\mathbf{segment_{<c,s,h>} = c*(H*S) + s*H + h.}$$

Na obrázku 6 je znázorněn tento způsob linearizace adres na disku.



Obr. 6: Způsob číslování segmentů

Poznámka: Nyní jsme definovali *segmenty*. Jejich velikost odpovídá velikosti sektorů u skutečných disků. Přečtení libovolného *segmentu* v nejhorším případě znamená součet doby vystavení hlav z jednoho cylindru na druhý, doby otáčky disku a doby přenosu čtených dat. Při čtení několika *segmentů* za sebou bude potřebný čas kratší (např. až pro H*S *segmentů*

může stačit jedno vystavení hlav). Pro účely této práce jsme však segmenty zavedli pouze jako mezistupeň abstrakce od fyzických diskových operací k operacím pracujícím nad páskou s *pozicemi*.

Definujme nyní dvě základní abstraktní operace pracující nad páskou s pozicemi. Představme si, že nad páskou s pozicemi se volně pohybuje *čtecí hlava*. Ta může být nastavena nad libovolnou pozicí. Pozicí, nad kterou se momentálně čtecí hlava nachází, nazýváme *aktuální pozice* a můžeme přečíst její obsah. Nechť i je libovolná pozice na pásce a nechť j, k jsou pozice různé od i takové, že $|j-i| > 1$ a $|k-i| = 1$. Je-li i aktuální pozicí na pásce, přečtení pozice j nazýváme **skokem**, zatímco přečtení pozice k považujeme za **přečtení vedlejší** položky. Nechť i je aktuální pozice na pásce (libovolná). Pro porovnání časové složitosti jednotlivých algoritmů budeme používat abstrakci operace skoku a operace přečtení vedlejší položky:

`jump(j)` – přečtení náhodné pozice j z pásky,
`next()` – přečtení následující (resp. předchozí) pozice ($i \pm 1$) na pásce.

Operace `next()` má časovou náročnost závislou na aktuální pozici. Mohou nastat následující případy:

1) Položka na následující pozici se nachází celá ve stejném segmentu, jako konec položky na aktuální pozici. V tomto případě je ve skutečnosti pozice už přečtena a operace tedy trvá 0 časových jednotek.
2) Přečtení položky na následující pozici vyžaduje přečtení vedlejšího segmentu (jednou za P provádění operace `next()`, kde P je průměrný počet pozic v jednom segmentu). Aktuálním segmentem rozumějme segment určen trojicí <c, s,h>. Čtení vedlejšího segmentu může znamenat:
    a) pouze dobu přenosu dat, pokud následující segment je určen trojicí <c,s, h+1>,
    b) součet doby přenosu dat a doby odpovídající části otáčky disku, pokud následující segment je určen trojicí <c, s+1, 0> (jednou za H přečtení následujícího segmentu), nebo
    c) součet doby přenosu dat, doby odpovídající části otáčky disku a doby vystavení hlav z jednoho cylindru na druhý, pokud následující segment je určen trojicí <c+1, 0, 0> (jednou za H*S přečtení následujícího segmentu).

Operace `jump()` je časově náročnější. Nechť trojice <c, s, h> označuje segment obsahující aktuální pozici. Uvědomme si, že celkový počet segmentů je $C*H*S$. Na přečtení náhodné pozice bude nutná:

1) doba přenosu dat pro všechny pozice kromě těch, jež se nachází ve stejném segmentu, jako aktuální pozice (tedy kromě $\frac{1}{C*H*S}$ případů),
2) doba půlotáčky disku (jakožto průměrný úhel otočení) pro všechny pozice kromě těch,

nacházejících se v některém ze segmentů určených trojicí <*,s,*> (tedy kromě $\frac{1}{S}$ případů) a

3) doba vystavení hlav z jednoho cylindru na druhý pro všechny pozice, kterých segmenty se nenacházejí na stejném cylindru jako segment aktuální pozice (tedy kromě $\frac{1}{C}$ případů).

Ve skutečnosti je *pozice* neadresovatelnou položku disku – váže se k logické představě dat na disku. Na jedné *pozici* je uložena konkrétní hodnota jednoho atributu jednoho řádku tabulky. Může tedy jít o položku velkou mnoho bytů (např. řetězec), ale také jeden bit (např. true, false). Implementace mapování pozic na segmenty ovlivní dobu provádění operace `next()`, ale hlavně operace `jump()`.

V této práci budou operace `next()` a `jump()` základními operacemi pro určení časové složitosti jednotlivých algoritmů. Nyní ještě uvedeme základní funkce, které budeme používat ve všech zkoumaných algoritmech:

`SetProjection(P)` – nastavení projekce (nebo mapovací tabulky) P jako aktuální (až do dalšího nastavení se všechny operace odehrávají nad daty v této projekci).

`SetCol(C)` – nastavení sloupce C (v aktuální projekci) jako aktuální.

`Search(X)` – nalezení prvního výskytu prvku X v aktuální projekci, v aktuálním sloupci. Zároveň nastaví záznam, kde je prvek nalezen jako aktuální. Tato funkce vyžaduje několik provedení operace `jump()` nebo `next()` a to podle toho, zda je aktuální projekce primárně řazena podle aktuálního sloupce nebo ne.

`Read(I)` – přečtení I-teho záznamu v aktuální projekci, v aktuálním sloupci. Zároveň nastaví I-tý záznam jako aktuální. Tato funkce odpovídá přesně jednomu provedení operace `jump()`.

`Next()` – přečtení následujícího záznamu v aktuálním sloupci. Zároveň nastaví následující záznam jako aktuální. Tato funkce přesně odpovídá provedení operace `next()`.

`Previous()` – přečtení předchozího záznamu v aktuálním atributu. Zároveň nastaví předchozí záznam jako aktuální. Složitost této funkce je totožná s provedením operace `next()`.

`ReadCol(C)` – přečtení hodnoty ve sloupci C(ve stejné projekci) aktuálního záznamu.

Dva použité mapovací algoritmy v následujících kapitolách budou hodnoceny na základě toho, jak jsou odpovídající datové modely optimální pro vyhodnocování XPath dotazů. Při zkoumání vyhodnocování XPath dotazů se omezíme pouze na vyhledání všech uzlů ležících na zvolené ose vzhledem k danému uzlu. Algoritmy pro hledání uzlů na jednotlivých osách budou doplněny o výpočet jejich časové složitosti a to na úrovni abstraktních diskových operací `next()` a `jump()`.

3 Algoritmus pro XML data bez schématu

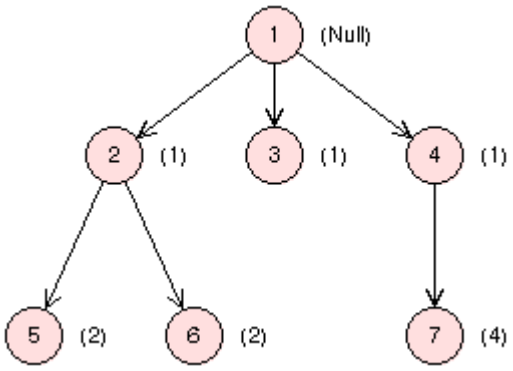
3.1 Datová struktura

Práce [2] popisuje mapování XML dokumentu do relační databáze pomocí metody *Structure-Centred mapping*. Základním prvkem této metody je znalost seznamu následníků pro každý uzel v XML stromu. V práci [2] je navrženo několik přístupů, jak tento seznam implementovat. Pro ukládání do úložiště C-store použijeme kombinaci dvou z těchto metod.

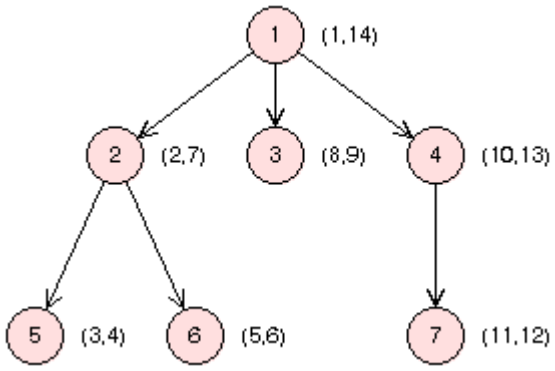
FK (Foreign Key) – metoda cizích klíčů. Tato metoda používá pro každý uzel XML stromu jednoznačný identifikátor. Pro každý uzel XML stromu je v tabulce uložen cizí klíč, jako odkaz na jeho rodiče (viz obrázek 7).

DF (Depth First) – metoda procházení XML stromu do hloubky. Tato metoda ke každému uzlu XML stromu ukládá dvojici hodnot (*min*, *max*), které jsou přidělovány průchodem stromu do hloubky. Při vstupu do uzlu je nastavena jeho hodnota *min* na aktuální hodnotu čítače, při vystoupení z uzlu se aktuální hodnota čítače nastaví jako hodnota *max* tohoto uzlu. Čítač je každým krokem zvyšován o jedničku (viz obrázek 8). Je-li *U* libovolný uzel XML stromu, značíme hodnoty *min* a *max* tohoto uzlu jako *min(U)* a *max(U)*.

```
<kniha id = 1>
  <kapitola id = 2>
    <sekce id = 5/>
    <sekce id = 6/>
  </kapitola>
  <kapitola id = 3/>
  <kapitola id = 4>
    <sekce id = 7/>
  </kapitola>
</kniha>
```



Obr. 7: Příklad FK přístupu – v závorkách uveden rodič



Obr. 8: Příklad DF přístupu – v závorkách dvojice hodnot (min, max)

3.1.1 Logický model

Uzly XML stromu budeme ukládat do logické tabulky:

```
Uzel (ID, Typ, Název, Hodnota, Rodič, Min, Max, Min_Rodiče),
```

kde *ID* je primárním klíčem, *Typ* určuje typ uzlu (element, atribut, text), *Název* jeho název (pouze u elementů a atributů, u textových dat je hodnota nastavena na null), *Hodnota* určuje hodnotu uzlu (u uzlů typu atribut a text, u elementů je hodnota nastavena na null). Atribut *Rodič* je cizím klíčem určujícím rodiče uzlu, *Min* a *Max* jsou hodnoty *min* a *max* DF metody, *Min_Rodiče* je hodnota *min* DF metody přiřazená rodiči uzlu.

Uvědomme si, že použitá logická tabulka je do určité míry redundantní. Dvěma způsoby implementuje seznam následníku uzlu ve stromu.

3.1.2 Fyzický model

Navrhněme nyní fyzický model, nad kterým budeme v úložišti C-store pracovat. XML strom je mapován do následujících projekcí:

```
FK1 (ID, Typ, Název, Hodnota, Rodič, Min| Rodič, Min)
DF  (ID, Typ, Název, Hodnota, Min, Max, Min_Rodiče| Min)
FK2 (ID, Rodič| ID)
```

Dále použijeme mapovací tabulky *FK2 >> DF*, *FK2 >> FK1*.

3.2 Získávání cest v XML stromu

V této kapitole popíšeme algoritmy získávání cest v XML stromu namapovaného na výše popsanou datovou strukturu. U každého algoritmu uvedeme také analýzu jeho časové složitosti. Jednotlivé algoritmy jsou zaměřeny na získávání určitého typu cest podle osy, na které výsledek XPath dotazu leží.

3.2.1 Osa dítě

Pro nalezení všech dětí daného uzlu využijeme sekvenčního čtení jednoho atributu, což je pro úložiště typu C-store nativní záležitost. Chceme vybrat všechny uzly, jejichž rodičem je daný uzel. Použijeme projekci *FK1*, která je setříděná podle atributu *Rodič*. Díky setřídění najdeme snadno první výskyt hledaného klíče a všechny další výskyty se nacházejí v sekvenci bezprostředně za ním. Přečtením odpovídající sekvence hodnot atributu *ID* dostaneme

výsledek dotazu.

Algoritmus 3.1 (Osa dítě)

Algoritmus dostane na vstup *id* daného uzlu a výstupem je množina *D* všech jeho dětí. Ta je na začátku prázdná.

Vstup: id
Výstup: D = {}

```
1       // Nalezení množiny odpovídajících záznamů
2       SetProjection(FK1);
3       SetCol(Rodič);
4       r = id; start = Search(id);
5       while r == id:
6            (r,stop) = Next();
7       stop = stop - 1;
8       // Operace projekce na odpovídající atribut
9       SetCol(ID);
10      D.add(Read(start));
11      while start < stop:
12           (d,start) = Next();
13           D.add(d);
```

Složitost algoritmu

Nechť *n* je počet všech uzlů XML stromu a *m* je počet uzlů nalezených na ose dítě. Vyhledání prvního dítěte, tedy provedení funkce *Search()* v atributu *Rodič,* má složitost *O(log n)*. Po nalezení prvního dítěte musíme najít všechny děti, což znamená *O(m)* provedení funkce *Next()*.

V následující tabulce je souhrnně uveden počet výskytů abstraktních diskových operací *next()* a *jump()*:

Operace	**Počet výskytů**
next()	O(m)
jump()	O(log n)

3.2.2 Osa potomek

Pro hledání všech potomků daného uzlu přístup FK vede na rekurzi operace spojení tabulek. Proto využijeme přístup DF a jeho matematické vlastnosti. Hledáme množinu všech uzlů,

které byly při průchodu XML stromem do hloubky navštíveny později než daný uzel, ale bylo z nich vystoupeno dřív než z daného uzlu. Platí tedy:

$$min(D) < min(P) \text{ a} \quad (1)$$

$$max(D) > max(P), \quad (2)$$

kde *D* je daný uzel a *P* je libovolný potomek daného uzlu.

Použijeme projekci *FK2* spolu s mapovací tabulkou *FK2>>DF* pro rychlé nalezení pozice daného uzlu v projekci *DF*. Jelikož tato projekce je setříděna podle atributu *Min*, víme, že všichni potomci daného uzlu se nacházejí za touto pozicí. Nakonec musíme podle atributu *Max* rozlišit opravdové potomky daného uzlu od uzlů, které splňují pouze podmínku *(1)* (uzly na ose následník).

Tvrzení 1

Nechť všechny uzly XML stromu jsou uloženy v poli *A* setříděném podle hodnoty *min* metody DF. Označme *poz(U)* pozici uzlu *U* v poli *A*. Nechť je dán uzel *D*. Potom pro všechny uzly *P* takové, že *poz(P) > poz(D)* platí:

Je-li *max(P)>max(D)*, neexistuje uzel P_1 takový, že $poz(P_1) > poz(P)$ a P_1 je potomkem *D*.

Důkaz:

Uvědomme si, jaké je pořadí uzlů v poli *A*. Při průchodu XML stromu do hloubky byly přiřazovány hodnoty *min* a *max* jednotlivých uzlů v tomto pořadí:

1) hodnota *min* daného uzlu *D*,
2) postupně všechny hodnoty *min* a *max* všech jeho potomků,
3) hodnota *max* daného uzlu *D*,
4) hodnoty *min* a *max* uzlů na ose následník vzhledem k *D*.

Uzly v poli *A* jsou řazeny podle hodnoty *min* DF metody, tedy v pořadí:

1) daný uzel *D*,
2) všichni jeho potomci,
3) uzly na ose následník vzhledem k *D*.

První uzel *P* splňující podmínky tvrzení je zjevně uzlem na ose následník vzhledem k danému uzlu *D* (splňuje podmínku *(1)* a nesplňuje podmínku *(2)*). Každý uzel P_1 nacházející se v poli

A za uzlem *P* je tedy také uzlem na ose následník vzhledem k danému uzlu *D* a tudíž nemůže být jeho potomkem. ■

Důsledkem tvrzení 2 je, že i když n-tice v projekci *DF* nejsou řazeny podle atributu *Max*, nemusíme testovat hodnotu *Max* všech uzlů s minimem větším jako *Min* daného uzlu. Jakmile totiž najdeme první uzel na ose následník, můžeme prohledávání ukončit, protože již určitě nenajdeme žádný uzel na ose dítě. Nalezení prvního uzlu, který už není potomkem daného uzlu je podmíněno tím, že přestane platit *(2)* (a to i v podle maxima nesetříděném poli)[1].

Algoritmus 3.2 (Osa potomek)

Algoritmus dostane na vstup *id* daného uzlu a výstupem je množina *P* všech jeho potomků. Ta je na začátku prázdná.

Vstup: id
Výstup: P = {}

```
1       // Nalezení daného uzlu
2       SetProjection(FK2);
3       SetCol(ID);
4       start = Search(id);
5       // Přes mapovací tabulku nalezení jeho pozice v projekci DF
6       SetProjection(FK2>>DF);
7       SetCol(position);
8       start = Read(start);
9       // Nalezení všech potomků jako sekvence start...stop
10      SetProjection(DF);
11      SetCol(Max);
12      Max = Read(start); m = Max;
13      while m <= Max:
14           (m,stop) = Next();
15      stop = stop -1;
16      // Projekce na odpovídající atribut
17      SetCol(ID);
18      Read(start); // neukladáme – start je pozice daného uzlu
19      while start < stop:
20           (p,start) = Next();
```

[1] Musíme však počítat s tím, že tato vlastnost není běžným databázím vlastní. Proto by do databáze musel být doimplementován modul umožňující využití této vlastnosti. Tento modul si můžeme představit jako klauzuli SQL dotazu `stop where`, určující podmínku, které platnost ukončí čtení a vyhledávání dalších záznamů v databázi. Bez této podpory musíme počítat s tím, že reálná složitost nalezení všech potomků daného uzlu je zatížená testováním všech záznamů, nalezených za daným uzlem, na invariant maxima (podmínka *(2)*). Další možností implementace bez klauzule `stop where` je použití kurzoru uložené procedury.

```
21              P.add(p);
```

Složitost algoritmu

Nechť *n* je počet všech uzlů XML stromu a *m* je počet uzlů nalezených na ose potomek. Pro nalezení daného uzlu je použita funkce *Search()*. Ta je prováděna na atributu *ID* projekce *FK2*, tedy na setříděné posloupnosti. Časová složitost této funkce je tedy *O(log n)*. Poté, co je nalezen daný uzel v projekci *FK2* a následně přes mapovací tabulku i v projekci *DF*, stojí nalezení všech jeho potomků *O(m)* provedení funkce *Next()*.

V následující tabulce je souhrnně uveden počet výskytů abstraktních diskových operací *next()* a *jump()*:

Operace	**Počet výskytů**
next()	O(m)
jump()	O(log n)

3.2.3 Osa rodič

Algoritmus pro vyhledání přímého rodiče daného uzlu je triviální. Používá jedinou projekci *FK2*. V ní logaritmickou složitostí vyhledá pozici daného uzlu a přečte hodnotu atributu *Rodič* na odpovídající pozici. Tento přístup však nevyužívá žádnou z vlastností sloupcového ukládání, naopak ukládání po řádcích by znamenalo menší počet diskových operací. Jelikož je však rodič právě jeden (i získání hodnoty null vyžaduje stejný počet diskových operací), není tento fakt tolik odrazující.

Algoritmus 3.3 (Osa rodič)

Algoritmus dostane na vstup *id* daného uzlu a výstupem je id jeho rodiče *id_r*.

Vstup: id
Výstup: id_r

```
1       // Nalezení daného uzlu
2       SetProjection(FK2);
3       SetCol(ID);
4       Search(id);
5       // Přečtení hodnoty v atributu Rodič
6       id_r = ReadCol(Rodič);
```

Složitost algoritmu

Nechť *n* je počet všech uzlů XML stromu. Časová složitost funkce *Search()* je *O(log n)* provedení abstraktní diskové operace *jump()*. Použití funkce *ReadCol()* místo *Read()* umožňuje prostor pro optimalizaci dotazů, které jsou nativně vhodnější pro databázi ukládanou po řádcích.

3.2.4 Osa předek

Ukládání tabulek po sloupcích je vhodné pro určité typy dotazů. Optimální způsob, jak přistupovat k datům uloženým po sloupcích je číst jeden atribut a na základě jeho hodnoty rozhodovat o výsledku dotazu. Jak bylo vidět v předchozích algoritmech, přínosem je pokud se množina odpovědí na dotaz nachází v sekvenci. V tomto případě je časová náročnost algoritmu shodná s časovou náročností nalezení začátku sekvence a projití právě množiny výsledných uzlů. Při čtení sekvence se už neprovádí žádné kroky navíc.

Zformalizujme nyní definici sekvence. Nechť $A = a_1,...,a_n$ je libovolná posloupnost. A nechť $M \subseteq \{a_1,...,a_n\}$. Potom M je *sekvencí* v A, právě když $\exists$ i $\leq$ n takové, že $\{a_i,...a_{i+|M|-1}\} = M$.

Tvrzení 3

Neexistuje algoritmus, který by mapoval uzly XML stromu (který dostal na vstupu) na takovou posloupnost *P*, aby pro libovolný uzel byla množina všech jeho předků sekvencí v *P* a přitom každý uzel XML stromu se v posloupnosti *P* nacházel právě jednou.

Důkaz:

Důkaz je triviální. Pokud by takový algoritmus existoval, musel by strom na obrázku 9 namapovat na posloupnost uzlů P tak, aby byly splněny všechny následující podmínky:

- {1,2} (množina všech předků uzlu 5) byla sekvencí v *P,*
- {1,3} (množina všech předků uzlu 6) byla sekvencí v *P,*
- {1,4} (množina všech předků uzlu 7) byla sekvencí v *P,*

- uzel 1 se v *P* nacházel právě jednou.

A taková posloupnost neexistuje. ■

Důsledkem tvrzení 5 je, že při hledání všech předků daného uzlu v XML stromu nelze využít výhody sekvenčního čtení (ani za použití jiné metody implementace seznamu následníků než *DF* nebo *FK*).

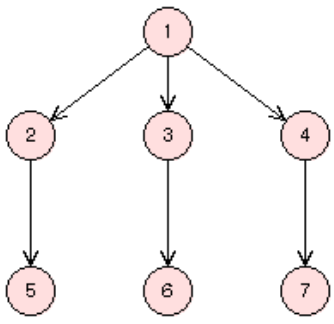
Nyní popíšeme dva algoritmy hledání všech předků daného uzlu. První je založen na *FK* přístupu, druhý je založen na metodě *DF*. Uvedeme diskusi o výhodách a nevýhodách obou algoritmů. Nakonec popíšeme algoritmus, který zkombinuje výhody obou přístupů a dosáhne tak lepších výsledků.

3.2.4.1 Algoritmus FK

Připomeňme, že metody FK i DF popisují způsob, jak pro stromovou strukturu implementovat seznam následníků. Metoda FK je založena na znalosti rodiče každého uzlu. Pokaždé, když jsme v předchozích kapitolách používali FK metodu, použili jsme atributy *ID* (jakožto primární klíč) a *Rodič* (jakožto cizí klíč). Nyní představíme jiný pohled na metodu FK. Jako primární klíč použijeme atribut *Min* a cizím klíčem bude atribut *Min_Rodiče*. Namísto projekce *FK1* nebo *FK2* pak použijeme projekci *DF*. Zřejmě, všechny principy FK metody zůstanou zachovány:

- atribut *Min* je opravdu jednoznačným identifikátorem uzlu,
- atribut *Min_Rodiče* obsahuje hodnotu atributu *Min*, nebo `null` (pokud uzel nemá rodiče), tedy je opravdu cizím klíčem.

Navíc je projekce *DF* řazena podle atributu *Min*. To má za následek, že pro vyhledání všech předků daného uzlu je použití metody FK nad projekcí *FK2* ekvivalentní s použitím metody FK nad projekcí *DF*. Jedna iterace vyhledání rodiče daného prvku, rodiče rodiče, atd. znamená v obou případech vyhledání v setříděném sloupci (*ID*, resp. *Min*) a přečtení hodnoty,



Obr. 9 K důkazu tvrzení 4

která bude vyhledávaná v příští iteraci (*Rodič* resp. *Min_Rodiče*).

Algoritmus 3.4 (Osa předek – FK metoda)

Algoritmus dostane na vstupu *id* daného uzlu, výstupem bude množina *P* všech jeho předků. Algoritmus použije projekci *FK2* a mapovací tabulku *FK2>>DF* pro vyhledání daného uzlu v projekci *DF*. Pak už použije výše popsanou metodu FK nad projekcí *DF*. Upřednostnění projekce DF před projekcí FK2, byť na první pohled nedůvodné, je základem pro snadnou integraci algoritmu s algoritmem *3.5 (Osa předek – DF metoda)* v kombinovaný algoritmus (viz kapitola 3.2.4.4 Kombinovaný algoritmus).

Vstup: id
Výstup: P = {}

```
1       // Nalezení daného uzlu
2       SetProjection(FK2);
3       SetCol(ID);
4       poz = Search(id);
5       // Skok do projekce DF
6       SetProjection(FK2>>DF);
7       SetCol(position);
8       poz = Read(poz);
9       SetProjection(DF);
10      SetCol(Min_Rodiče);
11      min_r = Read(poz);
12      // V cyklu hledání rodiče, rodiče rodiče atd.
13      while min_r is not null:
14           SetCol(Min);
15           poz = Search(min_r);
16           min_r = ReadCol(Min_Rodiče);
17           if min_r is not null:
18              P.add(ReadCol(ID));
```

Složitost algoritmu

Nechť *n* je počet uzlů XML stromu a *m* je počet uzlů nalezených na ose předek. Algoritmus nejprve nalezne daný prvek v projekci *FK2* pomocí funkce *Search()*. Následně se provedou dva skoky (abychom se dostali na sloupec *Min_Rodiče* projekce *DF*). Algoritmus je dále rekurzivní. V každé úrovni rekurze nejprve pomocí funkce *Search()* vyhledáme v setříděném sloupci *Min* projekce *DF* aktuální prvek. Poté přečteme ve sloupci *Min_Rodiče* té stejné projekce minimum rodiče tohoto prvku. Toto minimum se v příští úrovni rekurze stane

„aktuálním prvkem“. Nalezení nejbližšího předka tedy stojí *O(log n)* provedení operace *jump()*. Tento postup se opakuje právě *m*-krát.

V následující tabulce je souhrnně uveden počet výskytů abstraktních diskových operací *next()* a *jump()*:

Operace	**Počet výskytů**
next()	0
jump()	O(m*log n)

3.2.4.2 Algoritmus DF

Tento algoritmus využívá vlastnosti sekvenčního čtení. Sekvenční čtení aktuálního sloupce lze provést dvěma způsoby – pomocí funkce *Next()*, nebo pomocí funkce *Previous()*. Pro účely této práce považujeme oba přístupy za totožné. V praxi se funkce *Previuos()* dá snadno převést na funkci *Next()* a to vytvořením projekce řazené v opačném pořadí. V tomto algoritmu použijeme z důvodu konzistence s algoritmem *3.6 (Skoč a jdi)* funkci Previous().

Algoritmus 3.5 (Osa předek – DF metoda)

Algoritmus (stejně jako předchozí) dostane na vstupu *id* daného uzlu, výstupem bude množina *P* všech jeho předků. Algoritmus ale použije projekci *FK2* spolu s mapovací tabulkou *FK2 >> DF* pro vyhledání daného uzlu a projekci *DF* pro samotné nalezení všech předků.

Vstup: id
Výstup: P = {}

```
1       // Nalezení daného uzlu
2       SetProjection(FK2);
3       SetCol(ID);
4       stop = Search(id);
5       // Přes mapovací tabulku nalezení jeho pozice v projekci DF
6       SetProjection(FK2>>DF);
7       SetCol(position);
8       stop = Read(stop);
9       // Přečtení maxima daného uzlu
10      SetProjection(DF);
11      SetCol(Max);
12      Max = Read(stop);
```

```
13      // Test všech uzlů s hodnotou Min menší jako Min daného
14      // uzlu na podmínku Max
15      i = stop;
16      while i > 1:
17           (m,i) = Previous();
18           if m > Max:
19              P_poz.add(i);
20      // V množině P_poz se teď nachází seznam pozic,
21      // potřebujeme seznam ID
22      SetCol(ID);
23      i = stop; id = Read(i);
24      while i > 1:
25      (id,i) = Previous();
26           if i in P_poz:
27              P.add(id);
```

Složitost algoritmu

Nechť *n* je počet uzlů XML stromu a *m* je počet uzlů nalezených na ose předek. Nalezení daného uzlu, tedy provedení funkce *Search()* v atributu *ID* projekce *FK2,* má složitost *O(log n)* skoků na disku. Po nalezení daného uzlu musíme nejdříve provést O(1) skoků na disku, abychom se přes mapovací tabulku *FK2 >> DF* dostali na sloupec *Max* projekce *DF*, přečetli hodnotu *Max* daného uzlu. Nakonec algoritmus čte nanejvýš *n* prvků v sekvenci, což znamená *O(n)* provedení operace *next()*.

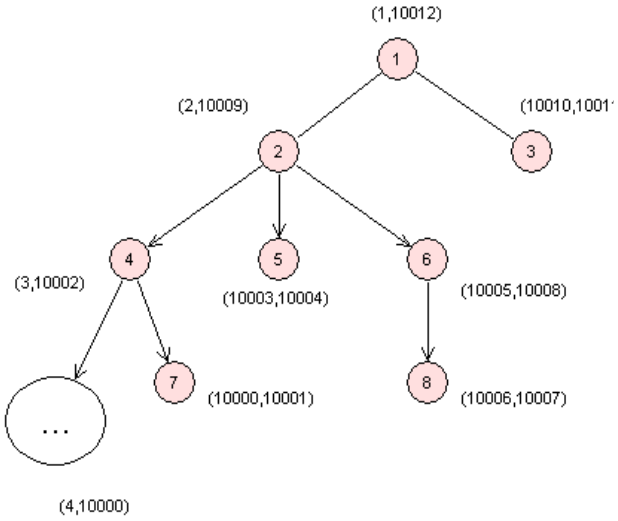
V následující tabulce je souhrnně uveden počet výskytů abstraktních diskových operací *next()* a *jump()*:

Operace	**Počet výskytů**
next()	O(n)
jump()	O(log n)

3.2.4.3 Srovnání algoritmů FK a DF pro dotazy na ose předek

Jak bylo vidět v předchozích sekcích, oba algoritmy mají stejnou časovou složitost pro nalezení daného uzlu. Budeme se nyní proto zabývat pouze porovnáním časové složitosti nalezení množiny předků daného uzlu.

Algoritmus 3.4 (Osa předek – FK metoda) sestává z mnoha skoků na disku. Těchto skoků se provede *O(m*log n)*, kde *m* je počet předků daného uzlu a *n* je počet všech uzlů XML stromu.



Obr. 11 V závorkách hodnoty min a max DF metody

	Uzel	Min
A	1	1
	2	2
	4	3
B	⋮	⋮
	7	10000
	5	10003
	6	10005
	8	10006
	3	10010

Obr. 10 DF řazení stromu z obrázku Obr. 11

Výhodou je, že algoritmus nečte žádná zbytečná data. Nevýhodou je, že čtené položky nejsou na disku seřazeny sekvenčně, tedy přečtení každé z nich znamená skok.

Naopak *Algoritmus 3.5 (Osa předek* – DF metoda) přečte více záznamů (i ty, které se do výsledku nedostanou), ale výhodou je, že tyto záznamy jsou na disku uloženy v sekvenci. U tohoto algoritmu se přečte *O(n)* položek seřazených v sekvenci za sebou.

Vhodnost jednoho nebo druhého algoritmu lze posuzovat na základě dvou faktorů – „*hustota*“ sekvence přečtené metodou DF a technické parametry disku. Hustota sekvence určuje, kolik z dat v sekvenci je relevantních pro výsledek dotazu. Čím větší je hustota sekvence, tím více se vyplatí použití metody DF. Se snižováním hustoty sekvence je metoda DF méně a méně efektivní, až v určitém momentě (v závislosti na technických parametrech disku) bude provedení *m* skoků trvat kratší dobu než provedení *n* operací *next()*.

Uvědomme si, že hustota sekvence přečtené DF metodou může být různá v různých částech sekvence (viz obrázek 10, 11). Uzel *7* má za svoje předky uzly *1*, *2*, *4*. Metoda DF však zařadí před uzel *7* také všechny prvky podstromu „ ... “. Všechny tyto uzly tvoří sekvenci přečtenou metodou DF. Zatímco v části *A* této sekvence je hustota vysoká (všechny uzly v části *A* jsou relevantní pro výsledek dotazu), hustota sekvence v části B je nízká (v této části se nenachází žádný uzel relevantní pro výsledek dotazu).

Znalost hustoty různých částí sekvence uzlů s minimem menším jako daný prvek vede na myšlenku zkombinovat pro vyhledání všech předků daného uzlu přístupy FK a DF. V těch částech sekvence, kde je hustota nízká, se použije *Algoritmus 3.4 (Osa předek* – FK metoda). Naopak, v částech sekvence s vysokou hustotou se aplikuje *Algoritmus 3.5 (Osa předek* – DF

metoda). V příkladě z obrázků 10, 11 bychom použili metodu FK pro vyhledání uzlu *4* a metodu DF pro vyhledání uzlů *1* a *2*. Nutnou podmínkou pro realizaci kombinovaného algoritmu je znalost hustoty dané části sekvence aniž bychom ji museli přečíst. Jinak by totiž algoritmus nedosahoval vyšší efektivity než samotný *Algoritmus 3.5 (Osa předek – DF metoda)*. V následující sekci popíšeme kombinovaný algoritmus detailně včetně implementace rozhodování, který ze dvou přístupů se má použít.

3.2.4.4 Kombinovaný algoritmus – „Skoč a jdi“

V předchozí sekci jsme uvedli způsob, jak zkombinovat *Algoritmus 3.4 (Osa předek – FK metoda)* a *Algoritmus 3.5 (Osa předek – DF metoda)* na základě znalosti hustoty určité části sekvence přečtené DF metodou. Nyní tento algoritmus popíšeme detailněji. Jeho název reflektuje pohyb, který algoritmus připomíná – skákání střídané s chůzi po krocích.

Algoritmus *„Skoč a jdi“* je založen na rekurzivním hledání rodiče daného prvku, rodiče rodiče, atd. Rozdíl oproti FK metodě je v tom, že kombinovaný algoritmus se u každé iterace dynamicky rozhodne, zda rodiče aktuálního prvku vyhledá za pomocí binárního hledání (operace *Search()*) nad setříděným sloupcem *Min* (skákání), nebo pomocí přečtení sekvence několik prvků nacházejících se v projekci *DF* bezprostředně před daným uzlem (chůze). Jak jsme již uvedli v předchozí sekci, skákání se vyplatí tam, kde je hustota sekvence nízká a naopak při vysoké hustotě sekvence se vyplatí chůze. Nyní zavedeme metriku pro měření hustoty dané části sekvence.

Uvědomme si, že část sekvence, o které rozhodujeme, obsahuje uzly nacházející se v projekci *DF* mezi rodičem aktuálního uzlu (včetně) a aktuálním uzlem (mimo něj). Například na obrázcích 10, 11 v případě hledání rodiče uzlu *7* rozhodujeme o části sekvence obsahující celou oblast *B* spolu s uzlem *4*. Daná část sekvence tedy obsahuje několik (≥ 0) uzlů pro výsledek dotazu nerelevantních a právě jeden uzel (rodič aktuálního uzlu) relevantní pro výsledek dotazu. Počet nerelevantních uzlů je tedy pro hustotu této části sekvence určující. Tento počet odpovídá právě počtu uzlů nacházejících se v projekci *DF* mezi aktuálním uzlem a jeho rodičem.

Tvrzení 6

Nechť všechny uzly XML stromu jsou uloženy v poli *A* setříděném podle hodnoty *min* metody DF. Nechť je dán uzel *U* a jeho rodič *R*. Označme *p* počet uzlů nacházejících se v poli *A* mezi uzly *R* a *U*. Platí:

$$p = \frac{min(U) - min(R) - 1}{2} \tag{3}$$

Důkaz:

Při průchodu XML stromu do hloubky byly přiřazovány hodnoty *min* a *max* jednotlivých uzlů v tomto pořadí:

1) hodnota *min* uzlu *R*,
2) postupně všechny hodnoty *min* a *max* všech uzlů ležících mezi uzly *R* a *U*,
3) hodnota *min* uzlu *U*.

Při průchodu stromu do hloubky se čítač DF metody mezi přiřazením *min(R)* a *min(U)* zvedl o $2 \times p$. Platí tedy:

$$min(R) + 2 \times p + 1 = min(U) \tag{4}$$

Triviální ekvivaletní úpravou rovnice *(5)* dostaneme vztah *(6)*. ■

Důsledkem tvrzení 7 je, že pro určení hustoty nejbližší aktuální sekvence potřebujeme znát pouze hodnotu *min* aktuálního uzlu a hodnotu *min* jeho rodiče. Obě tyto hodnoty jsou v projekci *DF* uloženy, tedy sekvenci nemusíme číst a její hustotu známe v jednotkovém čase.

Abychom dokázali pro danou část sekvence určit, jestli se vyplatí skákání nebo chůze, musíme porovnat jejich časové nároky. K tomuto účelu potřebujeme přesně znát, jak fungují.

Skákání funguje principiálně stejně jako *Algoritmus 3.4 (Osa předek – FK metoda)*. Za předpokladu, že jsme právě přečetli hodnoty atributů *Min* a *Min_Rodiče* aktuálního uzlu, použijeme pouze jeho odpovídající část – nalezení pozice, na které se hledaný rodič nachází. Pak už následuje nová iterace, tedy opět rozhodování, který přístup aplikujeme.

Chůze je založena na *Algoritmus 3.5 (Osa předek – DF metoda)*. Nejdříve přečte hodnotu atributu *Max* daného uzlu a pak čte hodnoty atributu *Max* u všech uzlů, které se nacházejí mezi daným uzlem a jeho rodičem (tyto uzly mají hodnotu *Max* menší jako daný uzel). Rodiče jsme nalezli ve chvíli, kdy hodnota atributu *Max* je větší než *Max* daného uzlu[2] a v tuto chvíli prohledávání ukončíme. Následuje nová iterace celého algoritmu.

Nechť *n* je počet všech uzlů XML stromu. Mějme aktuální uzel *U*, jeho minimum *min(U)*

[2] Namísto vyhledávání prvku na základě hodnoty atributu *Max* by bylo přirozenější použít vyhledávání na základě hodnoty atributu *Min*. Projekce *DF* je však seřazena podle atributu *Min*. Na aplikační úrovni by proto v praxi nebylo možné ovlivnit způsob vyhledávání nad tímto sloupcem (defaultně by se použilo binární vyhledávání).

a minimum jeho rodiče *min_r(U)*. Nalezení rodiče uzlu *U* metodou skákání trvá:

$$log(n) \times t_s, \tag{7}$$

kde t_s je průměrný čas potřebný na provedení skoku na disku (viz kapitola *2.1.1 Linearizace disku a související operace*). V praxi můžeme metodu skákání dokonce vylepšit tak, že vyhledáváme pouze v části projekce *DF* před aktuálním prvkem (viz kapitola 3.2.4.2 *Algoritmus DF*). Číslo *n* by bylo tedy každým krokem nižší. Nalezení rodiče uzlu *U* metodou chůze trvá (viz tvrzení 8):

$$\left(\frac{min(U) - min\_r(U) - 1}{2} + 1 \right) \times t_n, \tag{8}$$

kde t_n je průměrný čas operace *next()* (viz kapitola *2.1.1 Linearizace disku a související operace*). Čísla t_s a t_n jsou konstantní pro zvolený disk, číslo *n* je konstantní pro každý XML soubor (případný zápis do databáze číslo *n* změní). Porovnáním hodnot *(9)* a *(10)*, zjistíme, který přístup je pro aktuální část sekvence vhodnější:

- je-li menší hodnota *(11)*, aplikujeme metodu skákání,
- je-li menší hodnota *(12)*, aplikujeme metodu chůze.

Nakonec uvedeme formální popis kompletního algoritmu „Skoč a jdi“.

Algoritmus 3.6 (Skoč a jdi)

Algoritmus dostane na vstupu *id* daného uzlu, výstupem bude množina *P* všech jeho předků.

Vstup: id
Výstup: P = {}

```
1       // Nalezení daného uzlu
2       SetProjection(FK2);
3       SetCol(ID);
4       poz = Search(id);
5       // Skok do projekce DF
6       SetProjection(FK2>>DF);
7       SetCol(position);
8       poz = Read(poz);
9       SetProjection(DF);
10      SetCol(Min_Rodiče);
11      min_r = Read(poz);
12      // V cyklu hledání rodiče, rodiče rodiče atd.
13      while min_r is not null:
```

```
14      // Rozhodnutí, který přístup se v nejbližší iteraci použije
15      min = ReadCol(Min);
16      if log(konst.n)*konst.ts <=((min – min_r – 1)/2 + 1)*konst.tn:
17          // Skákání
18          SetCol(Min);
19          poz = Search(min_r);
20          P.add(ReadCol(ID));
21      else:
22          // Chůze
23          m = max = ReadCol(Max);
24          SetCol(Max);
25      while poz > 1 and m <= max:
26              (m,poz) = Previous();
27              if m > max:
28                    P.add(ReadCol(ID));
29          SetCol(Min_Rodiče);
30      min_r = Read(poz);
```

3.2.5 Osa sourozenec

Pro nalezení všech sourozenců daného uzlu opět využijeme vlastnost sekvenčního čtení jednoho atributu. Použijeme projekci *FK2*, ve které najdeme hodnotu atributu *Rodič* daného uzlu. Sourozenci jsou uzly, které mají tuto hodnotu stejnou. Za použití mapovací tabulky *FK2>>FK1* najdeme v projekci *FK1* výskyt daného uzlu. Díky druhotnému řazení záznamů podle atributu *Min* víme, že sekvence prvků se stejnou hodnotou atributu *Rodič* nacházející se před daným uzlem v projekci *FK1* odpovídá jeho mladším sourozencům a sekvence za tímto prvkem odpovídá jeho starším sourozencům.

Algoritmus 3.7 (Osa sourozenec)

Algoritmus dostane na vstup *id* daného uzlu a výstupem je množina *M* všech jeho mladších sourozenců a množina *S* všech jeho starších sourozenců. Obě jsou na začátku prázdné.

Vstup: id
Výstup: M = {}, S = {}

```
1       // Nalezení daného uzlu přes index
2       // Nalezení v FK2
3       SetProjection(FK2);
4       SetCol(ID);
5       start_stop = Search(id);
6       // Join-index
```

```
7       SetProjection(FK2>>FK1);
8       SetCol(position);
9       start_stop = Read(start_stop);
10      // FK1
11      SetProjection(FK1);
12      SetCol(Rodič);
13      rodič = Read(start_stop);
14      // Nalezení množiny M = [start, ...start_stop]
15      r = rodič;
16      while r == rodič:
17           (r,start) = Previous();
18      start = start + 1;
19      // Nalezení množiny S = [start_stop, …, stop]
20      r = rodič; Read(start_stop);
21      while r == rodič:
22           (r,stop) = Next();
23      stop = stop - 1;
24      // Projekce na odpovídající atribut
25      SetCol(ID);
26      i = Read(start);
27      while start < stop:
28           if start < start_stop:
29              M.add(i);
30           if start > start_stop:
31              S.add(i);
32           (i,start) = Next();
```

Složitost algoritmu

Nechť *n* je počet všech uzlů XML stromu a *m* je počet uzlů nalezených na ose sourozenec. Operace *Search()* má složitost *O(log n)*. Po nalezení daného prvku v projekci *FK1* je časová složitost selekce všech sourozenců *O(m)* operací *Next()/Previous()*.

V následující tabulce je souhrnně uveden počet výskytů abstraktních diskových operací *next()* a *jump()*:

Operace	**Počet výskytů**
next()	O(m)
jump()	O(log n)

3.2.6 Osa mladší (starší) sourozenec

Algoritmus 3.7 (Osa sourozenec), který jsme použili v předchozí kapitole, nalezl všechny sourozence daného uzlu jako sjednocení všech mladších a starších sourozenců. Triviální úpravou tohoto algoritmu můžeme výslednou množinu omezit pouze na mladší (resp. starší) sourozence. Časová složitost algoritmu se také nezmění.

3.2.7 Osa následník (předchůdce)

Nechť je daný uzel *U* XML stromu. Pokud pro libovolný uzel *V* platí:

$$min(V) < min(U), \tag{13}$$

je *V* buď předkem nebo předchůdcem uzlu *U*. Rozdíl mezi předkem a předchůdcem je v hodnotě *max*. Zatímco předchůdci mají hodnotu *max* menší jako uzel *U*, předci mají hodnotu *max* větší než uzel *U*. Aby byl uzel *V* předchůdcem uzlu *U*, musí kromě podmínky *(14)* navíc platit následující podmínka:

$$max(V) < max(U). \tag{15}$$

Naopak, pokud pro libovolný uzel *V* platí:

$$min(V) > min(U) \tag{16}$$

a současně:

$$max(V) > max(U), \tag{17}$$

je uzel *V* následníkem uzlu *U*.

Všimněme si, že hledání výsledné množiny *M* je v podstatě rozdílem množin. Nechť *Min_inv* je množina prvků, které splňují invariant minima – pro předchůdce podmínka *(18)* (mají *min* menší než daný uzel), pro následníky podmínka *(19)* (*min* větší než daný uzel). Dále *Max_not* je množina uzlů, které nesplňují invariant maxima (podmínka *(20)*, resp. *(21)*). *Max_not* je množina předků (pokud jde o hledání předchůdců) nebo potomků (pokud jde o hledání následníků). Výsledkem dotazu je pak množina:

$$M = Min\_inv \setminus Max\_not.$$

Uvedeme nyní univerzální algoritmus na hledání jak předchůdců, tak následníků. Algoritmus použije projekci *DF* na hledání výsledné množiny. Pro nalezení daného uzlu použije projekci *FK2* a mapovací tabulku *FK2 >>DF*.

Algoritmus najde všechny předchůdce nebo všechny následníky, podle toho, která dvojice operací *Init* a *MaxInvariant* se použije.

Předchůdce	**Následník**
`Init (pozice)` `  return (1, pozice)`	`Init (pozice)` `  return (pozice + 1, N+1)`
`MaxInvariant (max)` `  return max < MAX`	`MaxInvariant (max)` `  return max > MAX`

Operace Init() implementuje invariant minima – vrací dvojici (start, stop) v projekci FK1
N je počet všech uzlů XML stromu,
MAX je hodnota atributu Max daného uzlu.

Algoritmus 3.8 (Osa následník, předchůdce)

Algoritmus dostane *id* daného uzlu a vrátí množinu *M* všech předchůdců (resp. následníků) tohoto uzlu. Tato množina je na začátku prázdná.

Vstup: id
Výstup: M = {}

```
1       // Nalezení daného uzlu přes index
2       // Nalezení v FK2
3       SetProjection(FK2);
4       SetCol(ID);
5       pozice = Search(id);
6       // Použití join indexu
7       SetProjection(FK2>>DF);
8       SetCol(position);
9       pozice = Read(pozice);
10      // Nalezení množiny těch prvků, které do výsledku nepatří
11      SetProjection(DF);
12      SetCol(Max);
13      MAX = Read(pozice);
14      (start, stop) = Init(pozice); Max_not = {}; i = start;
15      m = Read(i);
16      while i < stop:
17           if not MaxInvariant(m):
18                Max_not.add(i);
19           (m,i)Next();
20      // Nalezení množiny výsledků
21      SetCol(ID);
22      i = start; id = Read(i);
23      while i < stop:
```

```
24            if i not in Max_not[3]:
25               M.add(id);
26            (id,i) = Next();
```

Složitost algoritmu

Nechť *n* je počet všech uzlů XML stromu, *m* je počet výsledků dotazu a *p* je počet uzlů v množině *Max_not*. Pro vyhledání daného uzlu je potřeba *O(log n)* skoků na disku. Pro samotné nalezení množiny výsledků algoritmus provede ještě *2*(*m+p+1) funkcí *Next()*.

V následující tabulce je souhrnně uveden počet výskytů abstraktních diskových operací *next()* a *jump()*:

Operace	**Počet výskytů**
next()	O(m + p)
jump()	O(log n)

Možné optimalizace

Vyjdeme-li z řazení uzlů v projekci *DF*, které je podrobně popsáno v kapitole *0*

Osa potomek, můžeme algoritmus upravit tak, aby v případě hledání následníků se operace *Next()* provedla místo *2*(m+p)* krát pouze *(m+p)* krát.

V algoritmu *3.8 (Osa následník, předchůdce)* nahradíme část hledání prvků, které do výsledku nepatří (řádky 10-19), pouhým průchodem sekvence potomků (tedy prvků bezprostředně za daným prvkem, které nesplňují invariant maxima). Tam, kde tato sekvence končí, začíná sekvence následníků. Pak v části algoritmu hledání množiny výsledků (řádky 20-26) můžeme vyhodit testování, jestli se daný uzel nachází v množině *Max_not*.

Nahrazená část algoritmu 3.8 (Osa následník, předchůdce)

```
10       // Nalezení množiny těch prvků, které do výsledku nepatří
11       SetProjection(DF);
12       SetCol(Max);
13       MAX = Read(pozice);
14       (start, stop) = Init(pozice); i = start;
15       m = Read(i);
```

[3] Implementace operace `not in Max_not` se v praxi díky setříděnosti množiny *Max_not* implementuje jako porovnání s prvním prvkem množiny a postupným odmazáváním prvního prvku, který už byl nalezen. Tato operace má složitost *O(1)*.

```
16        while i < stop and MaxInvariant(m):
17             (m,i)Next();
18        start = i;

20        // Nalezení množiny výsledků
21        SetCol(ID);
22        i = start; id = Read(i);
23        while i < stop:
24             M.add(id);
25             (id,i) = Next();
```

Pro hledání předchůdců můžeme použít místo průchodu *(m+p)* záznamů v atributu *Max Algoritmus 3.6 (Skoč a jdi)* pro nalezení všech předků. Průchod *(m+p)* záznamů v atributu *ID* (řádky 20-26) však zůstane, pouze nahradíme test, jestli aktuální pozice se nachází v množině *Max_not* (řádek 24) testem, jestli aktuální *id* se nachází v množině *Max_not*.

4 Algoritmy pro XML data se schématem

V této kapitole se budeme zabývat algoritmy mapování XML dokumentu na relační databázi za použití schématu XML dokumentu. Naším hlavním cílem je optimalizovat hledání cest v XML dokumentu, proto diskusi omezíme na možnosti schématu zapsaného formou DTD dokumentu. Navíc v DTD dokumentech nebudeme uvažovat entity, komentáře, řídící instrukce.

4.1 Datová struktura

Stejně jako pro algoritmy bez použití schématu XML dokumentu, je i tady důležité rozhodnout o implementaci vazby rodič - dítě. Budeme ji opět implementovat redundantně – FK přístupem a také DF přístupem. Každá tabulka tedy bude obsahovat atributy *min* a *max* přiřazené DF metodou a také cizí klíč odkazující na rodiče daného uzlu. Dále budou logický a následně fyzický model databáze závisetna DTD dokumentu.

4.1.1 Logický model

Práce [1] popisuje tzv. *DTD graf*. Jeho uzly jsou elementy, atributy a operátory definované v DTD dokumentu. Elementy se v DTD grafu vyskytují pouze jednou a atributy a operátory tolik krát, kolik krát se vyskytují v DTD dokumentu. Orientovanými hranami DTD grafu jsou pak vazby element - podelement, element - atribut, element - operátor a operátor – podelement. Na obrázku 12 je znázorněn příklad DTD grafu.

Stejný DTD graf použijeme pro postavení logického modelu relační databáze. Budeme nazývat každý uzel grafu, z kterého nevede žádná hrana, *listem* (i když nejde o strom a ani o acyklický graf). Všimněme si, že jde právě o atributy a elementy s textovým obsahem bez atributů.

Mějme uzly DTD grafu *E*,*F*. Budeme říkat, že *E* je podelementem *F* (a *F* je nedelementem *E*), právě když v DTD grafu existuje hrana z *F* do *E*. *E* je podelementem *F* přes operátor (a *F* je nadelementem *E* přes operátor), právě když existuje hrana z *F* do libovolného operátoru a hrana z tohoto operátoru do *E*.

Při sestavování schématu relační databáze budeme dodržovat následující pravidla:

- pro každý nelistový uzel DTD grafu kromě uzlů reprezentujících operátor vytvoříme

tabulku;
- každý listový uzel budeme do relačního schématu mapovat jako atribut tabulky jeho nadelementu nebo nadelementu přes operátor, pokud tímto operátorem není * nebo +;
- pro každý listový uzel, který je podelementem nějakého uzlu přes operátor * nebo +, vytvoříme tabulku.

Pro každý element *E* s množinou listových podelementů (event. přes operátor kromě * a +) {LPE_1,...,LPE_n} a s množinou nadelementů (event. přes operátor) {NE_1,...,NE_k} vytvoříme tabulku:

E(ID,LPE_1,...,LPE_n, Rodič, Tabulka_Rodiče, Min, Max),

kde Tabulka_Rodiče je typu výběr z hodnot {NE_1,...,NE_k} a konkrétní hodnota reprezentuje tabulku, ve které je uložen rodič daného záznamu. Rodič je cizím klíčem do této tabulky. Pokud je množina nadelementů prázdná, atributy Rodič a Tabulka_Rodiče nevytvoříme. Pokud je element *E* listový, množina listových podelementů je prázdná.

Kromě sestavení relačního schématu budeme také ukládat ke každé tabulce informace o tom, kde se můžou nacházet děti záznamů této tabulky. Tuto informaci získáme následovně:

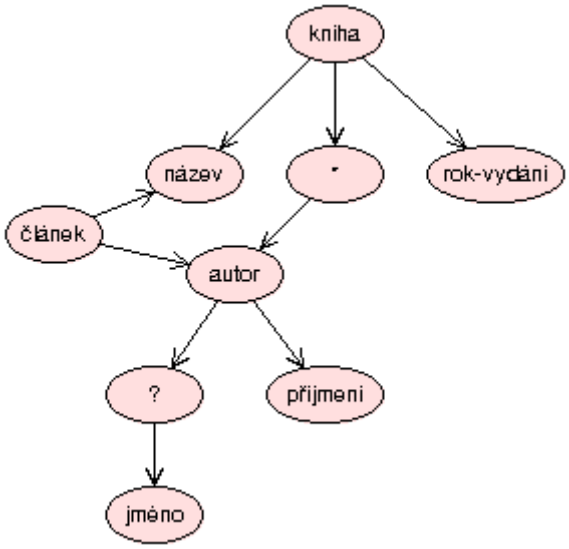
- Ke každé tabulce *E* uložíme množinu uzlů D_E DTD grafu, do kterých vede hrana (event. přes operátor) z uzlu *E*:
  $D_E = \{PE_1, \ldots, PE_m\}$.
- Každý element $PE_i \in D_E$, který je listový a není podelementem *E* přes operátor * nebo +, přesuneme do množiny A_E.

V množině D_E se tedy nachází seznam všech tabulek, ve kterých se můžou vyskytovat děti záznamů tabulky *E* a v množině A_E se nachází seznam všech atributů, ve kterých se vyskytují děti záznamů tabulky *E*, které byly namapovány přímo na atribut této tabulky.

Příklad: Logický model grafu z obrázku 12 by obsahoval tyto tabulky:

```
Kniha  (ID, Název, Rok-vydání, Min, Max)
Článek (ID, Název, Min, Max)
Autor  (ID, Jméno, Příjmení, Rodič, Tabulka_Rodiče, Min, Max)
```

```
<!ELEMENT kniha(název, autor*)>
<!ELEMENT název(#PCDATA)>
<!ELEMENT autor(jméno?, příjmení)>
<!ELEMENT jméno(#PCDATA)>
<!ELEMENT příjmení(#PCDATA)>
<!ATTLIST kniha rok-vydání CDATA>
<!ELEMENT článek(název,autor)>
```



Obr. 12 Příklad DTD grafu

Model z příkladu na obrázku 12 dále obsahuje informaci o možných dětech záznamů jednotlivých tabulek:

```
Kniha:D={Autor}, A={Název, Rok_vydání}
Článek:D={Autor}, A={Název}
Autor:D={}, A={Jméno, Příjmení}
```

4.1.2 Fyzický model

V úložišti typu C-store budeme používat pro každou tabulku *E*, která obsahuje atributy `Rodič` a `Tabulka_Rodiče`, následující projekce:

```
E_FK1 (ID, LPE1,...,LPEn, Rodič, Tabulka_Rodiče, Min| Rodič,
Tabulka_Rodiče, Min)
E_FK2 (ID, Rodič, Tabulka_Rodiče| ID),
```

a pro každou tabulku E projekce:

```
E_DF1 (ID,LPE1,...,LPEn, Min, Max| Min)
E_DF2 (ID,LPE1,...,LPEn, Min, Max| ID)
```

Příklad:

Fyzický model vzniklý z příkladu na obrázku 12 bude obsahovat tyto projekce:

```
Autor_FK1 (ID, Jméno, Příjmení, Rodič, Tabulka_Rodiče, Min|
  Rodič, Tabulka_Rodiče, Min)
Autor_ FK2 (ID, Rodič, Tabulka_Rodiče| ID)

Kniha_DF1 (ID, Název, Rok-vydání, Min, Max| Min)
Článek_DF1 (ID, Název, Min, Max| Min)
```

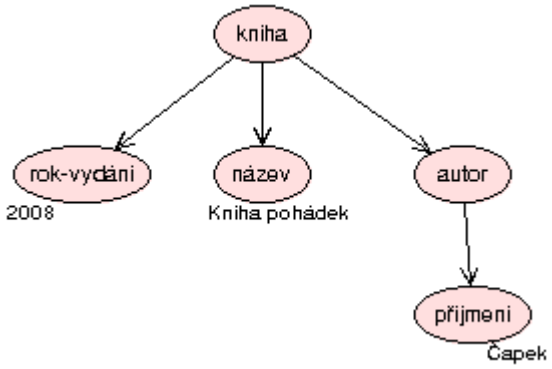
```
Autor_DF1 (ID, Jméno, Příjmení, Min, Max| Min)

Kniha_DF2 (ID, Název, Rok-vydání, Min, Max| ID)
Článek_DF2 (ID, Název, Min, Max| ID)
Autor_DF2 (ID, Jméno, Příjmení, Min, Max| ID)
```

4.2 Získávání cest v XML stromu

Zatímco u algoritmu, který na mapování XML dokumentu nepoužíval jeho schéma, byly všechny uzly XML stromu namapovány do jedné tabulky, v tomto případě jsou uzly v různých tabulkách. Proto id záznamu tabulky není dostatečným identifikátorem kontextu uzlu. Kromě id záznamu potřebujeme název tabulky. Navíc, některé uzly jsou namapovány pouze jako atribut jejich nadelementu. Pro tyto uzly určení kontextu znamená určení tabulky a id záznamu, kde se uzel nachází a navíc určení atributu, o který uzel přesně jde. V celé kapitole budeme tedy kontextem uzlu rozumět trojici (*T*, *id*, *A*), kde *T* je název tabulky, *id* je id záznamu v tabulce *T* a *A* je atribut. Pokud je uzlem přímo celý záznam, atribut bude „ID“.

Příklad na obrázku 13 ukazuje XML soubor validní vzhledem k DTD schématu z obrázku 12, XML strom postavený z tohoto dokumentu.

```
<kniha rok-vydání = 2008>
  <název> Kniha pohádek </název>
  <autor>
    <příjmení> Čapek </příjmení>
  </autor>
</kniha>
```



Obr. 13 Příklad XML stromu validního vzhůedem k DTD z obrázku Obr. 12

Tabulky popsané v kapitole *4.1.1 Logický model* budou v příkladu z obrázku 13 naplněny následovně:

Kniha

ID	Název	Rok-vydání	Min	Max
1	Kniha pohádek	2008	1	10

Autor

ID	Jméno	Příjmení	Rodič	Tabulka_Rodiče	Min	Max
1	Null	Čapek	1	Kniha	6	9

Tabulka *Článek* je prázdná.

Kontextem elementu *Kniha* z příkladu je trojice (*T*, *id*, *A*): `(Kniha,1,ID)`. Kontextem elementu *Příjmení* je trojice: `(Autor,1,Příjmení)`.

4.2.1 Osa dítě

Při hledání všech uzlů na ose dítě využijeme přístup FK. Algoritmus dostane na vstupu trojici určující kontextový uzel: (*T* , *id*, *A*). Pokud *A* není „ID“, víme, že kontextový uzel je určitě listový (určeno již DTD dokumentem) a tedy nemá děti. Pokud uzel má (může mít) děti, budeme je hledat ve všech tabulkách obsažených v množině D_T a také přímo v jednotlivých atributech daného záznamu, tedy v množině atributů A_T.

Pro algoritmus uzpůsobíme funkci *Search()* následovně:

Search(X, start, stop) – nalezení prvního výskytu prvku *X*, v intervalu pozic *start*, *stop* v aktuální projekci, v aktuálním atributu.

Algoritmus 4.1 (Osa dítě)

Algoritmus dostane kontextové určení daného uzlu a vrátí množinu *D* všech dětí tohoto uzlu. Tato množina je na začátku prázdná.

Vstup: (T, id, A)
Výstup: D = {}

```
1       // Pro listové elementy
2       if A <> "ID":
3            return D;
4       // Jinak
5       // Pro všechny projekce, kde se může dítě vyskytovat
6       for p in T.D:
7            //Nalezení množiny odpovídajících záznamů
8            SetProjection(MakeName(p,"FK1"));
```

```
9           SetCol(Rodič);
10          r = id; start = Search(id);
11          while r == id:
12              (r,stop) = Next();
13          stop = stop - 1;
14          SetCol(Tabulka_Rodiče);
15          t = T; start = Search(T, start, stop); st = start;
16          while t == T and st <= stop:
17              (t,st) = Next();
18          stop = st - 1;
19          // Projekce na odpovídající atribut, do množiny uložíme
        celý    kontext
20          SetCol(ID);
21          i = start;
22          D.add((p,Read(i),"ID"));
23          while i < stop:
24              (d,i) = Next();
25              D.add((p,d,"ID"));
26      // Nakonec přidání všech dětí – listů
27      SetProjection(MakeName(T,"DF2"));
28      SetCol(ID);
29      Search(id);
30      for a in T.A:
31          if ReadCol(a) is not null:
32              D.add((T,id,a));
```

Složitost algoritmu

Oproti algoritmu bez schématu závisí složitost tohoto algoritmu nejenom na počtu všech uzlů XML stromu, ale také na jejích rozdělení do jednotlivých tabulek. Nechť d je počet prvků množiny D_T, tedy počet tabulek, ve kterých se může nacházet dítě daného uzlu. Nechť a je počet prvků množiny A_T, tedy počet všech listových dětí namapovaných přímo na atribut dané tabulky.

Nechť $n_1,...,n_d$ jsou počty prvků v jednotlivých tabulkách, které algoritmus prohledává. Nechť n_0 je počet prvků v tabulce T. Nechť $k_1,...k_d$ jsou počty těch uzlů v jednotlivých tabulkách, jejichž rodič má *id* shodné s daným uzlem a $m_1,...m_d$ jsou počty uzlů nalezených na ose dítě v jednotlivých tabulkách (jejich rodič má *id* shodné s daným uzlem a nachází se ve stejné tabulce jako daný uzel). Označme $m = \sum m_i$, $k = \sum k_i$ a $n = \sum n_i$.

Tento algoritmus v každé tabulce z množiny D_T nejprve vyhledá (*Search()*) začátek sekvence uzlů, jejichž rodič má *id* stejné jako daný uzel (zatím však odkazuje do libovolné tabulky). Toto vyhledání má pro každou tabulku složitost $O(\log n_i)$. Dohromady tedy tato složitost pro všechny tabulky činí:

$$O(\sum \log(n_i)) = O(\log \prod n_i)$$

skoků na disku. Pak následuje nalezení konce této sekvence, což znamená *O(k)* provedení operací *next()*. Operace *Search(start,stop)* nám v každé tabulce zaručí vyhledání začátku sekvence záznamů, jejichž rodič je přímo daný prvek (tedy předtím nalezené *id* odkazuje do dané tabulky), v čase $O(\log k_i)$. Celkově je tedy tato složitost pro všechny tabulky:

$$O(\sum \log(k_i)) = O(\log(\prod k_i)) \text{ skoků.}$$

Nakonec nalezení konce této sekvence znamená projití *m* prvků pomocí operace *next()*.

Pro získání listových dětí daného uzlu namapovaných přímo na atribut rodiče je potřeba jednoho logaritmického vyhledání v tabulce, kde je uložen daný prvek a pak *O(a)* provedení operace *ReadCol()*.

V následující tabulce je souhrnně uveden počet výskytů abstraktních diskových operací *next()* a *jump()*:

Operace	**Počet výskytů**
next()	$O(m+k)$
jump()	$O(a + \log(n_0 \times \prod (n_i \times k_i)$

Funkce ReadCol()je taky skokem na disku. Případné její odlišení by umožnilo prostor pro její optimalizaci.

Udělejme v tuto chvíli odbočku zpět k fyzickému modelu popsanému v kapitole *4.1.2 Fyzický model*. Každá projekce *E_FK1* v modelu je řazena postupně podle těchto atributů:

- `Rodič`
- `Tabulka_Rodiče`
- `Min`

Řazení primárně podle atributu `Tabulka_Rodiče` a sekundárně podle atributu `Rodič` se může na první pohled zdát přirozenější. Avšak princip, který jsme využili pro nalezení uzlů na ose dítě a jehož využijeme i na dalších osách, ukazuje že zvolená priorita řazení bude pravděpodobně dosahovat lepších výsledků.

První vyhledání začátku sekvence v atributu, podle něhož je tabulka setříděna primárně, totiž v obou případech stojí *O(log n)* skoků na disků, kde *n* je počet prvků v této tabulce. Řazení podle prvního atributu rozdělí záznamy tabulky do *s* skupin, ve kterých je hodnota prvního atributu stejná. Druhé vyhledávání (*Search(start,stop)*) pak probíhá pouze nad počtem prvků v dané skupině, tedy řádově *n/s*. Čím vyšší je číslo *s*, tím je druhé hledání efektivnější.

Řazení primárně podle atributu `Tabulka_Rodiče` by znamenalo rozdělení všech záznamů tabulky maximálně do tolika skupin, kolik elementů se nachází v DTD stromu (tedy $s \leq$ počet vytvořených tabulek). Řazení primárně podle atributu `Rodič` však rozdělí záznamy do řádově tolik skupin, kolik uzlů se nachází v XML stromu (tedy $s \cong n$).

Algoritmus se schématem vs. bez schématu

Pro vyhledání sekvence dětí daného uzlu se v algoritmu bez schématu (*Algoritmus 3.1 (Osa dítě)*) vyhledává mezi všemi uzly XML stromu (jelikož jsou všechny namapovány do stejné tabulky). Naopak, algoritmus se schématem vyhledává vícekrát. Z vlastností logaritmické funkce vidíme, že v tomto ohledu je algoritmus bez schématu rychlejší (např. zvýší-li se počet prvků, ve kterých vyhledáváme, dvojnásobně, znamená to pouze o jeden skok navíc; zatímco vyhledávání v další stejně dlouhé posloupnosti znamená zdvojnásobení počtu kroků). Navíc algoritmus bez schématu provede přesně *O(m)* operací *Next()*, nemusí tedy procházet žádné prvky, které se nakonec do výsledku nedostanou. A konečně, provádění operace *ReadCol()* není v algoritmu bez schématu vůbec nutné. Pro vyhledání všech dětí daného uzlu dosahuje algoritmus bez schématu lepších výsledků.

Algoritmus se schématem by musel využít rozhraní distribuovaných výpočtů, aby fungoval optimálněji. Nicméně i algoritmus bez schématu skýtá možnost využití distribuovaného řešení. C-store totiž počítá s rozdělením dlouhé tabulky do několika segmentů. Podívejme se ještě na situaci, která přináší na světlo jednu výhodu algoritmu se schématem. Výsledkem běžných XPath výrazů nejsou všechny děti daného uzlu, nýbrž všechny děti daného uzlu určitého typu (např. selekce pouze na autory knihy). V tom případě můžeme výsledek vyhledávat v jediné tabulce, pokud je hledaný element namapován na samostatnou tabulku, a ještě lepší složitosti *O(1)* dostaneme, pokud je hledaný element namapován na atribut svého rodiče.

4.2.2 Osa potomek

Podobně, jako u algoritmu bez schématu (*Algoritmus 3.2 (Osa potomek)*), použijeme DF

přístup. Matematické vlastnosti této metody, jako např. invariant minima a maxima, nebo řazení uzlů, jsou stejné jako v kapitole 3.2.2.

Ke změně ovšem dochází tím, že potomci daného uzlu se nachází ve více tabulkách. Jelikož máme pro každou tabulku uloženou množinu D_T tabulek, kde se můžou nacházet děti záznamů dané tabulky, nahrazením množiny D_T jejím tranzitivním uzávěrem získáme množinu tabulek, kde má smysl hledat potomky daného uzlu.

Oproti algoritmu bez schématu tento algoritmus musí vyhledávat logaritmickou složitostí v každé tabulce, kde se můžou nacházet potomci daného uzlu (daný uzel se ve většině z nich ale nenachází). Toto vyhledávání je postaveno na invariantu minima a nalezne prvního potomka daného uzlu v této tabulce. Pro toto vyhledávání uzpůsobíme operaci *Search():*

Search(X,operátor) – nalezení prvního výskytu prvku *P*, pro který platí `P operátor X`, v aktuální projekci, v aktuálním atributu.

Podle atributu *Max* rozlišíme opravdové potomky daného uzlu od uzlů, které splňují pouze podmínku minima (uzly na ose následník). Nakonec pro všechny nalezené potomky v jednotlivých tabulkách přiřadíme do množiny potomků i jejich listové děti, které se nacházejí v jednotlivých atributech každé prohledávané tabulky.

Algoritmus 4.2 (Osa potomek)

Algoritmus dostane kontextové určení daného uzlu a vrátí množinu *P* všech jeho potomků. Ta je na začátku prázdná.

Vstup: (T, id, A)
Výstup: P = {}

```
1       // Listový uzel nemá potomky
2       if A <> "ID":
3           return P;
4       // Zjištění hodnot MIN a MAX daného uzlu
5       SetProjection(MakeName(T,"DF2"));
6       SetCol(ID);
7       Search(id);
8       MIN = ReadCol(Min);
9       MAX = ReadCol(Max);
10      // Přidání samotných listových dětí daného uzlu
11      for a in T.A:
12          if ReadCol(a) is not null:
```

```
13              P.add((T,id,a));
14        // Odvození množiny všech tabulek potomků
15        T.P = TransitiveClosure(T.D);
16        // Pro každou tabulku v množině T.P
17        for p in T.P:
18             // Nalezení všech potomků jako sekvence start...stop
19             SetProjection(MakeName(p,"DF1"));
20             SetCol(Min);
21             start = Search(MIN,">"); stop = start;
22             // Pokud jsme žádného potomka v tabulce nenalezli, další
               tabulka
23             if start == -1:
24                continue;
25             m = ReadCol(Max);
26             while m < MAX:
27                (m,stop) = Next();
28             stop = stop -1;
29             // Zahrnutí listových dětí do výsledku
30             // L je „matice" listových dětí uložených v této tabulce
31             for a in p.A:
32                SetCol(a);
33                i = start;
34                dítě = Read(i);
35                while i <= stop:
36                     if dítě is not null:
37                          L[i].add(a);
38                     (dítě,i) = Next();
39             // Projekce na odpovídající atribut
40             SetCol(ID);
41             i = start;
42             id = Read(i);
43             while i <= stop:
44                // Přidání samotného záznamu
45                P.add((p,id,"ID"));
46                // Přidání listových dětí záznamu
47                for a in L[i]:
48                     P.add((p,id,a));
49                (id,i) = Next();
```

Složitost algoritmu

Nechť p je počet prvků tranzitivního uzávěru množiny D_T, tedy počet tabulek, ve kterých se můžou nacházet potomci daného uzlu. Nechť a je průměrný počet prvků množiny A_T, tedy počet všech listových dětí jednotlivých tabulek.

Nechť $n_1,...,n_p$ jsou počty prvků v jednotlivých tabulkách, které algoritmus prohledává. Nechť n_0 je počet prvků v tabulce T. Nechť $m_1,...m_p$ jsou počty uzlů splňující invariant minima i maxima v jednotlivých tabulkách. Označme $m = \sum m_i$ a $n = \sum n_i$.

Tento algoritmus vyžaduje *O(p)* krát provedení operace *Search(),* která má složitost $O(\log n_i)$ skoků na disku. Následně se v každé tabulce přečte sekvence m_i po sobě jdoucích záznamů. Kromě toho se pro každou tabulku a pro každé její listové dítě provede skok na disku a pak čtení sekvence dlouhé m_i prvků, čtení více atributů jednoho záznamu se provádí pouze v jedné tabulce – v tabulce daného uzlu.

V následující tabulce je souhrnně uveden počet výskytů abstraktních diskových operací *next()* a *jump()*:

Operace	**Počet výskytů**
next()	$O(p \times a \times m)$
jump()	$O(p \times a + \log(n_0 \times \prod n_i)$

Výhody a nevýhody tohoto algoritmu oproti algoritmu bez schématu jsou zcela stejné jako na ose dítě (kapitola *4.2.1 Osa dítě*).

4.2.3 Osa rodič

Vzhledem k tomu, že u každého uzlu si uchováváme informaci o jeho rodiči, použijeme metodu FK. Jsou dvě možnosti, jak jsou dva uzly XML stromu – daný uzel a jeho hledaný rodič – namapovány do tabulek:

1) daný uzel je atributem záznamu, ve kterém je namapován jeho rodič (*A* je různé od „ID“), nebo
2) daný uzel je namapován v samostatné tabulce (*A* = „ID“) a v atributech `Tabulka_Rodiče` a `Rodič` jsou uloženy informace o jeho rodiči.

Algoritmus 4.3 (Osa rodič)

Algoritmus dostane kontextové určení daného uzlu a vrátí kontextové určení jeho rodiče.

Vstup: (T, id, A)
Výstup: R

```
1       // Mapování prvku na atribut jeho rodiče
2       if A <> "ID":
3           R = (T,id,"ID");
4       if A == "ID":
5           // Nalezení daného prvku
6           SetProjection(MakeName(T,"FK2"));
7           SetCol(ID);
8           Search(id);
9           // Přečtení informací o rodiči
10          t = ReadCol(Tabulka_Rodiče);
11          r = ReadCol(Rodič);
12          R = (t,r,"ID");
```

Složitost algoritmu

Časová složitost operace *Search()* je *O (log n)*, kde *n* je počet záznamů v dané tabulce *T*. Zjevně tento algoritmus je lepší než algoritmus bez schématu (*Algoritmus 3.3 (Osa rodič)*).

4.2.4 Osa předek

Pro hledání všech předků daného uzlu jsme pro mapování bez schématu uvedli dva algoritmy založené na různých přístupech. Zatímco přístup FK (*Algoritmus 3.4 (Osa předek – FK metoda)*) rekurzivně vyhledá rodiče daného prvku, rodiče rodiče, atd., přístup DF (*Algoritmus 3.5 (Osa předek – DF metoda)*) prochází sekvenci některých uzlů (těch, které splňují invariant minima) a testuje je na invariant maxima. Vzhledem k tomu, že se obecně nedá usoudit na to, který algoritmus je efektivnější, navrhli jsme algoritmus *(Skoč a jdi)*, který je kombinací obou dvou přístupů.

Pro mapování dat se schématem je situace poněkud odlišná. Metoda DF v tomto případě jistě není efektivnější než metoda FK. Předpokládejme, že máme množinu všech tabulek, ve kterých se potenciálně (podle DTD schématu) můžou nacházet předkové daného uzlu. Nechť $t_1,...,t_k$ jsou tyto tabulky a m_i je počet předků nalezených v tabulce t_i. Pokud v DTD grafu nebyl cyklus – a tedy XML elementy se nemůžou vnořovat rekurzivně, je pro každé $i \in \{1,\ldots,k\}$: $m_i \leq 1$. I pokud v DTD grafu cyklus byl, pouze ty elementy, které se cyklu

účastní, se mohou na ose předků libovolného uzlu vyskytovat vícekrát. Proto pouze v tabulkách reprezentujících uzly DTD grafu nacházejících se v cyklu, může být m_i větší jako 1. Metoda DF je standardně založena na matematických vlastnostech hodnot *min* a *max* jednotlivých uzlů. V každé tabulce $t_1,...,t_k$ bychom nejdříve vyhledali první prvek splňující invariant minima (se složitostí *O (log n)*) a poté bychom otestovali na invariant maxima $O(n_i)$ hodnot, kde n_i je počet prvků tabulky t_i. Ve většině tabulek bychom však našli jeden nebo žádný prvek relevantní pro výsledek dotazu. Ve zbytku kapitoly představíme algoritmus založený na metodě FK a ukážeme, že bude vyhledávat v jednotlivých tabulkách se stejnou složitostí, ale prohledá pouze opravdu relevantní tabulky (nikoliv potenciálně možné podle DTD schématu) a dál už nebude číst další zbytečná data.

Algoritmus 4.4 (Osa předek - FK metoda)

Algoritmus dostane kontextové určení daného uzlu a vrátí množinu *P* všech jeho předků.

Vstup: (T, id, A)
Výstup: P ={}

```
1       // Pro listový uzel nacházející se v atributu svého rodiče
2       if A <> "ID":
3            P.add((T,id,"ID"));
4       // V cyklu hledání rodiče rodiče atd.
5       t = T; r = id;
6       while t is not null:
7            // Posun o generaci
8            SetProjection(MakeName(t,"FK2"));
9            SetCol(ID);
10           Search(r);
11           t = ReadCol(Tabulka_Rodiče);
12           if t is not null:
13               r = ReadCol(Rodič);
14               P.add((t,r,"ID"));
```

Složitost algoritmu

Nechť *m* je počet uzlů nalezených na ose předek. Nechť $n_1,...,n_m$ jsou počty záznamů v tabulkách, kde se jednotliví předci nacházejí. Pak operace *Search()* proběhne *m*-krát a její vnitřní složitost je $O(\log n_i)$. Celková složitost algoritmu je tedy:

$$O(m \times \log n) \text{ operací } jump().$$

Algoritmus se schématem vs. bez schématu

Chceme-li porovnat modely dat se schématem a bez schématu vzhledem k časové složitosti vyhledávání cesty XML dokumentu na ose *předek*, musíme si uvědomit, že v obou modelech jsou použity různé principy. Zatímco pro data se schématem jsme použili algoritmus založený výlučně na metodě FK, u dat se schématem, jsme nalezli algoritmus, který se v každém kroku dynamicky rozhoduje, zda použije přístup FK nebo DF. Kombinovaný *Algoritmus 3.6 (Skoč a jdi)* tedy může dopadnout dvěma způsoby. Buď použije čistě metodu FK a v tom případě má přibližně stejnou složitost jako algoritmus pro data se schématem, až na čtení o jeden sloupec (`Tabulka_Rodiče`) v každé iteraci méně, nebo použije částečně i přístup DF a to proto, že to v tomto kroku bude efektivnější. Algoritmus pro model bez schématu je tedy lepší.

4.2.5 Osa sourozenec

Algoritmus pro vyhledání všech sourozenců daného uzlu nejprve zjistí identifikaci rodiče daného uzlu a poté vyhledá všechny jeho děti. Do výsledku se pak dostanou pouze ty děti, které nejsou daný uzel.

Algoritmus je triviální, proto ho nebudeme uvádět. Jeho časová náročnost a rovněž výhody a nevýhody ve srovnání s algoritmem bez schématu jsou stejné jako pro algoritmus nalezení všech dětí daného uzlu. Jediný rozdíl je, že tento algoritmus musí navíc logaritmicky prohledat tabulku daného uzlu, aby mohl přečíst rodiče.

4.2.6 Osa mladší (starší) sourozenec

Uvědomme si, že pořadí sourozenců v popsaném modelu není zcela implementováno. Na jedné straně, každému uzlu jsou sice přiřazeny hodnoty *min* a *max* (které pořadí sourozenců určují), na straně druhé, tyto hodnoty u listových elementů fyzicky neukládáme. Model, jenž používáme, umožňuje rozlišit pořadí sourozenců pouze pro nelistové uzly.

Rozšíření modelu

Ukážeme nyní způsob, jak lze tento model rozšířit a umožnit tím vyhledávání na ose *mladší* (resp. *starší*) *sourozenec*. Toto rozšíření bude zároveň sloužit i pro vyhledávání na ose *následník* (resp. *předchůdce*).

Pro každý element *E* DTD stromu s množinou listových podelementů (event. přes operátor

kromě * a +) $\{LPE_1,...,LPE_n\}$ a s množinou nadelementů (event. přes operátor) $\{NE_1,...,NE_k\}$ vytvoříme místo tabulky:

E(ID,LPE_1,...,LPE_n, Rodič, Tabulka_Rodiče, Min, Max)

tabulku:

E(ID,LPE_1, LPE_1_Min, LPE_1_Max...,LPE_n, LPE_n_Min, LPE_n_Max, Rodič, Tabulka_Rodiče, Min, Max),

kde hodnoty LPE_i_Min a LPE_i_Max jsou hodnoty *min* a *max* přiřazené metodou DF uzlu namapovanému jako atribut LPE_i.

Potom algoritmus na vyhledání všech mladších sourozenců daného uzlu pracuje takto:

Algoritmus 4.5 (Mladší sourozenec)

Vstup: (T, id, A)
Výstup: MS = {}

```
1       // Přečtení hodnoty Min daného prvku
2       SetProjection(MakeName(T,"DF2"));
3       SetCol(ID);
4       Search(id);
5       // Nechť funkce MakeName vrátí pro A =="ID" pouze Min
6       MIN = ReadCol(MakeName(A,"Min"));
7       // Nalezení rodiče
8       if A <> "ID":
9            tabulka_rodiče = T; rodič = id;
10      if A == "ID":
11           SetProjection(MakeName(T,"FK2"));
12           SetCol(ID);
13           Search(id);
14           tabulka_rodiče = ReadCol(Tabulka_Rodiče);
15           rodič = ReadCol(Rodič);
16      // Pro všechny projekce, kde se může dítě vyskytovat
17      for p in tabulka_rodiče.D:
18           //Nalezení množiny odpovídajících záznamů
19           SetProjection(MakeName(p,"FK1"));
20           SetCol(Rodič);
21           r = rodič; start = Search(rodič);
22           while r == rodič:
23                (r,stop) = Next();
```

```
24            stop = stop - 1;
25            SetCol(Tabulka_Rodiče);
26            t = tabulka_rodiče; start = Search(t, start, stop);
27            st = start;
28            while t == tabulka_rodiče and st <= stop:
29                (t,st) = Next();
30            stop = st – 1;
31            // Selekce na mladší sourozence
32            SetCol(Min);
33            m = Read(start); st = start;
34            while m < MIN and st <= stop:
35                (m,st) = Next();
36            stop = st – 1;
37            // Projekce na odpovídající atribut, do množiny uložíme
          celý    // kontext
38            SetCol(ID);
39            i = start;
40            D.add((p,Read(i),"ID"));
41            while i < stop:
42                (d,i) = Next();
43                D.add((p,d,"ID"));
44        // Nakonec přidání všech sourozenců – listů
45        SetProjection(MakeName(tabulka_rodiče,"DF2"));
46        SetCol(ID);
47        Search(rodič);
48        for a in tabulka_rodiče.A:
49            if (ReadCol(a) is not null) and
50                ReadCol(MakeName(a,"Min")) < MIN:
51                      D.add((tabulka_rodiče,rodič,a));
```

Časová složitost algoritmu je stejná, jako složitost algoritmu pro vyhledání všech sourozenců daného uzlu. Liší se pouze o multiplikativní konstantu v souvislosti s operací *next()*. Je to proto, že algoritmus navíc ještě testuje všechny předtím nalezené sourozence na hodnotu atributu *Min*.

Nakonec učiňme poznámku k rozšíření modelu. Původní model byl schopen zajistit určení pořadí sourozenců namapovaných do samostatné tabulky (a to jak pořadí dvou uzlů nacházejících se ve stejné tabulce, tak i pořadí sourozenců ve dvou různých tabulkách). Výše uvedené rozšíření umožňuje určit pořadí dvou libovolných sourozenců (tedy i těch

namapovaných jako atribut svého rodiče). Nicméně, i po rozšíření modelu umíme odpovědět pouze na dotaz „třetí autor dané knihy“, nikoliv však na dotaz „třetí dítě (ze všech) daného uzlu“. Znamenalo by to totiž setřídění množiny všech dětí daného uzlu. Model bychom pro tento účel mohli dál rozšířit o pořadí všech sourozenců v rámci rodiče.

4.2.7 Osa následník (předchůdce)

Jak jsme již dříve popsali v kapitole *3.2.7 Osa následník (předchůdce)*, pro daný uzel *U* XML stromu dokážeme určit všechny jeho následníky a předchůdce na základě invariantu minima a maxima. Proto je pro vyhledávání na ose následník (předchůdce) důležité znát hodnoty *min* a *max* DF metody pro všechny uzly XML stromu, tedy i pro listové uzly namapované jako atribut svého rodiče. Použijeme tedy rozšířený model, který je definován v kapitole *4.2.6 Osa mladší (starší) sourozenec*.

Připomeňme nyní invariant minima a maxima pro následníka (předchůdce) daného uzlu *U*. Uzel *V* je předchůdcem uzlu *U*, pokud platí:

$$min\ (V) < min\ (U) \tag{22}$$

a současně:

$$max\ (V) < max\ (U). \tag{23}$$

Naopak, uzel *V* je následníkem uzlu *U*, pokud platí:

$$min\ (V) > min\ U \tag{24}$$

a současně:

$$max\ (V) > max\ (U). \tag{25}$$

V modelu, kde některé uzly XML stromu jsou namapované jako atributy jejich rodičů je vhodné, uvědomit si následující tvrzení:

1) Pokud uzel *V* je předchůdcem uzlu *U*, pak i všichni potomci uzlu *V* jsou předchůdci uzlu *U*.
2) Pokud uzel *V* je následníkem uzlu *U*, pak i všichni potomci uzlu *V* jsou následníky uzlu *U*.
3) Pokud uzel *V* je předkem uzlu *U*, pak jeho potomci můžou ale nemusí být jak předchůdci, tak následníky uzlu *U*.
4) Pokud uzel *V* je potomkem uzlu *U*, pak i všichni potomci uzlu *V* jsou potomky uzlu *U*, tedy nejsou ani předchůdci ani následníky uzlu *U*.

Tvrzení 1 až 4 využijeme právě pro otestování, jestli listové děti daného záznamu můžou

nebo nemůžou patřit do výsledné množiny. Následující tabulka uvádí seznam situací, kdy zkoumaný uzel (sloupec **Uzel**) a jeho listové děti (sloupec **Listové dítě**) je následníkem (N) nebo předchůdcem (P) daného uzlu. Vstupními informacemi v tabulce jsou invarianty minima a maxima pro uzel nebo pro jeho listové dítě. V tabulce se u invariantu minima (maxima) zkoumaného uzlu (nebo jeho dítěte) nachází hodnota P, právě když zkoumaný uzel nebo jeho dítě splňuje invariant minima (maxima) pro to, aby byl *předchůdcem* daného uzlu. Hodnota N je pro *následníka* daného uzlu.

Invariant minima	Invariant maxima	Uzel	Invariant minima listového dítěte	Listové dítě
P	P	P	P	P
P	P	P	N	P
P	N	--	P	P
P	N	--	N	N
N	P	--	P	--
N	P	--	N	--
N	N	N	P	N
N	N	N	N	N

Algoritmus pro hledání všech předchůdců (resp. následníků) sestává ze dvou algoritmů. Jeden z nich prochází atribut *Min* dané tabulky a hledá prvky, které splňují invariant minima a druhý prochází atribut *Max* a jeho záznamy testuje na invariant maxima. Použijeme funkci `InvariantMin()` a `InvariantMax()` ve smyslu výše uvedené tabulky. Tímto způsobem prohledáme všechny tabulky. Prohledávání atributu *Min* má probíhá logaritmickou složitostí, u atributu *Max* se čte sekvenčně. Uvedeme řešení, které navíc používá paralelizmu při prohledávání atributu *Min* a atributu *Max* (je možné zavést ve všech algoritmech v distribuovaném prostředí). Nemusíme v atributu *Min* nejdříve vyhledat „stop pozici“ a pak teprve začít prohledávat všechny hodnoty atributu *Max* v intervalu [1..stop]. Místo toho použijeme dva paralelně běžící algoritmy *MinProcess* a *MaxProcess*, které navzájem komunikují proměnnou *STOP*. Hodnota proměnné *STOP* je na začátku nastavena na hodnotu n_i, tedy počet záznamů ve zpracovávané tabulce.

Dvojice algoritmů vytvoří pro každou tabulku pole *PredNas*, které každé pozici přiřadí jednu z následujících hodnot:

`Předchůdce` – uzel i jeho děti jsou předchůdcem daného uzlu.
`Následník` – uzel i jeho děti jsou následníkem daného uzlu.
`Předek` – uzel není ani předchůdcem ani následníkem, ale jeho děti můžou být předchůdcem nebo následníkem.
`Potomek` – uzel ani jeho děti nejsou předchůdcem ani následníkem daného uzlu.

Na začátku jsou všechny hodnoty nastaveny na `Null`.

Algoritmus *MinProcess* dostane na vstupu projekci *p* a číslo *MIN*. Tento algoritmus zapisuje do proměnné *STOP* (operace `LockWriteUnlock`) pokaždé, kdy to může znamenat ulehčení práce algoritmu *MaxProcess*. Algoritmus edituje pole *PredNas* a to na základě svých výpočtů a hodnot, které mezi tím mohl do pole zapsat algoritmus *MaxProcess*.

Algoritmus 4.6 (Osa předchůdce, následník - MinProcess)

Algoritmus uvádíme pro hledání předchůdce, diference pro hledání následníka jsou uvedeny v komentářích.

Vstup: p, MIN, PredNas
Výstup: STOP, PredNas

```
1       // Nalezení pozice, kde přestává být splněn invariant minima
2       start = 0; stop = n+1; první = n+1;
3       SetProjection(p);
4       SetCol(Min);
5       repeat
6            střed = (start + stop) div 2;
7            x = Read(střed);
8            if x >= MIN:
9               první = min(první, střed);
10              stop = střed;
11              // Pro hledání následníka neměníme proměnnou STOP –
                algoritmus MaxProcess musí v tomto případě projít
                všechny záznamy
12              STOP.LockWriteUnlock(první);
13           else:
14              start = střed;
15      until stop == (start + 1);
16      // Na pozici stop je teď první prvek nesplňující invariant
        minima
17      for i = 1 to stop – 1:
18           PredNas[i].Lock();
19           if PredNas[i].Value() in (predchudce, null):
20              PredNas[i].Set(predchudce);
21           if PredNas[i].Value() == naslednik:
22              PredNas[i].Set(predek);
23           PredNas[i].UnLock();
```

```
24        if x == MIN:
25             PredNas[stop].Lock();
26             PredNas[stop].Set(potomek);
27             PredNas[stop].UnLock();
28             stop = stop + 1;
29        for i = stop to n:
30             PredNas[i].Lock();
31             if PredNas[i].Value() in (naslednik,null):
32                 PredNas[i].Set(naslednik);
33             if PredNas[i].Value() == predchudce:
34                 PredNas[i].Set(potomek);
35        PredNas[i].UnLock();
```

Algoritmus *MaxProcess* dostane na vstup projekci *p*, maximum daného uzlu *MAX* a samozřejmě má k dispozici (ke čtení) proměnnou *STOP*. Algoritmus edituje pole *PredNas* a to na základě svých výpočtů a současně hodnot, které mezi tím mohl do pole zapsat algoritmus *MinProcess*.

Algoritmus 4.7 (Osa předchůdce, následník - Maxprocess)

Vstup: p, MAX, STOP, PredNas
Výstup: PredNas

```
1         // Pro každý prvek v atributu Max rozhodne o invariantu maxima
2         SetProjection(p)
3         SetCol(Max);
4         i = 1; m = Read(i);
5         while i < STOP.Value():
6              PredNas[i].Lock();
7                  if InvariantMax(m) == P:
8                       if PredNas[i].Value() in (predchudce, null):
9                            PredNas[i].Set(predchudce);
10                      if PredNas[i].Value() == naslednik:
11                           PredNas[i].Set(potomek);
12                 if InvariantMax(m) == N:
13                      if PredNas[i].Value() in (naslednik,null):
14                           PredNas[i].Set(naslednik);
15                      if PredNas[i].Value() == predchudce:
16                           PredNas[i].Set(predek);
17             PredNas[i].UnLock();
18             (m,i) = Next();
```

Po skončení obou algoritmů (ať už paralelně běžících nebo postupně nejdřív *Minprocess* a poté *MaxProcess*) máme v poli *PredNas* pro každou pozici v dané tabulce uloženou hodnotu, která vypovídá o tom, jestli daný uzel a jeho listové děti namapované jako jeho atributy jsou nebo nejsou předchůdcem (event. následníkem). Následující algoritmus dostane jako vstup pole *PredNas* a projekci *p* a výstupem bude množina všech předchůdců *P* a následníků N daného uzlu v dané tabulce.

Algoritmus 4.8 (Osa následník, předchůdce)

Vstup: p, PredNas
Výstup: P={}, N={}

```
1       t = UnmakeName(p,"DF1");
2       SetProjection(p)
3       // rozřazení uzlů reprezentovaných celým záznamem
4       SetCol(ID);
5       i = 1; id = Read(i);
6       // Pro hledání pouze předchůdce stačí skončit na pozici STOP,
        jinak na pozici n+1
7       while i < STOP.Value():
8            if PredNas[i].Value() == predchudce:
9                // je předchůdcem
10               P.add((t,id,"ID"));
11           if PredNas[i].Value() == naslednik:
12               // je následníkem
13               N.add((t,id,"ID"));
14           // pro hodnotu predek nebo potomek nepatří do žádné
        množiny
15           (i,id) = Next();
16           // rozřazení listových dětí
17           for a in t.A:
18                SetCol(MakeName(a,Min));
19                i = 1; m = Read(i);
20                if m is null:
21                     continue;
22                if PredNas[i].Value() == predchudce:
23                     P.add(t,id,a);
24                if PredNas[i].Value() == naslednik:
25                     N.add(t,id,a);
```

```
26                  if PredNas[i].Value() == predek:
27                      // rodic není předchůdce ani následník, jeho děti
                        podle hodnoty jejich min
28                      if InvariantMin(m) == N:
29                            N.add(t,id,a);
30                      if InvariantMin(m) == P:
31                            P.add(t,id,a);
```

Složitost algoritmu

Algoritmus prohledává všechny tabulky a pro každou z nich má dvě části. V první fázi se vytváří maska *PredNas* všech uzlů v tabulce. Do této části patří oba algoritmy *MinProcess* a *MaxProcess* běžící buď paralelně nebo postupně, nejdřív jeden a pak druhý. Ve druhé fázi se maska aplikuje na každý záznam tabulky a ten se na jejím základě do výsledné množiny přidá nebo nepřidá. Stejná maska se pak aplikuje na všechny sloupce ukládající hodnotu *min* DF metody pro listové děti jednotlivých uzlů.

Vyjádříme složitost algoritmu pro prostředí bez paralelního zpracování. Nechť n_i je počet záznamů jedné tabulky, n je počet všech uzlů, a_i je počet atributů ukládajících listové děti v dané tabulce, a a je průměrný počet těchto atributů. Problémem u tohoto algoritmu je, že pole `PredNas` se obecně nevejde do paměti. Aplikace masky `PredNas` znamená mnoho skoků. U algoritmu *MinProcess* se tímto složitost zvětší pouze o n_i operací *next()*. U algoritmu *MaxProcess* a u druhé fáze celého algoritmu to ale znamená provést místo n_i operací *next(), 2*n_i* skoků na disku.

V následující tabulce uvedeme počet výskytů operací *next()* a *jump()* bez zohlednění práce s maskou `PredNas`.

Operace	**Počet výskytů**
next()	$O(a \times n)$
jump()	$O(a + \log(\prod n_i)$

Operace *Search()* má vnitřní časovou složitost *O(log n_i)* skoků na disku.

5 Praktické výsledky implementace

V kapitolách 3 a 4 jsme uvedli algoritmy pro vyhledávání uzlů XML stromu na jednotlivých osách v XPath. Tyto algoritmy jsou popsány na dostatečně nízké systémové úrovni pro určování časové složitosti vzhledem k počtu abstraktních diskových operací. Cílem této kapitoly je ukázat řešení technických detailů potřebných pro implementaci těchto algoritmů v prostředí C-store na aplikační úrovni a zhodnotit dříve provedeny odhady časové složitosti na základě reálně provedených měření.

Tato kapitola je organizovaná takto. Nejprve představíme dotazovací jazyk, ve kterém by měly být algoritmy napsané (kapitoly 5.1 a 5.2). Při výběru vhodné úrovně dotazovacího jazyka se zaměříme především na zachování složitostí všech výše popsaných algoritmů. Dále ukážeme, že některé principy používané v algoritmech v kapitolách 3 a 4 přispěly ke snížení časových nároků algoritmů, ale v prostředí C-store nelze těchto zefektivnění využít. V kapitole 5.3 pak uvedeme rozšíření C-store o další operátory[4], tak aby byla zachována dříve odhadnutá časová složitost. U každého principu dále uvedeme, o kolik by se časová složitost jednotlivých algoritmů zhoršila, kdyby byla použita pouze základní sada operátorů C-store. Konečně, v kapitole 5.4 shrneme výsledky experimentů, které prokázaly výrazné zefektivnění algoritmů v případě použití rozšířené sady operátorů oproti standardní implementaci systému C-store. Nakonec v kapitole 5.5 ukážeme kompletní technické řešení implementace dvou zvolených algoritmů.

5.1 Dotazovací jazyk

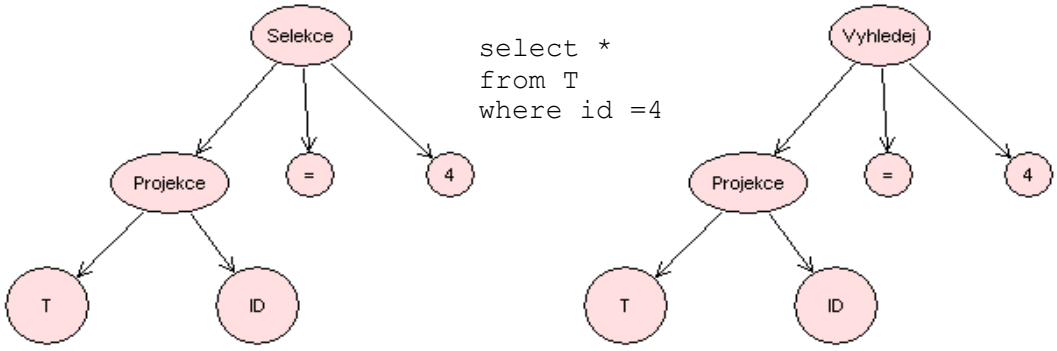
Dotazovacím jazykem pro databáze typu C-store je, stejně jako u relačních databází, SQL. Pokud je pro danou aplikaci výrazová síla SQL nedostatečná, lze navíc použít uložené procedury. Pro implementaci algoritmů uvedených v kapitolách 3 a 4 použijeme prvky uložených procedur, jako například kurzory nebo příkaz přiřazení.

Vnitřně funguje databáze C-store tak, že dotaz zadaný výrazovými prostředky SQL je nejdříve zpracován plánovačem dotazů. Ten s pomocí optimalizátoru naplánuje jednotlivé kroky vyhodnocení dotazu. Každý tento krok odpovídá provedení jednoho ze základních

[4] Navrhované rozšíření má obecný charakter. Je vhodné i pro jiné použití než vyhledávání cest v XML databázi. Jeho nevýhodou ale je, že do systému C-store zavádí určité prvky charakteristické pro běžné relační databáze. Vzhledem k tomu, že systém C-store je pořád vyvíjen a není zatím standardizován, lze zvážit, jestli by podobné rozšíření bylo přínosné (obecně nebo alespoň pro některé typy aplikací). Rozšířená sada operátorů není kompletním řešením, jde pouze o nastínění určité oblasti systému C-store, která by mohla být efektivnější.

definovaných operátorů systému C-store. Výstupy dříve provedených operátorů pak vstupují do později naplánovaných operátorů. Poslední naplánovaný operátor je právě jeden a jeho výstup odpovídá odpovědi na původní SQL dotaz.

Na obrázku 14 je znázorněn příklad, jak může plánovač dotazu naplánovat vyhodnocení jednoho dotazu dvěma různými způsoby. Operátory *Selekce* a *Projekce* z obrázku odpovídají stejnojmenným operátorům relační algebry. Operátor *Vyhledej* má stejný výsledek jako operátor *Selekce*. Liší se pouze v implementaci - namísto sekvenčního průchodu celým sloupcem v tomto sloupci binárně vyhledá daný prvek. Zřejmě lze operátor *Vyhledej* použít pouze na setříděných sloupcích tabulky. Který z možných plánů se použije, závisí na optimalizátoru. Operátory na obrázku 14 slouží pouze jako příklad. Skutečnou sadu operátorů popíšeme v následující kapitole.



Obr. 14 Příklad dvou různých exekučních plánů pro vyhodnocení dotazu

5.2 Sada operátorů definovaných v C-store

C-store definuje několik operátorů pro potřeby plánovače dotazů. Jednotlivé operátory přijímají vstupní operandy typu *Projekce*, *Sloupec*, *Bitový řetězec* a *Mapovací tabulku*. Výsledky jednotlivých operátorů jsou také těchto typů. Projekcí rozumíme množinu sloupců řazených ve stejném pořadí. Bitový řetězec indikuje, které záznamy z odpovídající množiny mají být přítomné v popisované podmnožině záznamů. Mapovací tabulka je pouze speciálním druhem sloupce.

Kromě těchto operandů pracují operátory navíc s predikáty, výrazy a názvy atributů. Nyní popíšeme všechny operátory, které C-store definuje.

`Decompress` – konvertuje komprimovaný sloupec na nekomprimovanou reprezentaci.

`Select` – odpovídá operaci selekce v relační algebře. Namísto omezení množiny záznamů však dává na výstup reprezentaci výsledné množiny pomocí bitového řetězce.

`Mask` – na vstupu dostane bitový řetězec *BS* a projekci *P*. Výstupem je omezení projekce *P* pouze na záznamy, jejichž odpovídající bit v *BS* je 1.

`Project` – je ekvivalentní operaci projekce v relační algebře (až na redundantní řádky ve výstupu).

`Sort` – setřídí všechny sloupce zadané projekce podle zadané podmnožiny těchto sloupců.

`Aggregate` – je ekvivalentní agregačním funkcím v SQL.

`Concat` – spojí více sloupců nebo projekcí řazených ve stejném pořadí do jediné projekce.

`Permute` – permutuje projekci nebo sloupec podle pořadí zadaného mapovací tabulkou.

`Join` – provede spojení dvou projekcí na základě zadaného predikátu.

`Logické operátory` – pracují nad bitovým řetězcem. `BAnd` a `BOr` vyprodukují bitové `And` a `Or` dvou bitových řetězců. `BNot` vrátí komplementární doplněk zadaného bitového řetězce.

Nyní jsme popsali základní sadu operátorů, které plánovač systému C-store používá na vyhodnocení SQL dotazu. V kapitole *5.1 Dotazovací jazyk* jsme uvedli, že každý dotaz zapsaný v syntaxi SQL se převede na posloupnost těchto operátorů. Můžeme tedy počítat s tím, že vyjadřovací síla jazyka zapsaného pomocí těchto operátorů není menší než vyjadřovací síla jazyka SQL (pokud má být SQL opravdu dotazovacím jazykem pro C-store databázi, musí se samotný návrh systému C-store vypořádat s případnými potížemi). Zapsání dotazu přímo pomocí operátorů (místo v notaci SQL) zavádí navíc možnost ovlivnit časovou složitost vyhodnocení dotazu. Použití jazyka operátorů namísto SQL při implementaci algoritmů z kapitoly 3 a 4 tedy usnadní diskusi o zachování odhadnuté časové složitosti algoritmů.

5.3 Časová složitost – odhadovaná vs. reálná

V této kapitole shrneme principy používané v algoritmech z kapitoly 3 a 4 které vedou k zefektivnění těchto algoritmů, ale nelze je popsat jazykem operátorů popsaných v předchozí kapitole. Tato kapitola je organizována do sekcí, které se zabývají diskusí jednotlivých principů. V každé sekci je popsán daný princip spolu s časovou úsporou, kterou přináší. Každá sekce začíná ukázkou implementace některého z algoritmů pomocí operátorů již definovaných. Následuje konfrontace zamýšleného principu s omezením daným sadou operátorů. Dále je uvedena diskuse, jak se tímto způsobem zhorší časová složitost algoritmů a nakonec je navržen operátor, který by implementaci tohoto principu umožnil.

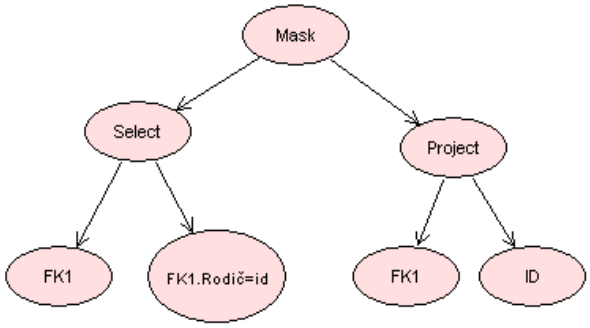
Jako příklady v jednotlivých sekcích jsou uváděny algoritmy nejčastěji používaných os XPath dotazů (osa dítě, potomek, předek). V této práci je upřednostněna ukázka implementace

algoritmů bez schématu (kapitola 3), a to proto že, XML data bez schématu jsou výrazně častěji používány.

5.3.1 Princip binárního vyhledávání

Všechny algoritmy v kapitolách 3 a 4 využívaly nalezení daného prvku v setříděné posloupnosti. Počítaly tedy se složitostí vyhledávání *O(log n)*, kde *n* je počet prvků. V systému C-store provedení operace selekce relační algebry sestává ze složení dvou operátorů: `Select` a `Mask` (viz kapitola *5.2 Sada operátorů definovaných v C-store*).

Na obrázku 15 je znázorněn plán vyhodnocování dotazu, který odpovídá algoritmu 3.1 (Osa dítě). Operátor `Project` funguje intuitivně. Na vstupu dostane celou projekci *FK1* na výstup pošle pouze její sloupec *ID*. Důležité je uvědomit si, jak se chová kombinace operátorů `Select` a `Mask`. Operátor `Select` vytvoří bitový řetězec stejné délky jako projekce *FK1*. V tomto bitovém řetězci jsou jedničky právě na pozicích, na kterých je v projekci *FK1* splněn zadaný predikát (tedy tam, kde je hodnota atributu *Rodič* rovná danému *id*). Operátor `Mask` přečte pouze ty záznamy výstupu operátoru `Project`, u kterých je na odpovídající pozici



Obr. 15 Vyhodnocení dotazu Algoritmus 3.1 (Osa dítě)

v bitovém řetězci jednička.

Operátor `Mask` tedy funguje v souladu s druhou částí (řádky 8-13) algoritmu 3.1 (Osa dítě). Čte totiž pouze relevantní záznamy. Nesoulad nastává u operátoru `Select`. První část (řádky 1-7) algoritmu počítá s řazením sloupce *Rodič*, a tudíž i s použitím binárního vyhledání první relevantní hodnoty a následného přečtení *m* po sobě jdoucích záznamů, kde *m* je počet záznamů splňujících predikát selekce. Operátor `Select` namísto toho vytváří bitový řetězec délky *n* (počet všech záznamů projekce *FK1*).

Definice systému C-store není striktní v tom, jakým způsobem má operátor `Select` výstup vytvářet. Obecně ho lze chápat čistě sekvenčně. Anebo může být tento operátor implementován tak, že se vnitřně rozhodne, zda použije sekvenční průchod celým sloupcem

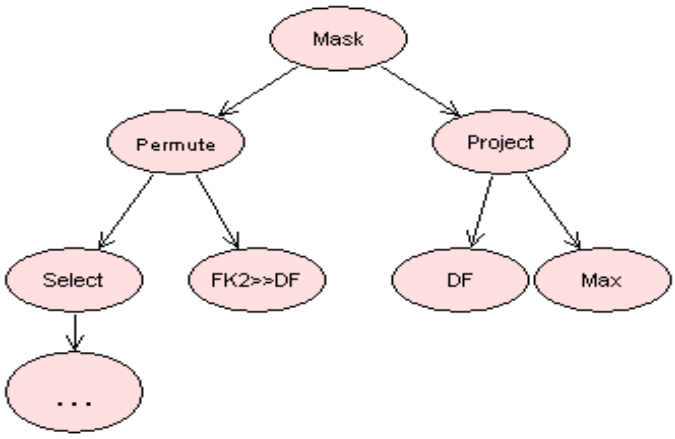
a pro každou pozici vyhodnotí zadaný predikát nebo použije binární vyhledávání a výsledný bitový řetězec vygeneruje automaticky doplňováním nul na všechny pozice kromě těch, kde byly nalezeny relevantní záznamy. K takovému rozhodnutí je potřebná pouze znalost, jestli daná projekce je setříděná podle hodnot v sloupci, ve kterém hledáme.

Jelikož informace o řazení projekcí je v databázi C-store uložena, zavedení binárního vyhledávání je triviální záležitost. Jsou dvě možnosti, jak tento princip implementovat: rozšíření definice stávajícího operátoru `Select`, nebo zavedení nového operátoru `SelectBin`. Výstupy operátorů `Select` a `SelectBin` by byly shodné, rozdíl by byl pouze v časové náročnosti. Operátor `SelectBin` by bylo možné použít pouze pro setříděné vstupy.

Pokud bychom předpokládali, že operátor `Select` systému C-store funguje čistě sekvenčně a nemáme k dispozici operátor `SelectBin`, časová složitost nalezení daného prvku by byla: *O(n)* operací *next()* namísto *O (log n)* operací *jump()*. Tento rozdíl by se projevil ve všech algoritmech, které využívají funkci *Search()*, tedy vyhledávání v setříděném poli.

5.3.2 Princip nalezení pozice odpovídajícího záznamu přes mapovací tabulku

Dalším principem využívaným v mnoha předchozích algoritmech je pro daný záznam jedné projekce nalézt pozici jemu odpovídajícího záznamu v jiné projekci za použití mapovací tabulky. Problém, který nastává v jazyku operátorů systému C-store, ilustrujeme na příkladě algoritmu 3.2 (Osa potomek).



Obr. 16 Použití mapovací tabulky na příkladu algoritmu Algoritmus 3.2 (Osa potomek)

Za použití výrazových prostředků popsaných v kapitole *5.2 Sada operátorů definovaných v C-store* není možné přistoupit pouze k jednomu záznamu mapovací tabulky. Abychom

dosáhli stejného výsledku, musíme použít operaci `Permute`, která permutuje celý sloupec obsahující daný záznam na pořadí dané mapovací tabulkou. Na obrázku 16 je znázorněn exekuční plán odpovídající vyhodnocení řádků 4-12 algoritmu 3.2 (Osa potomek). Operátor `Select` ve svém podstromu vyhledá daný prvek (ať už binárně nebo sekvenčně). Jeho výstupem je bitový řetězec reprezentující výskyt daného prvku ve sloupci. Tento bitový řetězec je následně permutován operátorem `Permute`. Aplikace operátoru `Mask` na takto permutovaný bitový řetězec a sloupec v projekci *DF* zaručí omezení projekce *DF* pouze na záznamy odpovídající záznamům vyhledaných operátorem `Select` (v tomto případě jediný záznam, protože se vyhledávalo nad sloupcem, jehož hodnoty jsou unikátní).

Tento postup je striktně založen na sloupcově orientované architektuře. Nepředpokládá čtení jednoho nebo málo záznamů. Nyní doplníme sadu základních operátorů o další operátor:

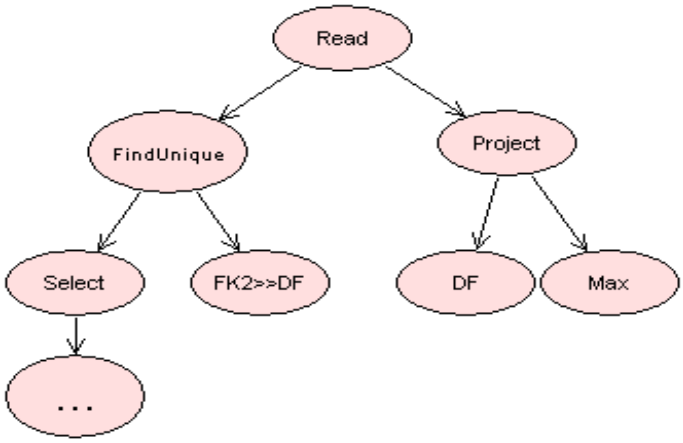
`FindUnique` – jako vstup dostane bitový řetězec s jedinou jedničkou a fakultativně mapovací tabulku. Výstupem je pozice jediné jedničky v bitovém řetězci, pokud nebyla zadána mapovací tabulka. Při zadané mapovací tabulce je výstupem hodnota mapovací tabulky na pozici, kde se v bitovém řetězci nachází jednička. Je to tedy pozice záznamu v jiné projekci, který odpovídá přes mapovací tabulku tomu záznamu, který je v bitovém řetězci označen jedničkou.

Zavedený operátor zavádí do systému C-store prvky typické pro řádkově orientované databázové systémy. Jeho použití je v souvislosti právě se sloupci, kterých hodnoty jsou unikátní. Přínos tohoto operátoru je nalezení odpovídajícího záznamu v jiné projekci za pomocí mapovací tabulky v čase *O(1)* operací *jump()*. Při použití operátoru `Permute` by tato složitost byla *O(n*log n)* operací *jump()*.

Operátor `FindUnique` zavedl typ výstupu, který není kompatibilní se vstupy ostatních operátorů. Žádný operátor systému C-store neumí zpracovat číslo pozice. Pro integraci operátoru `FindUnique` do systému zbylých operátoru zavedeme ještě jeden operátor:

`Read` – dostane na vstupu sloupec nebo projekci a *pozici*. Výstupem je pouze záznam daného sloupce nebo projekce na dané pozici.

Obrázek 17 ukazuje plán vyhodnocení dotazu, kterého výstup odpovídá přesně výstupu plánu z obrázku 16. Na obrázku 17 jsou však využity operátory `FindUnique` a `Read` pro zefektivnění časové náročnosti na vyhodnocení dotazu.



Obr. 17 Příklad použití mapovací tabulky - operátory FindUnique a Read

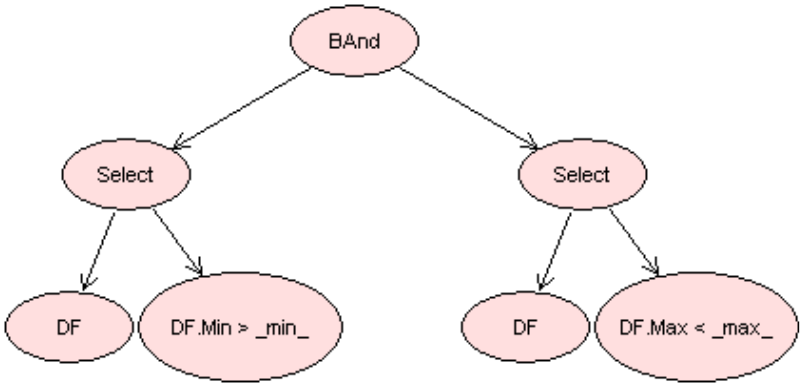
V této sekci jsme diskutovali pouze princip efektivního použití mapovací tabulky pro nalezení záznamu v jiné projekci odpovídajícího danému záznamu. V následující sekci budeme diskutovat další princip. Operátor `FindUnique` vykazuje ještě lepších výsledků v kombinaci s operátory zavedenými v následující sekci. Proto kompletní implementaci algoritmu *3.2 (Osa potomek)* uvedeme až poté, co definujeme další operátor.

5.3.3 Princip využití řazení sloupce a následného prohledávání pouze v relevantní části

Ve většině algoritmů popsaných v kapitole 3 a 4 jsme využívali znalost řazení některého ze sloupců. Pokud víme, že všechny prvky odpovědi na dotaz se díky řazení sloupce nacházejí za nebo před daným prvkem, omezili jsme vyhledávání pouze na tuto část sloupce. Jako příklad opět použijeme *Algoritmus 3.2 (Osa potomek)*.

Výsledkem složení operací na obrázku 17 je hodnota *max* daného prvku přiřazená metodou DF (označme dále jako konstantu *_max_*). Obdobným způsobem můžeme přečíst hodnotu *min* (dále používáno jako konstanta *_min_*). Řádky 12-15 algoritmu *3.2 (Osa potomek)* implementují selekci všech uzlů nacházejících se v projekci *DF* za daným uzlem a splňujících invariant maxima pro potomky (podmínka *(2)* kapitola *3.2.2 Osa potomek*). Využívá se přitom tvrzení 9. V notaci jazyka operátorů systému C-store bychom stejný dotaz vyjádřili jako bitové `And` (operace `BAnd`) bitových řetězců vyjadřujících selekci prvků s minimem

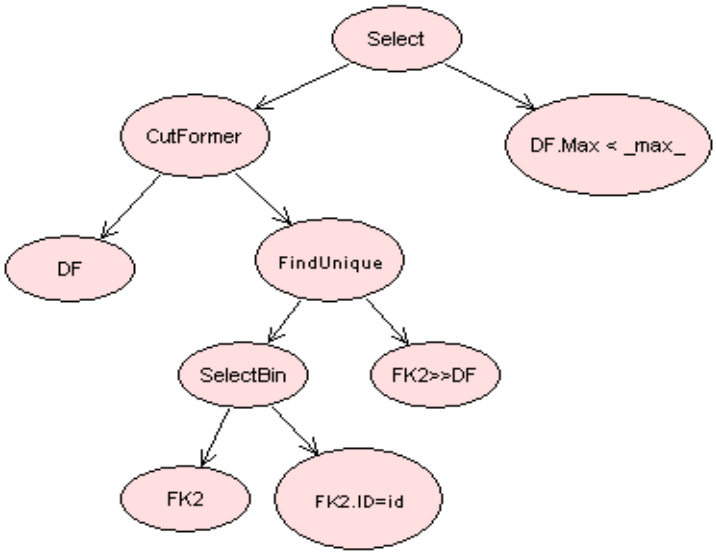
větším jako *_min_* a selekci prvků s maximem menším jako *_max_* (viz obrázek 18).



Obr. 18 Příklad selekce s podmínkou nad dvěma sloupci

Tímto způsobem však není využívána setříděnost projekce *DF* podle sloupce *Min*. Dotaz z obrázku 18 přečte 2-krát *n* po sobě jdoucích prvků. Abychom přečetli hodnoty *_min_* a *_max_*, museli jsme jistě „stát“ v projekci *DF* na pozici daného prvku. Víme-li, tuto pozici, a dále to, že projekce je setříděná podle hodnot v sloupci *Min*, můžeme si ušetřit ne jenom čtení celého sloupce *Min*, ale také čtení části sloupce *Max* (hodnoty sloupce *Max* nacházející se před daným prvkem číst nemusíme). Zavedeme operátor, který umožní eliminaci projekce nebo sloupce pouze na prvky za danou pozicí:

`CutFormer` – omezí vstupní projekci nebo sloupec vynecháním záznamů před danou pozicí.



Obr. 19 Použití kombinace operátoru FindUnique a CutFormer

Kombinací operátoru `FindUnique` a operátoru `CutFromer` (viz obrázek 19) dostaneme

Algoritmus 3.2 (Osa potomek) zapsaný v jazyku operátorů systému C-store způsobem, který zachovává jeho časovou složitost popsanou v kapitole *3.2.2 Osa potomek*, až na důsledek tvrzení 10, tedy ukončení prohledávání ve sloupci *Max*, v momentě kdy je nalezen první prvek nesplňující invariant maxima. Řešení této optimalizace implementujeme dalším operátorem:

`CutFurther` – na vstupu dostane predikát `Pred` a sloupec nebo projekci `Proj`. Na výstup postupně posílá záznamy projekce `Proj` dokud nepřestane platit predikát `Pred`.

Operátor `CutFurther` tedy implementuje klauzuli `stop where` SQL dotazu (viz poznámka[1]). Druhá možnost řešení by byla implementace pomocí vnořené procedury.

5.4 Provedené experimenty

Jako součást této práce byly provedeny experimenty založené na úvahách z předchozí kapitoly. Cílem bylo ukázat, zda se opravdu zlepší časová složitost jednotlivých dotazů přidáním nově definovaných operátorů. Pro tyto experimenty jsme si zvolili *Algoritmus 3.2 (Osa potomek)* a *Algoritmus 3.5 (Osa předek – DF metoda)*. Volba algoritmů byla založena na potřebě otestovat, pokud možno, co nejvíce nově zavedených operátorů. Zatímco *Algoritmus 3.2 (Osa potomek)* využívá operátory *FindUnique*, *CutFormer* a *CutFurther*, *Algoritmus 3.5 (Osa předek – DF metoda)* otestuje pouze dvojici *FindUnique* a *CutFormer*. Můžeme tedy vidět výsledky jak více, tak méně optimalizovaných algoritmů. Nebyla provedena měření efektivity binárního a sekvenčního vyhledávání. Jednak proto, že jde o všeobecně známé metody a za druhé proto, že by integrace binárního vyhledávání do použité databázové implementace nebyla triviální.

Pro účely provádění experimentů jsme si propůjčili jednoduchou implementaci databáze C-store popsanou v práci [3]. Doimplementovali jsme do ní potřebné operátory. Na základě kolekce reálných XML dokumentů jsme postavili databázi odpovídající modelu popsaném v kapitole *3.1 Datová struktura*. Poté jsme pro algoritmy na obou sledovaných osách (potomek a předek) připravili sadu náhodně vygenerovaných dotazů. Tyto dotazy jsme vyhodnotili jak za použití přidaných operátorů, tak i bez nich – pouze v původně připraveném rozhraní. Doba vyhodnocování dotazů byla pokaždé měřena.

Výsledky potvrdily markantní rozdíl ve prospěch systému rozšířeného o námi definované operátory (přehled výsledků a informace o zdrojových datech společně se zdrojovými

i přeloženými soubory testů lze najít na přiloženém CD).

5.5 Detailní popis implementace vybraných algoritmů

V této kapitole uvedeme příklady kompletní implementace algoritmů *3.2 (Osa potomek)* a *3.6 (Skoč a jdi)*[5]. Tyto by měly sloužit jako návod pro implementaci jakéhokoliv dalšího z algoritmů.

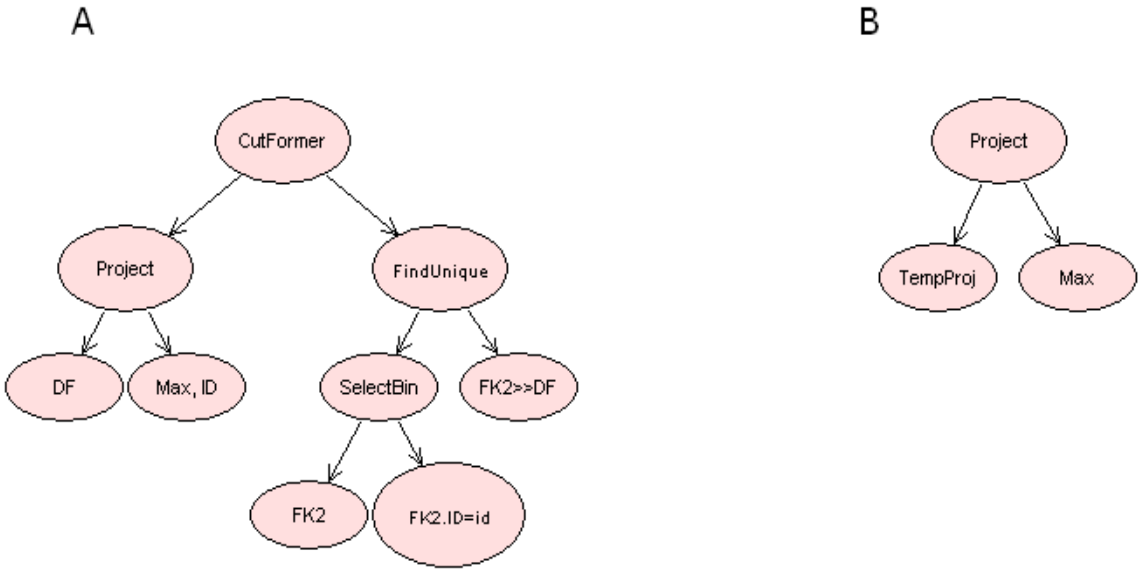
Algoritmus 5.1 (Implementace algoritmu 3.2 (Osa potomek))

Algoritmus dostane na vstup *id* daného uzlu a výstupem je množina *P* všech jeho potomků. Ta je na začátku prázdná. Algoritmus používá dotazy A, B zakreslené formou skládání operátorů systému C-store na obrázku 20.

Vstup: id
Výstup: P = {}

```
1       // Přečtení hodnoty _max_ daného uzlu
2       new Cursor TempProj = query(A, id);
3       new Cursor c = query(B, TempProj);
4       _max_ = c.next(Max);
5       Delete c;
6       // Nalezení všech potomků daného uzlu jako sekvence od daného
        uzlu po prvního následníka (Max > _max_)
7       m = _max_;
8       // První záznam projekce je daný prvek
9       TempProj.next();
10      // Ostatní řádky už jsou zajímavé
11      (m,id_p) = TempProj.next(Max, ID);
12      while m < _max_:
13           P.add(id_p);
14           (m,id_p) = TempProj.next();
15      Delete TempProj;
```

[5] Ukážeme popis složitějšího algoritmu, i když v provedených experimentech jsme se omezili pouze na DF metodu. Omezení vycházelo z faktu, že pro metodu „Skoč a jdi“ nebylo naimplementováno binární vyhledávání. Kombinace metody FK a DF by tedy neměla smysl.



Obr. 20 Dotazy pro algoritmus Algoritmus 5.1 (Implementace algoritmu 3.2 (Osa potomek))

Algoritmus 5.2 (Implementace algoritmu 3.6 (Skoč a jdi))

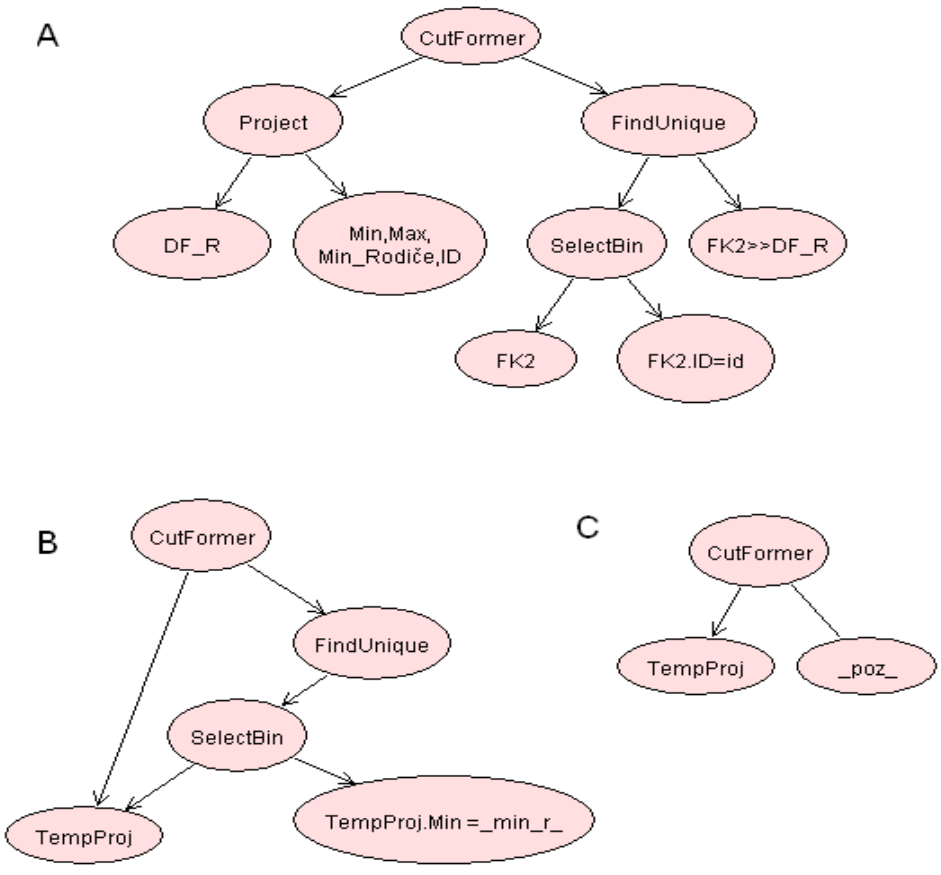
U algoritmu *3.6 (Skoč a jdi)* se používá funkce *Previous()*. Snadno ji převedeme na funkci *Next()*. Místo projekce *DF* použijeme projekci *DF_R*, která je řazena obráceně jako projekce *DF*. Pak už můžeme všude namísto funkce *Previous()* použít funkci *Next()*.

Algoritmus dostane na vstup *id* daného uzlu a výstupem je množina *P* všech jeho předků. Ta je na začátku prázdná. Algoritmus používá dotazy A, B, C zakreslené formou skládání operátorů systému C-store na obrázku 21.

Vstup: id
Výstup: P = {}

```
1       // Přečtení důležitých hodnot daného uzlu
2       new Cursor TempProj = query(A, id);
3       (min, max, _min_r_, id) = TempProj.next(Min, Max, Min_rodiče,
        ID);
4       // V cyklu hledání rodiče, rodiče rodiče, atd.
5       while _min_r is not null:
6           // Rozhodnutí mezi metodou „skoč" a „jdi" – viz kapitola
            3.2.4.4
7           // Od pozice aktuálního prvku závisí časová složitost
            v případě metody skákání
8           i = TempProj.position();
9           if log(i)*konst.ts <=((min – min_r – 1)/2 + 1)*konst.tn :
```

```
10              // Metoda skákání
11              TempProj = query(B, TempProj, _min_r_);
12          else:
13              // Metoda chůze
14              m = max;
15              while m <= max:
16                  m = TempProj.next(Max);
17              // Nalezen další předek
18              _poz_ = TempProj.position();
19              TempProj = query(C, TempProj, _poz_);
20          // V každém případě TempProj má jako první řádek dalšího
            předka
21          (min, max, _min_r_, id) = TempProj.next(Min, Max,
            Min_rodiče, ID);
22          P.add(id);
```



Obr. 21Dotazy pro algoritmus Algoritmus 5.2 (Implementace algoritmu 3.6 (Skoč a jdi)

V této kapitole jsme uvedli návod, jakým způsobem lze přistupovat k implementaci jednotlivých algoritmů. Tato kapitola předpokládala rozšíření základní sady operátorů systému C-store. Při zachování původně navržené sady operátorů bude implementace vypadat

obdobně, změní se pouze časová náročnost jednotlivých algoritmů.

6 Závěr

Závěrem shrneme průběh i výsledky práce. Primárním cílem bylo ukázat, jak lze do sloupcově orientovaného databázového systému uložit XML data s možností získávat rychle cesty v XML stromu. Byly zváženy dva modely. První model nevyužívá (případně existující) schéma XML dat. Druhý model staví databázové schéma na základě schématu XML dokumentu. Ukázalo se, že znalost schématu XML dat není pro řešenou problematiku přínosem.

V průběhu práce jsme uvažovali použití systému C-store. U všech navrhovaných algoritmů jsme se tedy snažili co nejvíce využít jeho přirozených vlastností. Pro problematiku vyhledávání cest v XML databázi byly nejvíce zajímavé dvě vlastnosti: ukládání dat po sloupcích a redundanci. Kombinace těchto dvou vlastností má za následek možnost čtení sekvence po sobě jdoucích dat bez zbytečného zatížení čtením dat nerelevantních pro výsledek dotazu.

U modelu dat bez schématu se podařilo navrhnout algoritmy tak, aby většina z nich měla (po nalezení daného prvku v databázi) časovou složitost přečtení $O(m)$ prvků uložených v sekvenci za sebou, kde m je počet uzlů, které jsou výsledkem dotazu. Jediný algoritmus, který má horší časovou složitost je algoritmus hledání všech předků daného prvku. U modelu dat se schématem je časová složitost většinou horší, pokud se nevyužije distribuované řešení.

Aby se u mnoha algoritmů dosáhlo těchto výsledků, byly používány různé principy, jako například využití setřídění projekce podle určitého sloupce, nalezení odpovídajícího záznamu přes mapovací tabulku nebo pouze matematické vlastnosti použitých metod. Tyto principy jsme nakonec konfrontovali s reálnou možností jejich uplatnění v rozhraní C-store definovaným v práci [1]. Ukázalo se, že některé principy v tomto systému nejsou realizovatelné. Proto jsme navrhli rozšíření základní sady operátorů pro vyhodnocení dotazů v C-store. Provedli jsme experimentální měření, které potvrdilo účelnost zavedených operátorů.

Prvky, o které by tímto způsobem byl systém C-store rozšířen, zavádějí obecnou formu optimalizace některých dotazů, často využívanou v řádkově orientovaných relačních databázích. Vznikl by tak systém, který kombinuje výhody ukládání po sloupcích s přístupem k vyhodnocování dotazů způsobem, že není nutné číst celý sloupec, pokud se využije znalost řazení celé projekce. Máme za to, že toto doplnění systému C-store by bylo vhodné i pro jiné

typy aplikací než XML databázi. Tuto problematiku necháváme otevřenou. Navíc doplnění sady operátorů systému C-store si nečiní nároky být kompletní. Pro případný další vývoj doporučujeme rozšíření systému C-store uvedené v této práci brát pouze jako motivaci a detailně dodefinovat celou sadu nových operátorů v obecnějším kontextu.

Přílohou k práci je i prototypová implementace některých navržených algoritmů. Implementace je postavena na stávající definici operátorů pro vyhodnocování dotazů v C-store. Proto nebyla provedena měření, která by srovnala rychlost vyhledávání cest v XML dokumentu touto metodou a metodou řádkově orientované databáze. Jako další cíl výzkumu v této oblasti navrhujeme vypracování srovnávací studie jejíž součástí by mělo být:

- navržení vhodných metod pro implementaci odpovídajících dotazů v řádkově orientovaném úložišti dat,
- jejich kompletní implementace,
- kompletní implementace algoritmů navržených v této práci (včetně rozšíření systému C-store o potřebné operátory),
- porovnání obou přístupů na reálných datech.

Seznam použité literatury

[1] Stonebraker, M.:C-store: A Column-oriented DBMS. In: Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005.

[2] Mlýnková, I., Pokorný, J.: XML in the World of (Object-)Relational Database Systems. In: Information Systems Development Advances in Theory, Practice and Education. Vasilecas, O.; Caplinskas, A.; Wojtowski, G.; Wojtowski, W.; Zupancic, S(Eds.), Springer Science+Business Media, Inc., 2005, pp. 63-76

[3] Částek, P.: C-store: Úložiště relačních dat po sloupcích. Diplomová práce.[6]

[6] V době psaní této práce zatím neobhájena. Byla použita především implementace databáze C-store pro účely provedení experimentů.