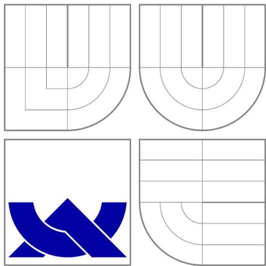
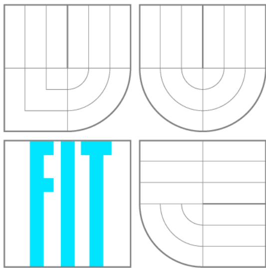


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TEMPORÁLNÍ XML DATABÁZE

TEMPORAL XML DATABASES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — TOMÁŠ KUNOVSKÝ
AUTHOR

VEDOUCÍ PRÁCE — RNDr. MAREK RYCHLÝ, Ph.D.
SUPERVISOR

BRNO 2016

Zadání diplomové práce/18028/2015/xkunov05

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Kunovský Tomáš, Bc.**
Obor: Informační systémy
Téma: **Temporální XML databáze**
Temporal XML Databases
Kategorie: Databáze

Pokyny:

1. Seznamte se s principy a s existujícími implementacemi temporálních a XML databází a s dotazovacími jazyky, které používají. Prostudujte a popište problematiku uložení temporálních dat v XML dokumentech.
2. Navrhněte novou či přizpůsobte stávající XML databázi či rozhraní a implementaci přístupu k XML dokumentům pro uložení temporálních dat při zachování principů temporálních databází.
3. Po konzultaci s vedoucím navržené řešení implementujte vč. testů a příkladů demonstrujících klíčové vlastnosti řešení.
4. Výsledky zveřejněte jako open-source, zhodnoťte a navrhněte případná rozšíření.

Literatura:

- Richard T. Snodgrass. Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann Publishers, Inc., San Francisco, July, 1999, 504+xxiii pages, ISBN 1-55860-436-7. [http://www.cs.arizona.edu/people/rts/tdbbook.pdf, http://www.cs.arizona.edu/people/rts/cdrom.tgz]
- Yong Tang, Xiaoping Ye, Na Tang. *Temporal Information Processing Technology and Its Applications*. Springer Science & Business Media, 2011, ISBN 978-3-642-14959-7.
- Flavio Rizzolo, Alejandro A. Vaisman. *Temporal XML: modeling, indexing, and query processing*. The VLDB Journal: 17(5), 2008, pp. 1179-1212. [http://dx.doi.org/10.1007/s00778-007-0058-x]
- Jiří Tomek. *TSQL2 interpret nad relační databází*. Diplomová práce, FIT VUT v Brně, 2009. [https://www.fit.vutbr.cz/study/DP/DP.php?id=8603]
- Jakub Horčička. *Temporální rozšíření pro Java Data Objects*. Diplomová práce, FIT VUT v Brně, 2012. [https://www.fit.vutbr.cz/study/DP/DP.php?id=12079]

Při obhajobě semestrální části projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

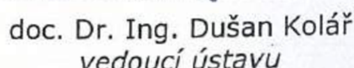
Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.



doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Primárním cílem práce je implementace temporální XML databáze v jazyce Java. Jsou zde popsány databáze ukládající XML dokumenty a temporální databáze s důrazem na jejich dotazovací jazyky a je zde rozebrána problematika uložení dat v temporálních databázích. Zdrojové kódy výsledné aplikace jsou zveřejněny jako open-source.

Abstract

The primary goal of this work is a implementation of temporal XML database in Java. There are described databases for XML documents and temporal databases with emphasis on their query languages and problem data storing is also analyzes for temporal databases. Source codes of the resulting application are public as open-source.

Klíčová slova

temporální, databáze, XML, Java, SQL, XPath, XQuery, Oracle, nativní XML databáze, čas platnosti, čas transakce

Keywords

temporal, databases, XML, Java, SQL, XPath, XQuery, Oracle, native XML databases, valid time, transaction time

Citace

KUNOVSKÝ, Tomáš. *Temporální XML databáze.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Rychlý Marek.

Temporální XML databáze

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Tomáš Kunovský
25. května 2016

Poděkování

Děkuji svému vedoucímu práce RNDr. Marku Rychlému, Ph.D. za odborné vedení a náměty, které mi při řešení této práce poskytl.

© Tomáš Kunovský, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **4**

2 Formát XML **6**
- 2.1 Typy dokumentů 6
  - 2.1.1 Datové XML dokumenty 6
  - 2.1.2 Dokumentově zaměřené XML dokumenty 7

3 Dotazovací jazyky pro XML **8**
- 3.1 XPath 8
  - 3.1.1 Model dat 8
  - 3.1.2 Výraz 9
  - 3.1.3 Krok 10
  - 3.1.4 Zkracování syntaxe 10
- 3.2 XQuery 11
  - 3.2.1 Struktura dotazu 12
  - 3.2.2 Vytváření elementů 13

4 XML v relačních databázích **15**
- 4.1 XML-enabled 15
- 4.2 Ukládání XML dokumentů 15
  - 4.2.1 Uložení do relační tabulky 15
  - 4.2.2 Uložení do CLOB 17
  - 4.2.3 Objektově-relační uložení 17
  - 4.2.4 Uložení jako BLOB 17
- 4.3 Standard SQL 17
- 4.4 Oracle databáze 11g 18

5 XML databáze **20**
- 5.1 Nativní XML databáze 20
- 5.2 Efektivní uložení XML dokumentu 20
  - 5.2.1 Abstraktní model 20
  - 5.2.2 Implementace modelu 21

6 Temporální relační databáze **24**
- 6.1 Reprezentace času 24
- 6.2 Datový model 25
- 6.3 Restrukturalizace a shlukování 25
- 6.4 Mazání 25

6.4.1 S transakčním časem . . . 25
6.4.2 S časem platnosti . . . 26
6.4.3 Kombinace času platnosti a transakčního času . . . 26
6.5 Aktualizace . . . 26
6.5.1 S transakčním časem . . . 27
6.5.2 S časem platnosti . . . 27
6.5.3 Kombinace času platnosti a transakčního času . . . 27
6.6 Vkládání . . . 28
6.7 TSQL2 . . . 28
6.7.1 Transakční čas a čas platnosti . . . 28
6.7.2 Druhy tabulek . . . 28
6.7.3 Typ SURROGATE . . . 28
6.7.4 Popis jazyka . . . 29
6.8 Shrnutí . . . 29

7 Temporální XML databáze **31**
7.1 Abstraktní model . . . 31
7.2 Platnost uzlu . . . 31
7.3 Souvislá cesta a maximální souvislá cesta . . . 32
7.4 Podmínky konzistence . . . 33
7.4.1 První podmínka . . . 33
7.4.2 Druhá podmínka . . . 33
7.4.3 Třetí podmínka . . . 34
7.4.4 Čtvrtá podmínka . . . 34
7.4.5 Pátá podmínka . . . 35
7.4.6 Šestá podmínka . . . 35
7.5 Možné nekonzistence . . . 35
7.5.1 Nekonzistence typu I . . . 36
7.5.2 Nekonzistence typu II . . . 37
7.5.3 Nekonzistence typu III . . . 38
7.5.4 Nekonzistence typu IV . . . 39
7.6 Rozšíření XPath . . . 39
7.7 Uložení datového modelu . . . 40
7.7.1 Uložení souvislých cest . . . 40
7.7.2 Třídy LCP . . . 40
7.7.3 Tabulky LCP . . . 42
7.7.4 Třídy CP . . . 43
7.7.5 Tabulky CP . . . 43
7.8 Vyhodnocení XPath dotazů . . . 44
7.9 Aktualizace s transakčním časem . . . 46
7.9.1 Vložení nového uzlu . . . 46
7.9.2 Odstranění uzlu . . . 47
7.9.3 Aktualizace obsahové hrany . . . 48

8 Analýza hotových řešení **50**
8.1 Temporální rozšíření objektově-relačního mapování . . . 50
8.2 TSQL2 interpret nad relační databází . . . 51

9 Návrh **53**
9.1 Uložení dat v databázi 53
9.2 Návrh aplikace 56
9.3 Návrh gramatiky jazyka 60
9.4 Sémantika jazyka 64
9.4.1 Vložení nového uzlu 64
9.4.2 Odstranění uzlu 65
9.4.3 Změna umístění elementu nebo atributu 65
9.4.4 Temporální XPath dotaz 66

10 Implementace **67**
10.1 Filtry 67
10.2 Referenční integrita 67
10.3 Přenositelnost mezi databázemi 68
10.4 Podmínky konzistence 68
10.5 Jmenné prostory 68
10.6 Textové uzly 68
10.7 Aktualizace 69
10.8 Řazení 69
10.9 Validace 69
10.10Zamykání tabulek 70

11 Testování **71**
11.1 Ověření správnosti výstupu 71
11.2 Vložení nového elementu 72
11.3 Smazání elementu 72
11.4 Přesun elementu 73
11.5 Řazení elementů 74
11.6 XPath dotazy 74
11.7 Zhodnocení testů 76

12 Závěr **80**

Literatura **81**

Přílohy **83**
Seznam příloh 84

A Obsah CD **85**

Kapitola 1

Úvod

Jazyk XML se běžně používá jako standardní formát pro výměnu dat a dokumentů mezi aplikacemi. S tím je spojeno i trvalé ukládání samotných dat, a proto se v dnešní době stále častěji setkáváme s potřebou ukládat XML dokumenty do databáze. U uložených dokumentů je kromě načtení celého dokumentu vyžadována možnost dotazování nad dokumentem a aktualizace jednotlivých jeho částí. K tomu lze využít objektově-relačních databází a vznikla i řada rozšíření klasických relačních databází. V databázích byl zaveden pro XML dokument nový datový typ a nové funkce, které s tímto typem pracují.

Relační model se od modelu XML poměrně liší. Je to hlavně nepravidelná struktura XML dokumentů, co celou věc značně komplikuje. XML dokumenty mohou vytvářet složitou stromovou strukturu, zatímco struktura relačního modelu je plochá. Dalším velkým problémem je, že XML data mají určené pořadí (například kapitoly v knize), a dodržení pořadí elementů může být proto nezbytné. Kromě toho součástí XML dokumentu jsou i jeho metadata a často se také setkáváme s chybějícími elementy a atributy. Naproti tomu metadata jsou v relační databázi striktně oddělena a chybějící hodnoty jsou zde spíše výjimkou.

Proto vedle relačních databází vznikly nativní XML databáze, umožňující přirozenější způsob uložení dokumentů. Použití takových databází je vhodné zejména u XML dokumentů, jejichž obsahem jsou dokumenty jako knihy, články, XHTML dokumenty, katalogy zboží či záznamy o pacientech. Výhodami je efektivnější indexování, přesné zachování obsahu XML dokumetu, ten se může po uložení XML dokumentu do relační databáze a následném načtení mírně změnit, a možnost nerespektovat schéma dokumentu. Nativní XML databáze nepoužívají dotazovací jazyk SQL a namísto něho se zde používají především jazyky XPath a XQuery, ty dokáží lépe pracovat se stromovou strukturou.

V některých případech je vyžadováno, aby v databázi kromě aktuálních dat byla uložena i jejich historie. Například katastrální úřad potřebuje, aby u každého pozemku bylo možné kromě aktuálního majitele dohledat i předchozí majitele. Pro relační databáze vznikla celá řada řešení, jakým způsobem rozšířit záznamy o čas platnosti a jak s ním poté pracovat. U nativních XML databází tato podpora chybí. Cílem této práce bude právě návrh a implementace rozšíření databází o možnost temporálního dotazování nad XML dokumentem.

Práce se věnuje následujícím tématům. Druhá kapitola popisuje dokumenty ve formátu XML. Třetí kapitola popisuje jazyky XPath a XQuery pro dotazování nad těmito dokumenty. Čtvrtá kapitola popisuje možnosti a problémy spojené s ukládáním XML dokumentů do databáze. Pátá kapitola krátce pojednává o XML databázích. Šestá kapitola je věnována temporálním relačním databázím. Jsou v ní popsány principy temporálních databází a jakým způsobem je nutné rozšířit data v relačních databázích. Sedmá kapitola podrobně

popisuje fungování temporální XML databáze. To je zároveň demonstrováno na obsáhlém příkladu takové databáze. Osmá kapitola popisuje řešení temporálních databází dvou podobných projektů. Devátá kapitola je návrh. Popisuje uložení dat v databázi a dekompozici navržené knihovny. Také je v ní popsána gramatika jazyka pro dotazování a modifikace uložených dokumentů v databázi. Desátá kapitola se zabývá problémy implementace a jedenáctá kapitola testuje použitelnost výsledného řešení.

Kapitoly dva, tři, šest, osm a část kapitol jedna, čtyři, pět a sedm byly převzaty ze semestrálního projektu. Pro psaní úvodu byly použity zdroje [10], [11], [14] a [7].

Kapitola 2

Formát XML

Tato kapitola popisuje dokumenty ve formátu XML. Jsou zde vysvětleny základní pojmy objevující se v následujících kapitolách. Bylo zde čerpáno ze zdroje [15].

2.1 Typy dokumentů

Podle obsahu lze XML dokumenty rozdělit do dvou kategorií. První kategorie se označuje jako datové XML dokumenty nebo datově orientované XML dokumenty a druhá jako dokumentově zaměřené XML dokumenty.

2.1.1 Datové XML dokumenty

Jedná se o takové XML dokumenty, které se typicky používají pro přenos dat například mezi aplikacemi. Běžný životní cyklus je takový, že data z databáze se převedou do podoby XML dokumentu a ten se pošle nebo přesune. Tedy výsledkem aplikace je XML dokument, který přijme jiná aplikace, která ho zpracovává a typicky takto přenesená data zase uloží do databáze. Pro tento typ XML dokumentů je charakteristické, že je obvykle používán k přenosu, struktura je poměrně jemná, data jsou hodně strukturovaná a nezáleží na pořadí jednotlivých elementů v XML dokumentech, rozhodující je název elementu případně atributu. Častými případy takových dokumentů jsou faktury, exporty dat z databáze, objednávky a obecně dokumenty vytvořené podle šablony. Aplikace pracující s takovýmto typem dokumentů se označují jako datově zaměřené a tento typ dokumentu je určen pro strojové zpracování.

Na příkladu 2.1 je zobrazen datový XML dokument. Je zde vidět, že granularita struktury dokumentu je oproti příkladu dokumentově zaměřeného dokumentu 2.2 jemnější. U příkladu 2.1 si lze představit data z relační databáze a je vidět, že nezáleží na pořadí elementů. Je možné přehodit například datum vystavení a datum splatnosti. Podstatná je tedy pouze hodnota elementu nebo atributu. V relační databázi by tento dokument byl uložen v tabulkách Faktura a Dodavatel, kde ICO by hrálo roli cizího klíče.

```
<?xml version="1.0" encoding="iso-8859-2"?>
<FAKTURA cislo_faktury="10022006/159">
  <DATUM_VYSTAVENI>1.4.2006</DATUM_VYSTAVENI>
  <DATUM_SPLATNOSTI>1.5.2006</DATUM_SPLATNOSTI>
  <DODAVATEL>
    <JMENO>Jan Novák</JMENO>
```

```
    <ULICE>Božetěchova 2</ULICE>
    <MESTO>Brno</MESTO>
    <PSC>61200</PSC>
    <ICO>12345678</ICO>
  </DODAVATEL>
...
</FAKTURA>
```

Příklad 2.1: Datový XML dokument.

2.1.2 Dokumentově zaměřené XML dokumenty

Obsahem dokumentu je textový dokument obsahující části jako nadpis, odstavce, abstrakt a autora. Převažuje textová informace a počet elementů je oproti datovým XML dokumentům menší. Pořadí elementů je významné, kupříkladu záleží na pořadí odstavců. Tento typ dokumentu je určen pro čtení či zpracování lidmi, což je dáno jeho obsahem. Data zpravidla nepochází z databáze. Častými případy takových dokumentů jsou knihy, e-maily a webové stránky v XHTML. Odpovídající aplikace pracující s touto kategorií dokumentů se označují jako dokumentově zaměřené a takovéto aplikace mohou například zobrazovat obsah dokumentu bez značek jako textový dokument.

Na příkladu 2.2 je ukázán dokumentově zaměřený XML dokument. Na dokumentu je vidět, že pořadí prvků je podstatné, například nelze prohodit pořadí elementů *NADPIS1* a *ODSTAVEC*.

```
<?xml version="1.0" encoding="iso-8859-2"?>
<PRIRUCKA>
  <NADPIS1>Uživatelská příručka k systému</NADPIS1>
  <ODSTAVEC>
    Pokud nejste zaregistrován do systému a rád byste se
       zapsal na nějaký kurz, tak se musíte nejdříve
    <ODKAZ url="/registrace">zaregistrovat</ODKAZ>.
    Obdržíte tak své klientské číslo a heslo, pod kterými se
       budete přihlašovat.
  </ODSTAVEC>
</PRIRUCKA>
```

Příklad 2.2: Dokumentově zaměřený XML dokument.

Kapitola 3

Dotazovací jazyky pro XML

Podobně jako vznikl jazyk SQL pro dotazování nad relační databází, vznikly i jazyky pro práci s XML dokumenty. Na rozdíl od relačních databází může XML dokument vytvářet složitou nepravidelnou stromovou strukturu. Z tohoto důvodu bychom si nevystačili pouze s jazykem SQL, ale je třeba použít i jazyk jako XPath. Navíc pro složitější dotazování vznikl jakyk XQuery, který dále jazyk XPath rozšiřuje. Oba tyto jazyky jsou standardizovány W3C. Bylo zde čerpáno ze zdrojů [11], [3] a [7].

3.1 XPath

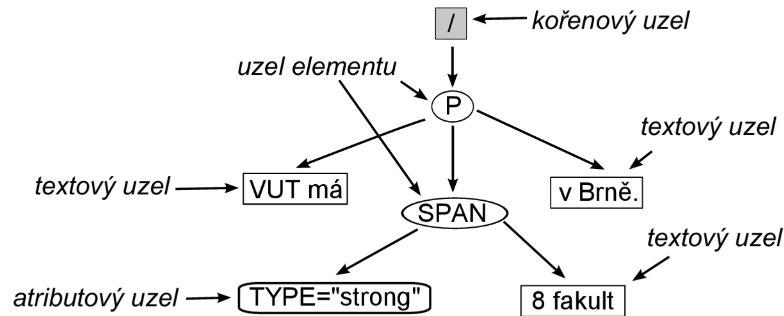
Tento jazyk se používá pro navigaci v XML dokumentech. Příkladem může být například zjištění čísla účtu z objednávky. Většinou se používá ve spojení s dalším dotazovacím jazykem jako je SQL nebo XQuery. Cílová část XML dokumentu je popsána jako cesta podobným způsobem jako cesta k souboru v klasickém operačním systému. Existují dvě verze tohoto jazyka, v této práci bude popsána verze 2.0.

3.1.1 Model dat

Dokument XML je převeden na strom, tedy je zde jediný kořen. Každý prvek v dokumentu je reprezentován uzlem. Rozlišují se zde následující typů uzlů.

- kořenový uzel
- uzly elementů
- textové uzly
- atributové uzly
- uzly pro komentáře
- uzly instrukcí pro zpracování
- uzly jmenných prostorů

Podobně jako v DOM i zde jsou vazby mezi prvky reprezentovány hranami. Nejčastější typy uzlů jsou zobrazeny na obrázku 3.1. Ten popisuje strom pro fragment v příkladu 3.1. Uzly jsou uspořádány přesně podle dokumentu, ten lze zpětně ze stromu získat pomocí průchodu preorder.



Obrázek 3.1: Popis uzlů.

```
<P>VUT má <SPAN TYPE="strong">8 fakult</SPAN> v Brně.</P>
```

Příklad 3.1: Ukázkový XML fragment.

3.1.2 Výraz

XML dokument je možné si představit jako strom. Příklad takového stromu je zobrazen na obrázku 3.1. Výraz jazyka XPath je cesta skládající se z kroků oddělených lomítkem. Výraz se interpretuje zleva doprava po krocích a každý interpretovaný krok definuje, jakým směrem se ve stromě pohybujeme a které uzly vybíráme. Vybrané uzly poté slouží jako vstup následujícího kroku, ten se aplikuje na každý vstupní uzel. Stejně jako v souborovém systému, i zde cesta může být jak relativní tak absolutní. Absolutní cesta začíná lomítkem a aplikuje se od kořene stromu. U relativní cesty se předpokládá, že zde byl již předtím vybrán nějakým způsobem aktuální uzel, který výrazu pak slouží jako vstup.

Příklad 3.2 ukazuje použití relativní cesty a příklad 3.3 použití absolutní cesty na příkladu dokumentu 3.4.

```
moznosti/moznost
```

Příklad 3.2: Příklad relativní cesty v XPath.

```
/anketa
```

Příklad 3.3: Příklad absolutní cesty v XPath.

```
<anketa>
  <otazka>Kolik hodin strávíte denně u počítače?</otazka>
  <moznosti>
    <moznost hlasu='12'>12-15 hodin</moznost>
    <moznost hlasu='5'>15-20 hodin</moznost>
    <moznost hlasu='15'>20-24 hodin</moznost>
    <moznost hlasu='10'>Můj počítač nefunguje</moznost>
  </moznosti>
</anketa>
```

Příklad 3.4: XML pro popis jazyka XPath.

3.1.3 Krok

Syntaxe kroku má tvar *osa::test-na-uzel[predikát]**. První část kroku je identifikátor osy, ten určuje směr pohybu ve stromě a pokud není uveden, doplní se za něj výchozí hodnota child::, možné identifikátory osy jsou popsány v tabulce 3.1.

Identifikátor	**Vyhodnocené uzly**
child::	přímí potomci aktuálního uzlu
descendand::	všichni potomci aktuálního uzlu
descendand-or-self::	aktuální uzel a všichni potomci
self::	aktuální uzel
ancestor-or-self::	aktuální uzel a všichni jeho předci
ancestor::	všichni předci aktuálního uzlu
parent::	rodič aktuálního uzlu
following::	Všechny uzly, které se v toku XML dokumentu nacházejí za aktuálním uzlem.
preceding::	Všechny uzly, které se v toku XML dokumentu nacházejí před aktuálním uzlem.
following-sibling::	všichni následující sourozenci aktuálního uzlu
preceding-sibling::	všichni předcházející sourozenci aktuálního uzlu
attribute::	atributy aktuálního uzlu
namespace::	deklarované jmenné prostory

Tabulka 3.1: Identifikátory osy.

Test na uzel a predikát pak dále filtrují cílovou množinu uzlů. Obecně je možné v jednom kroku uvést libovolný počet predikátů v hranatých závorkách, tedy nula až n, stejného efektu pak dosáhneme, pokud spojíme všech n predikátů pomocí konjunkce do jednoho predikátu. Pokud je místo predikátu uvedeno v hranatých závorkách pouze číslo (pozice), vybere se uzel s danou pozicí. V podmínkách lze dále použít funkce uvedené v tabulce 3.2.

Jediná nutně uvedená část kroku je tedy test na uzel, zde existují dvě možnosti. První možností je určit vybrané uzly názvem, tedy uvedeme název a vyberou se XML elementy s daným názvem. Druhou možností je vybrat uzly zadaného typu, zde se obvykle používá volba *text()* pro výběr textových uzlů.

Příklad 3.5 vrátí elementy *moznost*, jejichž atribut *hlasu* má hodnotu rovnu *5* a příklad 3.6 vrátí v pořadí poslední element *otazka* v příkladu dokumentu 3.4.

```
/anketa/moznosti/moznost[attribute::hlasu="5"]
```

Příklad 3.5: Ukázka predikátu v XPath s podmínkou.

```
/anketa/otazka[last()]
```

Příklad 3.6: Ukázka predikátu v XPath s pozicí.

3.1.4 Zkracování syntaxe

U některých často používaných konstrukcí je možné výraz zjednodušit. Velice často se například používají dvě lomítka k překonání víceúrovňové struktury. Příklad 3.7 vrátí všechny

Funkce	Popis
position()	pořadové číslo aktuálního XML uzlu
last()	pořadové číslo posledního uzlu
count()	počet uzlů v daném kontextu
name()	úplné kvalifikované jméno aktuálního XML uzlu
string()	Převede libovolný objekt na řetězcovou hodnotu.
number()	Převede libovolný objekt na číselnou hodnotu.
boolean()	Převede libovolný objekt na logickou hodnotu.
concat()	Spojí řetězce v parametrech.
sum()	Objekty v parametru jsou zkonvertovány na čísla, a funkce vrátí jejich součet.
not()	negace logického výrazu

Tabulka 3.2: Často používané funkce v XPath.

elementy *moznost*, jejichž atribut *hlasu* má hodnotu rovnou *1*. Také je zde ukázána možnost nahradit identifikátor osy *attribute::* zavináčem. Příklad 3.8 zase vybere všechny textové uzly v dokumentu.

```
//moznost[@hlasu="1"]
```

Příklad 3.7: Ukázka použití dvou lomítek.

```
//text()
```

Příklad 3.8: Ukázka použití dvou lomítek a výběru uzlů zadaného typu.

Chceme-li vybrat všechny poskytnuté elementy, můžeme použít hvězdičku, jak je to ukázáno na příkladu 3.9. Ten vrací všechny elementy podřízené kořenovému elementu *anketa*.

```
/anketa/*
```

Příklad 3.9: Ukázka použití hvězdičky pro testování uzlu.

Podobně jako v souborovém systému i zde je možné v cestě použít jednu nebo dvě tečky, sémantika je prakticky stejná. Příklad 3.10 vrátí všechny podřízené elementy aktuálního uzlu a příklad 3.11 vybere všechny sourozence aktuálního elementu včetně jeho samotného.

```
./*
```

Příklad 3.10: Ukázka tečky.

```
../*
```

Příklad 3.11: Ukázka dvou teček.

3.2 XQuery

Jedná se o funkcionální jazyk, kde na rozdíl od imperativních jazyků je pouze uvedeno, co se má udělat, ale už není popsáno jakým způsobem, díky čemuž nástroj může provádět

nad dotazem optimalizace. Hlavní rozdíl oproti jazyku XPath, který je zde použit pro navigaci v XML dokumentech, je možnost vytvářet nové elementy. S tím je spojeno zavedení proměnných a řídících struktur do tohoto jazyka.

3.2.1 Struktura dotazu

Dotaz se skládá z prologu a těla. Prolog obsahuje úvodní deklarace jako import schémat, jmenných prostorů a definice funkcí. V této kapitole se jím nebudeme podrobněji zabývat. Tělo dotazu popisuje jeho výstup, má pevnou strukturu, kterou je možné syntakticky popsat pomocí výrazu *FLWR*, což je zkratka vytvořená ze slov *FOR*, *LET*, *WHERE* a *RETURN*. Syntaxe má tvar *(FOR_exp |LET_exp)? ... WHERE_exp? ORDER_BY? RETURN_exp* a význam jednotlivých částí je následující.

- FOR_exp slouží k iteraci nad nějakou posloupností. Většinou se iteruje nad výsledkem výrazu XPath, ale je možné definovat i vlastní číselnou posloupnost. Prvky posloupnosti se poté postupně váží na danou proměnnou. Má tvar *FOR var IN exp [, var IN exp]*. Je tedy možné iterovat přes více proměnných.
- LET_exp naváže na proměnnou nějaký výraz, většinou XPath. Má tvar *LET var := exp [, var := exp]*.
- WHERE_exp obsahuje podmínku, pomocí které je možné například omezit množinu výstupních elementů. Také lze použít kvantifikátory ve formě *SOME |EVERY var IN exp SATISFIES pred*. Tato složka není povinná.
- ORDER BY je nepovinná složka a slouží pro uspořádání výstupu, je možné použít volby *ascending* a *descending*.
- RETURN_exp se pak volá pro každou sadu uzlů daných složkami FOR, LET, WHERE.

Počet složek LET a FOR není omezen a FLWR výrazy lze také řetězit. Jazyk je case-sensitive, ale u klíčových slov je možné psát jak velká, tak malá písmena. Syntaxe pro vytvoření číselných posloupností má tvar *start_number to end_number*. Pro začátek respektive konec komentáře se používá posloupnost znaků *{--* respektive *--}*. Proměnné začínají dolarem. Řídící konstrukce pro selekci má tvar *if pred then exp1 else exp2*. V tabulce 3.3 jsou uvedeny funkce, které se často používají ve výrazech.

Funkce	**Popis**
document()	Vrací kořen daného dokumentu.
distinct()	odstranění stejných prvků z posloupnosti
count()	počet prvků posloupnosti
sum()	součet prvků posloupnosti
avs()	průměrná hodnota prvků posloupnosti

Tabulka 3.3: Často používané funkce v XQuery.

3.2.2 Vytváření elementů

Jak již bylo uvedeno v úvodu této podkapitoly, hlavním přínosem XQuery oproti samotnému XPath je možnost vytváření nových elementů. Struktura nového elementu (postupně vzniká les) je popsána v části RETURN_exp, pro lepší pochopení tohoto jazyka zde budou uvedeny ukázky kódu na příkladech dokumentů 3.12 a 3.13.

```
<items>
  <item status="prodáno">
    <itemno>1234</itemno>
    <seller>Jakoubek ze Stříbra</seller>
    <description>Středověké záchodové prkýnko</description>
    <reserve-price>12 345.60</reserve-price>
    <end-date>1. 1. 2003</end-date>
  </item>
  <item>...</item>
</items>
```

Příklad 3.12: XML dokument items.xml.

```
<bids>
  <bid>
    <itemno>1234</itemno>
    <bidder>John Smith</bidder>
    <bid-amount>9876.50</bid-amount>
    <bid-date>3. 1. 2003</bid-date>
  </bid>
  <bid>...</bid>
</bids>
```

Příklad 3.13: XML dokument bids.xml.

Pro vyhodnocení výrazů v části *return* je nutné uvést každý výraz ve složených závorkách, jak je to ukázáno v příkladu zdrojového kódu 3.14. Za použití výrazu XPath se zde iteruje přes jednotlivé položky z dražby v XML dokumentu 3.12. V části uvozené klíčovým slovem *let* jsou získány uzly nesoucí informace o příhozech na danou položku. Pomocí klíčového slova *where* omezujeme výstup pouze na položky s více jak deseti příhozy. Poslední částí je část uvozená klíčovým slovem *return*. Zde je popsáno, že výsledkem dotazu má být sekvence elementů *popular-item*, v našem případě je obsahem každého elementu sekvence tří elementů. Pro vytvoření sekvence elementů stačí zapsat elementy ve složených závorkách za sebe, jak je to ukázáno v příkladu. První dva elementy se zkopírují z dokumentu 3.12, poslední se spočítá a výraz udávající počet příhozů vyhodnotíme tak, že ho uvedeme ve složených závorkách. Získáme tedy položky v dražbě, které mají více než deset příhozů, a bude u nich uvedeno číslo dražené položky, popis a počet příhozů.

```
for $i in document("items.xml")/*/item
let $b := document("bids.xml")/*/bid[itemno = $i/itemno]
where count ($b) > 10
return
<popular-item> {
$i/itemno
```

```
$i/description
<bidCount> {count ($b)} </bidCount>
} </popular-item>
```

Příklad 3.14: Vyhodnocení výrazů.

Příklad zdrojového kódu 3.15 vrací nejvyšší aktuální příhozy na položky v dražbě, které doposud nebyly prodány. Výstup je seřazen podle čísla položky.

```
let $bids := document("bids.xml")/*/bid
for $i in document("items.xml")/*/item/itemno
where $i/../@status != "prodáno"
order by $i
return
<highbid>
<itemno> {$i} </itemno>
<bid-amount>
{ max($bids[itemno=$i]/bid-amount) }
</bid-amount>
</highbid>
```

Příklad 3.15: Vyhodnocení výrazů.

Kapitola 4

XML v relačních databázích

Tato kapitola popisuje relační databázové softwarové produkty pracující s typy XML dokumentů uvedenými v kapitole 2.1. Obecně je zde popsán princip, jakým databáze s XML dokumenty pracují, a způsob, jakým umožňují spravovat XML dokumenty. Uvedené databázové produkty poskytují podporu buď přímo na úrovni samotné databáze nebo mohou fungovat na vyšší úrovni jako nadstavba nad databází. Bylo zde čerpáno ze zdrojů [15], [1], [6], [10] a [12].

4.1 XML-enabled

Takto označené databázové systémy jsou schopny z databáze vygenerovat odpovídající XML dokument a naopak z XML dokumentu importovat data do databáze. Tedy podpora je cílená pro datově zaměřené aplikace.

4.2 Ukládání XML dokumentů

Tato část obecně popisuje různé způsoby uložení XML dat v tabulkách databáze.

4.2.1 Uložení do relační tabulky

Tuto možnost lze použít pro datově orientované dokumenty. Před samotným uložením je nutné XML dokument vhodně rozdělit, protože struktura tabulek je plochá, zatímco XML dokument lze reprezentovat pomocí stromové struktury (DOM). Jinak řečeno struktura XML dokumentu umožňuje zanořování. Po uložení pak relační tabulky odpovídají struktuře XML dokumentu.

Celý proces si ukážeme na příkladu XML dokumentu 4.1, který bude použit i v následujících kapitolách. Dokument popisuje jednoduchou databázi jedné společnosti skládající se z několika oddělení. V každém oddělení pracují zaměstnanci, u kterých sledujeme různé statistiky. Například zde je zachycen počet odpracovaných hodin. Zaměstnanci také mohou pracovat spolu v týmu na nějakém projektu. To je v dokumentu vyjádřeno tak, že se sdruží pod společným elementem *team*.

```
<?xml version="1.0" encoding="iso-8859-2"?>
<company>
  <department>
    <name>development</name>
```

```
    <team>
      <employee>
        <name><last>Lewis</last></name>
        <stats><hours>15</hours></stats>
      </employee>
      <employee>
        <name>Scott</name>
        <stats><hours>22</hours></stats>
      </employee>
    </team>
  </department>
  <department>
    <name>testing</name>
    <employee>
      <name>White</name>
      <stats><hours>11</hours></stats>
    </employee>
    <employee>
      <name>Lee</name>
      <stats><hours>27</hours></stats>
    </employee>
  </department>
  <department>
    <name>support</name>
  </department>
</company>
```

Příklad 4.1: XML dokument před uložení do databáze.

Dokument bude uložen do dvou tabulek s názvy *Department* a *Employee*. Roli cizího klíče zde bude hrát identifikátor oddělení. Výsledné tabulky relační databáze jsou uvedeny jako tabulky 4.1 a 4.2.

id	**name**
1	development
2	testing
3	support

Tabulka 4.1: Tabulka Department

id	**name**	**hours**	**team**	**department**
10	Lewis	15	5	1
14	Scott	22	5	1
16	White	11	NULL	2
24	Lee	27	NULL	2

Tabulka 4.2: Tabulka Employee

Z příkladu je patrné, že rozdělení XML dokumentu a jeho uložení do relační databáze je obtížně automatizované a neobejde se bez zásahu uživatele.

4.2.2 Uložení do CLOB

Jedná se o uložení do rozsáhlého textového dokumentu, kdy hodnotou typu CLOB (Character Large Object) je celý dokument. Po uložení XML dokumentu není explicitně uložena struktura dokumentu, proto jsou některé operace nad dokumentem časově náročnější. Tento způsob není z hlediska efektivity obecně nejlepší. Výhodou je rychlé získání celého dokumentu, nevýhodou je velká náročnost operací modifikujících část obsahu dokumentu. Navíc tyto operace také zvyšují paměťové nároky. Pro práci s takovým dokumentem se používá pak XPath a XQuery.

4.2.3 Objektově-relační uložení

Využívá se objektově-relačních typů. Podobně jako se v objektově-relační databázi hodnoty objektů ukládají do jednoho sloupce nebo řádku tabulky, tak se vytváří v tabulce sloupcové a řádkové XML dokumenty. Tento přístup je možné považovat za kompromis mezi uložením XML dokumentu do CLOB a jeho rozdělením na atomické hodnoty a uložením v tabulkách relační databáze. Výhodou tohoto uložení oproti CLOB je, že lze použít integritní omezení, která se běžně používají v relačních databázích.

Mapování XML dokumentů do relačních dat je provedeno pomocí XML schématu.

4.2.4 Uložení jako BLOB

Dokument je uložen ve sloupci tabulky binárně jako BLOB (Binary Large Object).

4.3 Standard SQL

Pro jazyk SQL existuje standard popisující dotazování nad XML dokumenty. Vzhledem k tomu, že se toto rozšíření objevilo v SQL standardu až v roce 2003 a řada komerčních produktů poskytovala podporu pro práci s XML už před tím, není v některých případech kompatibilita se standardem stoprocentní.

Standard zavádí nový datový typ pro XML. Hodnotou tohoto typu je přímo XML dokument.

Dále definuje funkce umožňující ze sloupců a řádků tabulek databáze vygenerovat XML dokument. Mezi základní funkce patří následujících šest.

- xmlelement() pro vytvoření jednoho elementu
- xmlattributes() pro vytváření atributů
- xmlroot() pro vytvoření kořenového prvku
- xmlforest() jako obdoba xmlelement(), nevytváří ale jeden element, nýbrž skupinu elementů
- xmlconcat() pro sloučení více XML hodnot
- xmlagg() pro vytvoření lesu elementů ze seznamu elementů

Standard také definuje pravidla pro mapování SQL a XML datových typů a mapování SQL a XML metadat. Díky tomu je možné pracovat s XML dokumenty prostřednictvím SQL příkazů.

4.4 Oracle databáze 11g

Stejně jako standard SQL i Oracle zavedl typ pro XML dokumenty pod názvem XMLType, pro který definuje funkce a operátory. Umožňuje také v databázi zaregistrovat XML schéma a vůči němu XML dokumenty validovat. Dotazování pomocí XPath a XQuery probíhá přímo na úrovni databázového serveru. XML a SQL jsou duální, jinak řečeno lze používat SQL operace nad XML daty a naopak. Pro uložení XML dokumentů lze použít způsoby popsány v části 4.2.

Přímo podporovány jsou i dokumentově orientované dokumenty, čemuž je přizpůsobeno uložení dat. Dokumenty jsou hierarchicky ukládány v části databáze zvané XML DB Repository. V nich jsou data uložena v podobě XML typovaných tabulek a pohledů. V tabulce je pak možné reprezentovat XML dokument pomocí jednoho sloupce nebo jako celý řádek. Nad těmito daty lze vytvářet indexy, konkrétně speciální index pro indexování XML, index v podobě B-Tree, indexy založené na funkci a speciální indexy na podporu fulltextového vyhledávání. XML dokumenty uložené v této části lze organizovat hierarchicky, v zásadě je možné se na ně dívat, jako kdyby byly uloženy v adresářích souborového systému.

Nad typem XMLType jsou v Oracle implementovány funkce uvedené v části 4.3. Na příkladu 4.2 je ukázáno použití funkcí xmlelement a xmlagg. Funkce xmlelement vytvoří jeden element. Prvním parametrem této funkce je název elementu a druhým jeho obsah. Funkce xmlagg vytvoří les XML elementů ze zadaného seznamu elementů. Dotaz v příkladu vrátí element s názvem Department. Obsahem tohoto elementu jsou elementy představující zaměstnance z oddělení s identifikátorem 30. V elementech zaměstnanců je uveden identifikátor zaměstnání a příjmení, podle něhož jsou elementy i řazeny.

```
SELECT XMLELEMENT("Department",
  XMLAGG(XMLELEMENT("Employee",
  e.job_id||' '||e.last_name)
  ORDER BY last_name))
  as "Dept_list"
  FROM employees e
  WHERE e.department_id = 30;

Dept_list
-------------------------------------------------------------
<Department>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Himuro</Employee>
  <Employee>PU_CLERK Khoo</Employee>
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Tobias</Employee>
</Department>
```

Příklad 4.2: Ukázka funkcí xmlelement() a xmlagg().

Na příkladu 4.3 je demonstrováno použití funkce xmlattributes, která vytváří atributy. Jejími parametry je seznam atributů. Jako název atributu se použije název sloupce uvedeného v parametru, případně lze jeho název specifikovat tak, jak je to ukázané v příkladu. Uvedený dotaz vrací element s názvem Emp, představující zaměstnance s identifikátorem 206. Atributy tohoto elementu jsou identifikátor zaměstnance a příjmení. Obsahem jsou elementy s názvy Dept respektive Salary reprezentující identifikátor oddělení respektive plat daného zaměstnance.

```
SELECT XMLELEMENT("Emp",
  XMLATTRIBUTES(e.employee_id AS "ID", e.last_name),
  XMLELEMENT("Dept", e.department_id),
  XMLELEMENT("Salary", e.salary)) AS "Emp Element"
  FROM employees e
  WHERE e.employee_id = 206;

Emp Element
-------------------------------------------------------------
<Emp ID="206" LAST_NAME="Gietz">
  <Dept>110</Dept>
  <Salary>8300</Salary>
</Emp>
```

Příklad 4.3: Ukázka funkce xmlattributes().

V příkladu 4.4 je ukázáno použití funkce xmlconcat pro konkatenaci XML hodnot. Jako parametry jsou funkci v ukázce předány dva elementy zastupující křestní jméno a příjmení zaměstnance. Dotaz pak vrací zaměstnance, jejichž identifikátor je větší jak 202, kde každý zaměstnanec je reprezentován dvojicí elementů obsahujících jeho křestní jméno a příjmení.

```
SELECT XMLCONCAT(XMLELEMENT("FName", e.first_name),
  XMLELEMENT("LName", e.last_name)) AS "Result"
  FROM employees e
  WHERE e.employee_id > 202;

Result
--------------------------------------------------------------
<FName>Susan</FName>
<LName>Mavris</LName>

<FName>Hermann</FName>
<LName>Baer</LName>

<FName>Shelley</FName>
<LName>Higgins</LName>

<FName>William</FName>
<LName>Gietz</LName>
```

Příklad 4.4: Ukázka funkce xmlconcat.

Kapitola 5

XML databáze

Tato kapitola se zmiňuje krátce o nativních XML databázích a popisuje, jak by bylo možné implementovat jednoduchou XML databázi. Bylo zde čerpáno ze zdrojů [10] a [12].

5.1 Nativní XML databáze

Tyto databázové systémy jsou šité na míru pro práci s XML dokumenty. Samotný datový model a uložení dat v databázi je orientováno na model XML dokumentu. Jako základní model se používá zpravidla objektový model dokumentu (DOM). Díky použití speciálního úložiště mohou nativní XML databáze dosahovat vyšší rychlosti při práci se složitějšími XML dokumenty, zejména v případě jedná-li se o dokumentově orientované XML dokumenty. Databáze tohoto typu poskytují zpravidla podporu jak pro datově zaměřené, tak pro dokumentově zaměřené aplikace. Příkladem takového databázového systému je databáze *eXist*.

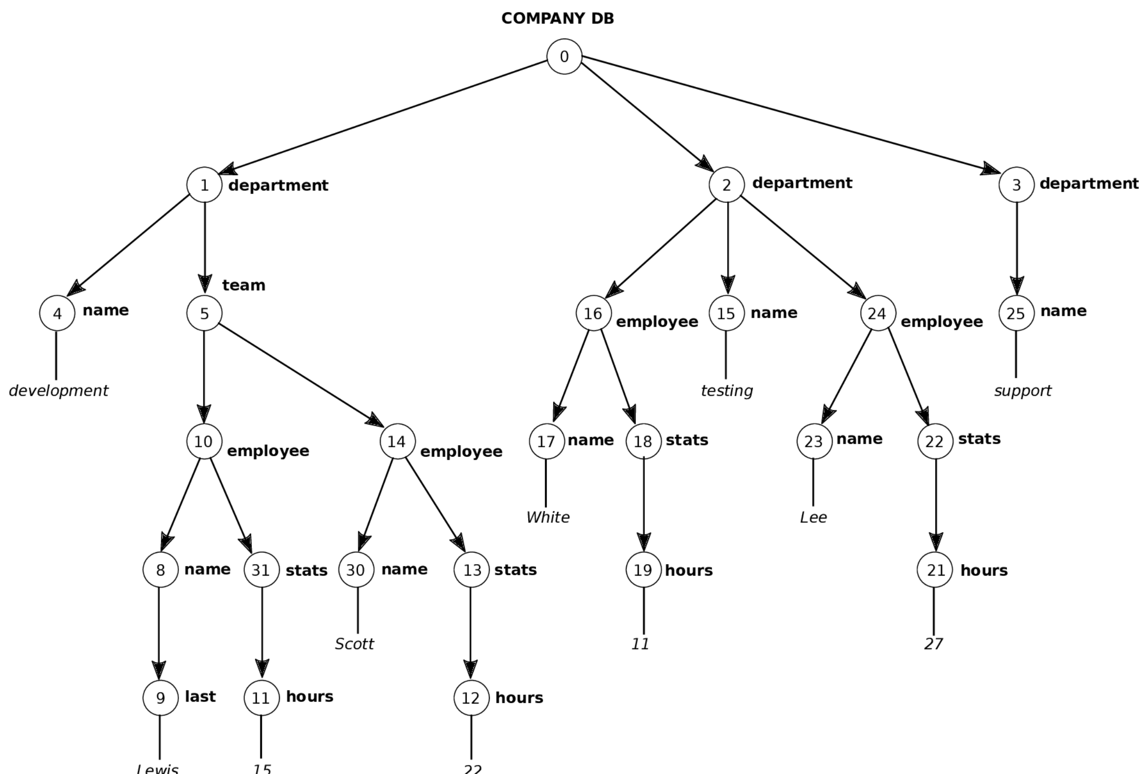
5.2 Efektivní uložení XML dokumentu

Jakým způsobem může být sofistikovaně uložen dokument v databázi si ukážeme na příkladu dokumentu 4.1. Tento způsob uložení předpokládá, že jako dotazovací jazyk je použit *XPath*. Pro dokument nejprve definujeme abstraktní model, poté se podíváme na jazyk *XPath* a spolu s abstraktním modelem navrhneme pro tento jazyk takový způsob uložení dokumentu, aby bylo vyhodnocení dotazů co nejefektivnější.

5.2.1 Abstraktní model

Abstraktní model XML dokumentu je orientovaný graf s následujícími vlastnostmi.

- Existuje zde kořenový uzel, z kterého je dostupný každý uzel v dokumentu, a který nemá žádné příchozí hrany.
- Jsou zde různé typy uzlů: elementový, atributový a hodnotový.
- Uzly elementové a atributové jsou označeny jednoznačným číselným identifikátorem a názvem elementu respektive atributu.



Obrázek 5.1: Abstraktní model XML dokumentu.

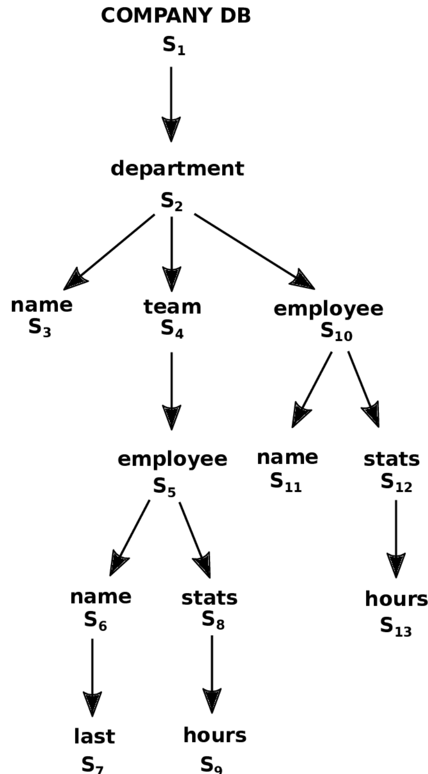
- Atributový uzel kromě názvu atributu ještě může být označen jako *ID*, nebo *REF*. Tyto označení jsou zde z toho důvodu, že XML nabízí možnost, aby element prostřednictvím atributu *REF* odkazoval na jiný element. Z tohoto důvodu rozlišujeme dva typy hran.

  1. *Referenční hrany*, jenž spojují atributový uzel typu *REF* s elementovým uzlem.
  2. *Obsahové hrany*, jejichž sémantika vyjadřuje zanoření jednoho prvku uvnitř druhého. Může se jednat o zanoření elementů, příslušnost atributu k elementu, hodnoty k elementu anebo hodnoty k atributu.

Pro názornost je uveden na obrázku 5.1 abstraktní model pro příklad dokumentu 4.1. Z obrázku je patrné, že pokud dokument neobsahuje žádné referenční hrany, jedná se vždy o strom. Za pozornost taktéž stojí, že model umožňuje, aby bylo k jednomu elementovému uzlu přiřazeno více hodnotových uzlů. Na první pohled taktéž není patrný rozdíl mezi elementovým a atributovým uzlem a jsou v tomto modelu zaměnitelné.

5.2.2 Implementace modelu

Jazyk XPath je podrobně popsán v části 3.1. Jedná se o výraz sestávající se ze sekvence kroků oddělených lomítky. Výraz je vyhodnocen zleva doprava a výstupem každého kroku je množina uzlů, která zároveň slouží jako vstup pro krok další nebo jako konečný výsledek. Pro efektivní vyhodnocení XPath dotazu je tedy nutné dokázat efektivně vyhodnotit krok. K tomu je zapotřebí zajistit, aby data pro vyhodnocení jednoho kroku byla uložena společně, ideálně aby k jejich načtení byl potřeba jediný přístup na disk. Nabízí zde poměrně



Obrázek 5.2: Rozdělení uzlů do tříd relace ekvivalence.

jednoduché řešení. Nad abstraktním modelem definujeme relaci ekvivalence, kde do stejné třídy budou zařazeny uzly, které mají stejnou sekvenci značek (název tagu či atributu) od kořene. Každá třída poté odpovídá jedné tabulce, ta bude obsahovat identifikátor daného uzlu, identifikátor rodiče uzlu a sloupec pro hodnotový uzel. Hodnota nemusí být přiřazena. K vyhodnocení jednoho kroku je poté potřeba obvykle jedné tabulky, v případě dvou lomítek se jedná o tabulky pro elementy či atributy s určitým názvem.

Pro názornost je zde několik jednoduchých příkladů. Nejprve rozdělíme XML dokument do tříd. Rozdělení je zobrazeno na obrázku 5.2. Každá třída S_1 až S_{13} odpovídá jedné tabulce. Hrana mezi dvěma třídami značí vztah rodič-potomek mezi uzly v tabulkách.

V příkladu 5.1 je uveden výraz složen ze dvou kroků. Pro jeho vyhodnocení nejprve uložíme do prázdného seznamu všechny uzly z tabulky S_2.

```
//department/name
```

Příklad 5.1: Ukázka vyhodnocení kroku s jedinou tabulkou.

Tím jsme zpracovali první krok. V druhém kroku vybereme z tabulky S_3 všechny uzly, jejichž rodičovský uzel je v seznamu. Ty jsou poté výsledkem XPath výrazu. Na každý krok nám tedy stačilo zpracování řádků v jediné tabulce.

V příkladu 5.2 je uveden další výraz se dvěma lomítky. Na rozdíl od toho předešlého zde existují dvě třídy s elementovými uzly s tagem *employee*.

```
//employee
```

Příklad 5.2: Ukázka vyhodnocení kroku s více tabulkami.

Tedy výsledkem XPath výrazu budou uzly z tabulek S_5 a S_{10}. Výsledné uzly z XPath dotazů pak lze doplnit o uzly ze tříd potomků a vypsat je jako části XML dokumentu.

Kapitola 6

Temporální relační databáze

Podobně jako nad relačními databázemi vznikla podpora pro práci s XML dokumenty, byly vytvořeny i mechanismy umožňující, aby v databázi kromě aktuálních dat byla uložena i jejich historie. Tato podpora je ovšem poskytována na vyšší úrovni, buď na aplikační anebo ve formě nadstavby, kterou zprostředkovává nějaká knihovna. Dalším rozdílem je, že se často jedná o produkty třetích stran.

Tyto databáze se například využívají v informačních systémech bank, kde chceme znát kromě aktuálního stavu účtu i jeho historii nebo chceme zaznamenávat vývoj kurzu měny, v systémech pro evidenci skladu, kde nás zajímají mimo aktuálního stavu položek i stavy předchozí a obdobně v systémech katastru, kde je potřebné znát i předchozí vlastníky pozemků.

Hlavní problém temporálních databází vychází z jejich podstaty. Tím, že při každé změně hodnoty jsou data v relační databázi až na změněnou hodnotu duplikována, je celkový objem uložených dat oproti běžným relačním databázím mnohem vyšší. Proto se jednou za čas musí provést smazání některých dat, zpravidla se jedná o nejstarší data, a přicházíme tak o část jejich historie.

V této kapitole budou popsány problémy temporálních databází, které využívají relačních databází, a jejich možná řešení. Také zde budou popsány dostupné dotazovací jazyky a nastíněn způsob jejich implementace. V souvislosti s implementací je zde také popsáno rozšíření databázových tabulek. Bylo zde čerpáno ze zdrojů [4], [13], [14] a [5].

6.1 Reprezentace času

Podle účelu můžeme pro vyjádření času použít *bod* (časový okamžik, v relační databázi například typ Datetime), *interval* (doba mezi dvěma okamžiky, v relační databázi typ Interval) nebo *periodu* (počáteční a koncový okamžik). Obecně reprezentace libovolné číselné hodnoty má na počítači jistá omezení. Číslo s plovoucí desetinou čárkou lze uložit pouze s určitou přesností, ta je dána teoreticky omezenou pamětí počítače, prakticky se jedná o velikost paměti 64 bitů. Navíc tuto přesnost dále omezují operace prováděné s těmito hodnotami, to je důsledkem způsobu, jakým je číslo v paměti uloženo a konkrétněji je to popsáno ve standardu IEEE 754. Prakticky proto se ve všech třech případech potýkáme s problémy.

Pro reprezentaci časových údajů tedy nelze použít reálná čísla, ale pouze racionální. Navíc nedokážeme reprezentovat celou množinu racionálních čísel, ale pouze její malou konečnou podmnožinu. Důsledkem toho je, že zde existuje nejmenší časové kvantum vyja-

dřující s jakou přesností čas ukládáme. Pokud se jedná o racionální číslo, mluvíme o tzv. Hustém časovém modelu, v případě celého čísla o tzv. Diskrétním časovém modelu. Mezi libovolnými dvěma časy pak existuje konečný počet časových hodnot.

6.2 Datový model

K tomu, aby bylo možné v databázi uchovávat historii dat, je nutné rozšířit záznamy v tabulce obecně o časovou značku. Tedy kromě dat je uložena i jejich platnost. Pro zachování referenční integrity je pak tato časová značka v záznamu součástí primárního klíče a všech cizích klíčů. Díky tomu je primární klíč unikátní a cizí klíče odkazují na odpovídající záznamy (jejich existenci je nutné ošetřit). V relačních databázích existují již od standardu SQL-92 operace a datové typy DATE, TIME, TIMESTAMP a INTERVAL poskytující podporu pro práci s časovými údaji. Ty je možné použít, další možností je vytvořit si vlastní datový typ. To je často preferovaná volba, protože čas je běžně reprezentován celočíselnou hodnotou uvádějící počet sekund od určitého data. Tímto datem jsme poté omezeni, neboť nedokážeme zakódovat časy platnosti, které mu chronologicky předcházejí.

Časová značka může mít dvojí sémantiku. Buď může reprezentovat transakční čas, tedy čas vzniku daného záznamu v databázi, anebo čas platnosti, ten popisuje, kdy jsou data platná v modelovaném světě. Je možné ukládat si oba časy, v některých případech nás může zajímat pouze transakční čas. Podstatné je uvědomit si, že tyto časy mohou být stejné, ale také se můžou lišit například poté, co provedeme obnovu zálohy databáze. Dalším rozdílem je, že čas platnosti může uživatel libovolně měnit, ale čas transakce je neměnný. Pokud se v tabulce nachází jak čas platnosti, tak i transakční čas, mluvíme o tzv. bitemporální tabulce.

6.3 Restrukturalizace a shlukování

Po dotazu SELECT je vhodné ještě sloučit záznamy lišící se pouze v časové značce. Výsledkem je potom jediný záznam s upravenou časovou značkou, ta je sjednocením původních časových značek. Tomuto procesu se říká restrukturalizace.

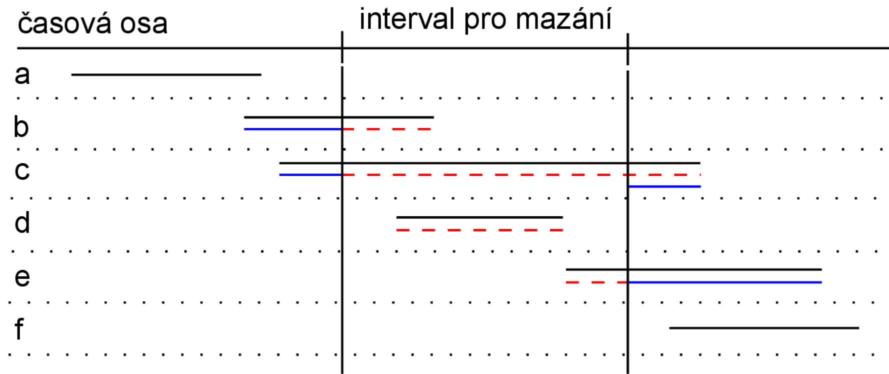
Shlukování v kontextu temporálních databází je proces, při němž se slučují záznamy lišící se pouze časovou značkou. To, zda je možné záznamy sloučit, je závislé na reprezentaci a sémantice časových značek.

6.4 Mazání

Mazání záznamů v temporálních databázích má jiný charakter než mazání v klasické netemporální databázi. Pokud tabulka obsahuje časovou značku, která popisuje časový úsek záznamu, může nastat situace, kdy po příkazu *DELETE* přibude v databázi nový záznam. Protože je tato problematika netriviální, bude podrobněji popsána. Nejprve bude popsána tabulka pouze s časem transakce, pak pouze s časem platnosti a nakonec kombinace předchozích dvou možností. V případě, že tabulka neobsahuje transakční čas, ani čas platnosti, probíhá mazání obvyklým způsobem.

6.4.1 S transakčním časem

U aktuálního záznamu je nastaven koncový čas na aktuální čas.



Obrázek 6.1: Mazání temporálních dat.

6.4.2 S časem platnosti

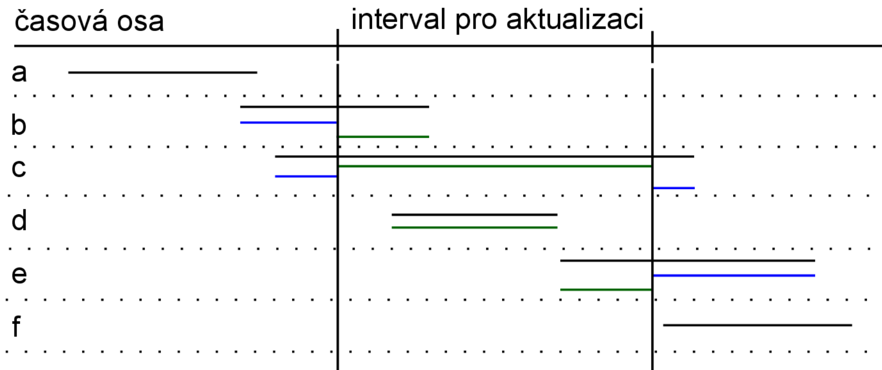
Proces bude demonstrován na obrázku 6.1. Ten popisuje různé situace, ke kterým může dojít. Pokud je průnik intervalu pro mazání a intervalu platnosti záznamu prázdný, tak jak je to v případě *a* a *f*, nedochází v databázi k žádné změně. Je-li interval platnosti záznamu podintervalem intervalu pro mazání, dojde k fyzickému smazání záznamu, jak je to ukázáno v případě *d*. V případě, že je podintervalem interval pro mazání a rozděluje interval platnosti na dva, na obrázku možnost *c*, je u fyzického záznamu aktualizována platnost a navíc přibude v databázi nový záznam se zbytkem platnosti. U variant *b* a *e* dochází ke změně platnosti fyzického záznamu. Pokud časový interval pro mazání není zadán, použije se interval od aktuálního času do nekonečna.

6.4.3 Kombinace času platnosti a transakčního času

Ukládá-li tabulka kromě času platnosti i transakční čas, je situace daleko složitější. Čas platnosti je na rozdíl od transakčního času spravován převážně uživatelem. Je to de facto uživatelem definovaná historie dat. Transakční čas je plně ve správě databáze a jeho úkolem je zachytit jakékoliv změny v databázi a to i změny týkající se změny času platnosti. V první řadě je tedy třeba si uvědomit, že při mazání bereme v úvahu pouze data s aktuálním transakčním časem a ostatní záznamy se ignorují. S vybranými záznamy s aktuálním transakčním časem pak pracujeme stejně, jak bylo popsáno v předchozí sekci s tím rozdílem, že nejprve uděláme kopii záznamu. S touto kopií pak provedeme kroky popsané v předchozí sekci a navíc změníme její počáteční čas transakce na aktuální čas. U původní verze následně změníme koncový transakční čas na aktuální čas. Tím jsme zajistili, že původní stav záznamu zůstane v databázi a bude dohledatelný, a zároveň v databázi přibyla nová verze záznamu s upravenou platností, u které je zaznamenán korektní čas transakce (kdy k úpravě došlo).

6.5 Aktualizace

Aktualizace v temporálních databázích je ještě o něco složitější než mazání. Některé knihovny jako například *TimeDB* tuto operaci nepodporují. Nejprve bude popsána tabulka pouze s časem transakce, pak pouze s časem platnosti a nakonec kombinace předchozích dvou možností. V případě, že tabulka neobsahuje transakční čas, ani čas platnosti, probíhá aktualizace obvyklým způsobem.



Obrázek 6.2: Aktualizace temporálních dat.

6.5.1 S transakčním časem

U záznamu s aktuálním transakčním časem je nastaven koncový transakční čas na aktuální čas. Dále se vytvoří kopie, ta se aktualizuje novými hodnotami, a je jí přiřazen transakční čas od aktuálního času do nekonečna.

6.5.2 S časem platnosti

Pro popis využijeme obrázek 6.2. Na tomto obrázku je ukázáno šest různých možností, ke kterým může dojít. Nejjednodušším případem je možnost *a* a *f*, kdy se interval platnosti záznamu míjí s intervalem pro aktualizaci, v takovém případě nedojde v databázi k žádné změně. Triviální je i případ *d*. Ten popisuje situaci, kdy interval platnosti záznamu je podintervalem intervalu pro aktualizaci. Platnost fyzického záznamu zůstane nezměněna a změní se pouze ostatní hodnoty, co se mají aktualizovat. O něco složitější jsou případy *b* a *e*. U fyzického záznamu se zmenší interval platnosti a navíc je vložen do databáze nový záznam s novými daty a zbytkem platnosti, o který byl zkrácen původní záznam. Případ *c* je nejsložitější. Interval pro aktualizaci rozděluje záznam na tři části. Konkrétně starý záznam se rozdělí na dva záznamy tak, že u původního záznamu se změní platnost a navíc se vloží do databáze další záznam se starými daty. Třetí část představuje nový záznam s novými daty a jeho interval platnosti se shoduje s intervalem pro aktualizaci. Pokud časový interval pro aktualizaci není zadán, použije se interval od aktuálního času do nekonečna.

6.5.3 Kombinace času platnosti a transakčního času

Proces aktualizace probíhá podobně jako u mazání. S vybranými záznamy s aktuálním transakčním časem pracujeme stejně, jak bylo popsáno v předchozí sekci s tím rozdílem, že nejprve uděláme kopii záznamu. S touto kopií pak provedeme kroky popsané v předchozí sekci a navíc změníme její počáteční čas transakce na aktuální čas. U původní verze následně změníme koncový transakční čas na aktuální čas. Tím jsme zajistili, že původní stav záznamu zůstane v databázi a bude dohledatelný, a zároveň v databázi přibyla nová verze záznamu s upravenou platností, u které je zaznamenán korektní čas transakce (kdy k úpravě došlo).

6.6 Vkládání

Jedná se o poměrně jednoduchou operaci. Pokud není uveden čas platnosti a tabulka jej ukládá, je použit aktuální čas, konkrétně v případě intervalu aktuální čas až nekonečno. Ukládá-li tabulka transakční čas, je na jeho místě uložen interval od aktuálního času do nekonečna.

6.7 TSQL2

Je jazyk pro temporální dotazování rozšiřující SQL-92. Používá mimo jiné i režim Snapshot, ten ignoruje historii záznamů v databázi a pracuje pouze s těmi záznamy, které jsou aktuálně platné.

6.7.1 Transakční čas a čas platnosti

Aby bylo možné reprezentovat bez omezení libovolný časový okamžik, tedy například i dávnou minulost, je zaveden nový datový typ PERIOD. Ten je určen počátečním a koncovým časem a přesností. Rozsah a přesnost času platnosti je možné definovat u každé tabulky, pro reprezentaci jsou zde pak dvě možnosti, buď je čas platnosti reprezentován u každého záznamu množinou disjunktní intervalů anebo množinou časových okamžiků. U transakčních časů není možné nastavit přesnost ani rozsah a jedná se vždy o jeden interval.

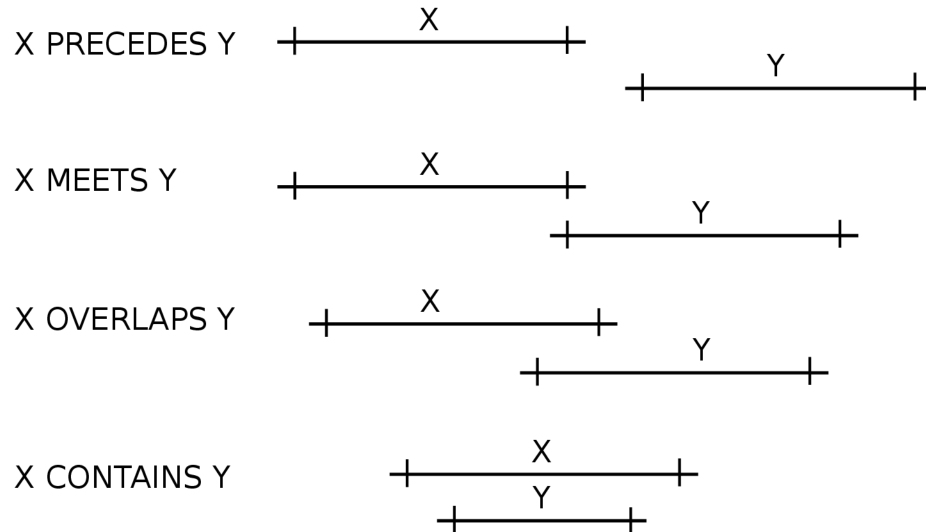
6.7.2 Druhy tabulek

Rozlišují se zde následující čtyři druhy tabulek. U dvou z nich existují ještě dvě různé jejich varianty. Celkem tedy můžeme mít šest různých typů tabulek. Pokud mezi dvěma tabulkami existuje vazba cizí klíč-primární klíč, je nutné aby byly obě tabulky stejného typu.

- *Snímková tabulka* (snapshot) je tabulka, která u záznamů neukládá čas platnosti, ani čas transakce.
- *Stavová tabulka* ukládá u záznamů pouze čas platnosti. Existují dva typy této tabulky lišící se reprezentací času platnosti. Ten může být vyjádřen okamžikem platnosti anebo intervalem platnosti.
- *Transakční tabulka* ukládá u záznamů pouze čas transakce.
- *Bitemporální tabulka* ukládá u záznamů čas transakce i čas platnosti. Opět zde existují dva typy podle způsobu reprezentace času platnosti.

6.7.3 Typ SURROGATE

Jedná se o nový datový typ používaný pro vytvoření unikátního identifikátoru v rámci jednoho sloupce tabulky. Hodnota je generována databází a jedinou povolenou operací nad tímto typem je porovnání.



Obrázek 6.3: Sémantika nových predikátů.

6.7.4 Popis jazyka

Při vytvoření tabulky se určí, zda má obsahovat transakční data (pomocí klauzule *AS TRANSACTION*) a zda má obsahovat čas platnosti (klauzule *AS VALID STATE* pro reprezentaci pomocí intervalu, *AS VALID EVENT* pro reprezentaci pomocí časového okamžiku). Klauzule *VACUUM časový údaj* pak slouží pro nastavení odsávání, pomocí klauzule *NOBIND()* ho lze nastavit relativně.

Příkaz *DROP* zůstává stejný jako v SQL-92, u příkazu *INSERT*, *DELETE* a *UPDATE* je navíc volitelná klauzule *VALID* pro určení platnosti.

SELECT je rozšířen o možnost vybírat z tabulky čas platnosti pomocí volby *VALID(tabulka)* a transakční čas pomocí volby *TRANSACTION(tabulka)*. Pokud požadujeme provedení restrukturalizace, je nutné uvést sloupce, které se mají slučovat. Ty se uvádí v klauzuli *FROM* za názvem tabulky ve formátu *nazev_tabulky(sloupec1, sloupec2, ...)*.

Nejvíce rozšíření najdeme v klauzuli *WHERE*. Je zde možnost používat klauzule *VALID(tabulka)* a *TRANSACTION(tabulka)* zastupující u záznamu čas platnosti respektive čas transakce. Pro specifikaci časového intervalu je možné použít konstrukci *PERIOD [od – do]*. Dále jsou zde nové predikáty, pomocí kterých lze tyto části skládat. Jejich sémantika je popsána na obrázku 6.3.

6.8 Shrnutí

Pokud se rozhodneme rozšířit existující databázi tak, aby umožňovala zaznamenávat kromě aktuálního stavu hodnot i jejich předchozí stavy, narazíme na následující problémy.

- rychlý nárůst dat v databázi
- zajištění referenční integrity
- kódování času
- záznamy je třeba shlukovat

- nad výsledky dotazu je třeba provést restrukturalizaci
- zajištění nepřekrývání časových intervalů platnosti
- operace mazání a aktualizace jsou netriviální

Kapitola 7

Temporální XML databáze

Pro temporální uložení XML dokumentu by teoreticky bylo možné použít čistě nativní XML databázi bez jakýchkoliv dalších rozšíření. Takové řešení je ovšem velice neefektivní a v této části navrhneme efektivnější řešení, které je rozšířením návrhu popsaného v části 5.2. K tomu budeme potřebovat dvě struktury. První pro vyhodnocování XPath dotazů a druhou pro vytváření snímků dokumentu v zadaném okamžiku. Kromě toho bude nutné rozšířit dotazovací jazyk XPath. Bylo zde čerpáno ze zdroje [12].

7.1 Abstraktní model

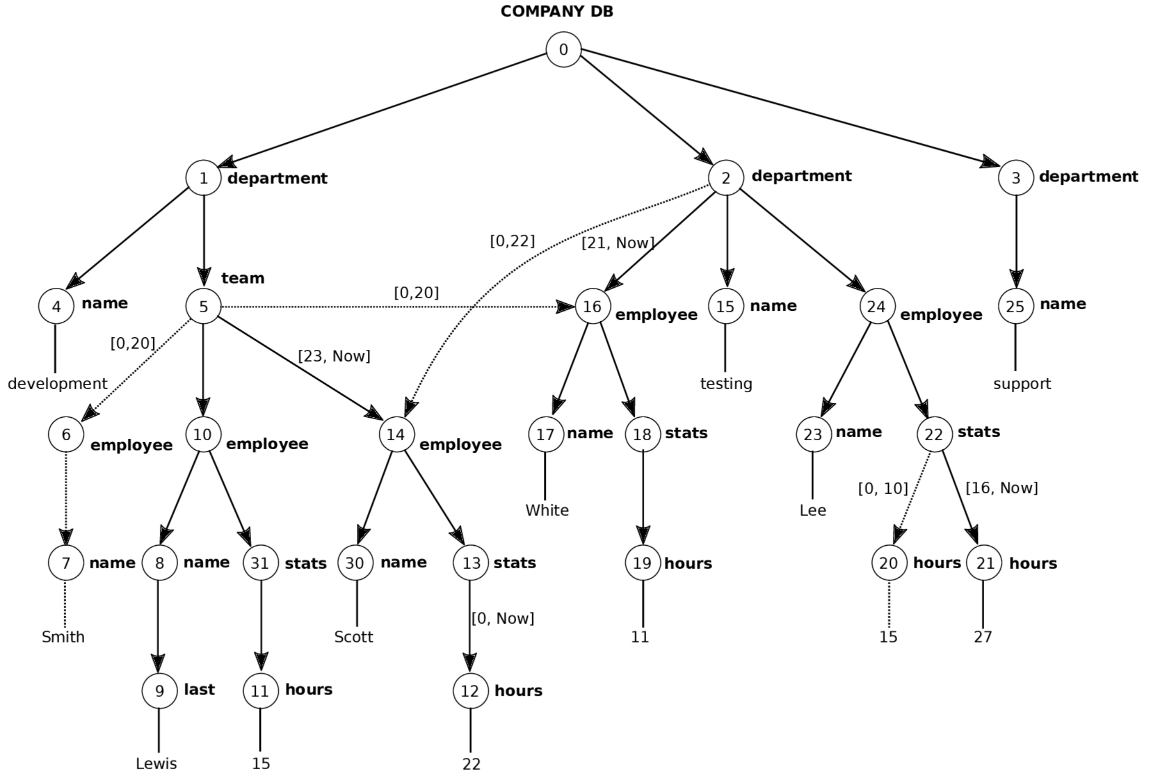
V sekci 5.2.1 byl definován abstraktní model pro XML databázi. Tento model použijeme a navíc rozšíříme následujícím způsobem.

- Každé hraně je přiřazen právě jeden interval (z důvodu zefektivnění implementace). Ten udává čas existence hrany mezi dvěma uzly a je označený pomocí uspořádané dvojice [a, b], která značí uzavřený interval od a do b, kde $a \leq b$.
- Obě hranice intervalu jsou celočíselné a čas je diskrétní.
- Aktuální čas je označen slovem *Now*.

Na obrázku 5.1 je uveden abstraktní model pro XML dokument, tento model obsahuje pouze aktuální části XML dokumentu. Na obrázku 7.1 je uveden abstraktní model pro temporální XML dokument, který kromě aktuálních hodnot z obrázku 5.1 zachycuje i historii dokumentu. Pro lepší čitelnost jsou neaktuální hrany označeny tečkovaně a u hran neomezených intervalem platnosti je tento interval vynechán.

7.2 Platnost uzlu

Platnost jednotlivých hran je dána intervaly, které jsou jim přiřazeny. Tyto intervaly říkají, kdy daný vztah v dokumentu existuje. Pomocí obsahových hran, které jsou příchozí do daného uzlu, jsme schopni určit čas, kdy se tento uzel bude v dokumentu vyskytovat. Toho dosáhneme tak, že provedeme sjednocení intervalů všech příchozích obsahových hran. Například na obrázku 7.1 je platnost uzlu *16* spočítána jako $[0, 20] \cup [21, Now] = [0, Now]$.



Obrázek 7.1: Abstraktní model temporálního XML dokumentu.

7.3 Souvislá cesta a maximální souvislá cesta

Rozšíříme pojem cesta o časovou dimenzi. Tomuto rozšíření budeme dále říkat souvislá cesta. Zjednodušeně řečeno jedná se o cestu, která je platná v abstraktním modelu v určitém intervalu T. Formálně jde o sekvenci $(n_1, \ldots, n_k, T)$, kde k je počet uzlů cesty, mezi uzly jsou obsahové hrany $e_1(n_1, n_2, T_1), e_2(n_2, n_3, T_2), \ldots, e_k(n_{k-1}, n_k, T_k)$ a interval platnosti cesty T je spočítán následujícím vztahem.

$$T = \bigcap_{1}^{k} T_i$$

Interval T pro maximální souvislou cestu z uzlu n_1 do uzlu n_k získáme jako sjednocení největší možné množiny navazujících intervalů T_i takových, že existuje souvislá cesta z n_1 do n_k s intervalem T_i. Protože intervaly musí být navazující, může existovat více maximálních souvislých cest. Díky podmínkám 1 a 2 ze sekce 7.4 stačí k výpočtu maximální souvislé cesty navštívit každý uzel pouze jednou. Navíc chceme-li spočítat maximální souvislou cestu z kořene r do uzlu n_2 a známe-li maximální souvislou cestu z r do uzlu n_1, který leží na této souvislé cestě, stačí nám spočítat maximální souvislou cestu z n_1 do uzlu n_2 a provést průnik. Jinak řečeno souvislá cesta je vždy dána intervalem platnosti poslední hrany.

Na obrázku 7.1 je například souvislá cesta (0, 1, 5, 14, 13, [23, Now]) a (0, 2, 14, 13, [0, 20]) a maximální souvislá cesta z kořene do uzlu 13 s intervalem [0, Now].

7.4 Podmínky konzistence

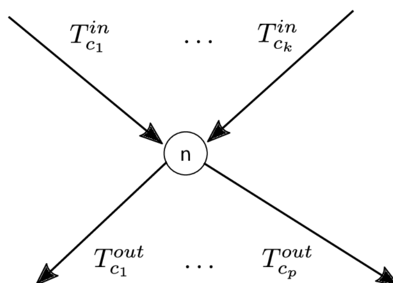
Jako součást abstraktního modelu jsou zavedeny, především z důvodu efektivity, první dvě podmínky konzistence, které umožňují v konečném důsledku rychlé vyhodnocování temporálních XPath dotazů. Třetí podmínka ošetřuje možnost změny času platnosti, kdy by mohl vzniknout v abstraktním modelu nežádoucí cyklus. Cílem dalších podmínek je zajistit, že pokud je platná referenční hrana, pak je fyzicky v dokumentu přítomen jak její počáteční, tak cílový uzel. Je nutné ošetřit, že každý dokument bude splňovat tyto podmínky před a po každé úpravě dokumentu.

7.4.1 První podmínka

První podmínka je popsána pomocí obrázku 7.2 a následujícího vztahu.

$$\bigcup_{1}^{p} T_{c_j}^{out} \subseteq \bigcup_{1}^{k} T_{c_i}^{in}$$

Jedná se o vztah mezi příchozími a odchozími obsahovými hranami uzlu, kde $T_{c_j}^{out}$ jsou intervaly platnosti odchozích hran, $T_{c_i}^{in}$ intervaly platnosti příchozích hran, p je počet odchozích hran a k počet příchozích hran. Zjednodušeně řešeno podmínka zajišťuje, že potomek uzlu je v dokumentu přítomen pouze tehdy, pokud je v dokumentu přítomen jeho rodič.



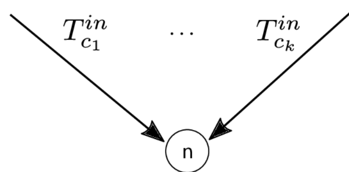
Obrázek 7.2: První podmínka konzistence temporální XML databáze.

7.4.2 Druhá podmínka

Druhá podmínka je definována pomocí obrázku 7.3, pro který platí následující vztah.

$$T_{c_i}^{in}.FROM = T_{c_{i-1}}^{in}.TO + 1$$

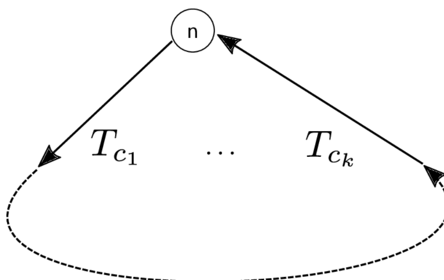
Podmínka popisuje vztah pouze mezi příchozími obsahovými hranami uzlu, kde $T_{c_i}^{in}$ jsou intervaly platnosti příchozích hran, $T_{c_i}^{in}.FROM$ jsou začátky intervalů, $T_{c_i}^{in}.TO$ jsou konce intervalů, k je počet hran a platí $2 \leq i \leq k$. Vztah říká, že příchozí hrany na sebe musí navazovat. Tedy nesmí se překrývat jejich intervaly platnosti a nesmí být přítomny mezery v intervalu platnosti uzlu. Podmínka nepřekrývajících se intervalů platnosti existuje i u temporálních relačních databází a v případě temporálních XML databází zajišťuje, že uzel nemůže mít v jeden okamžik dva rodičovské uzly, což by se v kontextu dokumentu interpretovalo několikanásobnou existencí stejného elementu či atributu.



Obrázek 7.3: Druhá podmínka konzistence temporální XML databáze.

7.4.3 Třetí podmínka

Třetí podmínka říká, že uděláme-li nad abstraktním modelem snímek v čase t, pak podgraf skládající se z obsahových hran a uzlů platných v tento okamžik t je stromem s původním kořenovým uzlem. Aby to bylo splněno, nesmí být v okamžiku t v grafu cyklus. Vezměme



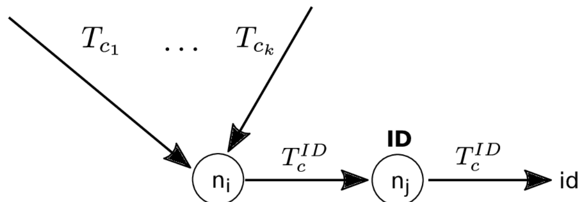
Obrázek 7.4: Třetí podmínka konzistence temporální XML databáze.

si původní abstraktní model před provedením snímku. Tato podmínka znamená, že pokud existuje v modelu cyklus ilustrovaný na obrázku 7.4, kde T_{c_i} jsou intervaly platnosti hran v cyklu a k je počet hran v cyklu, pak musí platit pro intervaly platnosti hran následující vztah, aby byl model konzistentní v rámci třetí podmínky.

$$\bigcap_{1}^{k} T_{c_i} = \emptyset$$

7.4.4 Čtvrtá podmínka

Zajištění konzistence pro referenční hrany je netriviální. Proto je nutné definovat jistá omezení. Prvním z nich je, že *ID* elementu je konstantní pro všechny snímky dokumentu. Druhé říká, že nesmí existovat v dokumentu dva elementy se stejnou hodnotou atributu *ID*. Podmínka je popsaná na obrázku 7.5. Na tomto obrázku je zobrazen atributový uzel n_j typu



Obrázek 7.5: Čtvrtá podmínka konzistence temporální XML databáze.

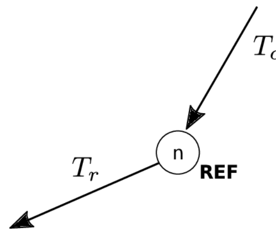
ID s hodnotou atributu *id*. Dále je zde elementový uzel n_i se vstupními hrami s intervaly

platnosti T_{c_i}. Všechny hrany jsou obsahové. Pro intervaly platnosti musí platit následující vztah.

$$T_c^{ID} = \bigcup_{1}^{k} T_{c_i}$$

7.4.5 Pátá podmínka

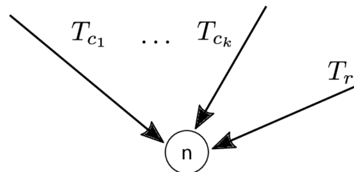
Tato podmínka zajišťuje existenci počátečního uzlu referenční hrany v dokumentu v případě, že je tato referenční hrana platná. Konkrétně se jedná o atributový uzel typu *REF*. Na obrázku 7.6 je zobrazen tento uzel spolu s obsahovou hranou s intervalem platnosti T_c a referenční hranou s intervalem platnosti T_r. Aby byla podmínka splněna, musí platit vztah $T_c = T_r$.



Obrázek 7.6: Pátá podmínka konzistence temporální XML databáze.

7.4.6 Šestá podmínka

Obdobně jako u páté podmínky je nutné zajistit v případě platnosti referenční hrany, že bude existovat v dokumentu cílový elementový uzel. To popisuje obrázek 7.7. Na obrázku



Obrázek 7.7: Šestá podmínka konzistence temporální XML databáze.

je elementový uzel n s příchozími obsahovými hranami s intervaly platnosti T_{c_i} a jednou příchozí referenční hranou s intervalem platnosti T_r. Pro splnění šesté podmínky musí platit následující vztah.

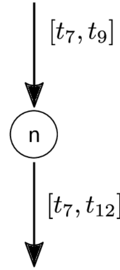
$$T_r \subseteq \bigcup_{1}^{k} T_{c_i}$$

7.5 Možné nekonzistence

V této sekci na základě sekce 7.4 identifikujeme možné typy nekonzistencí a analyzujeme, jak mohou být odstraněny, případně jak mohou vzniknout.

7.5.1 Nekonzistence typu I

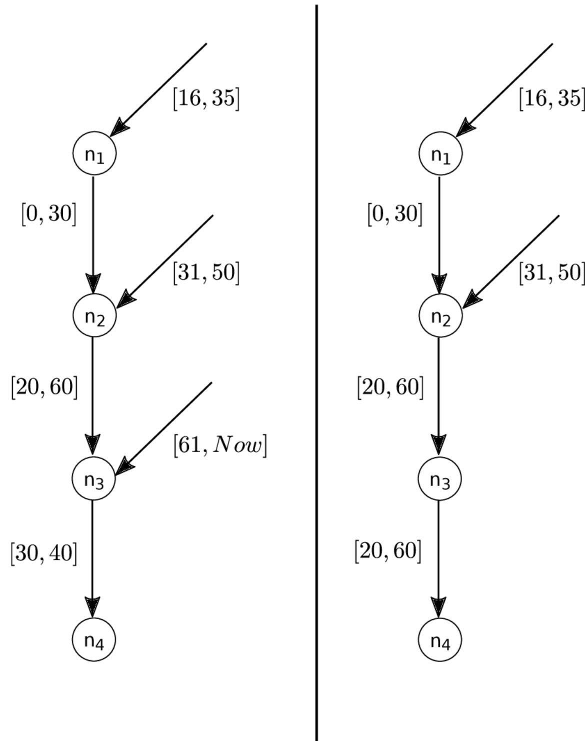
Nekonzistence nastává, existuje-li odchozí obsahová hrana, jejíž interval je mimo platnost uzlu. Příklad nekonzistence je zobrazen na obrázku 7.8. Problémovým intervalem je zde $[t_9 + 1, t_{12}]$, který tuto nekonzistenci vytváří.



Obrázek 7.8: Příklad nekonzistence typu I.

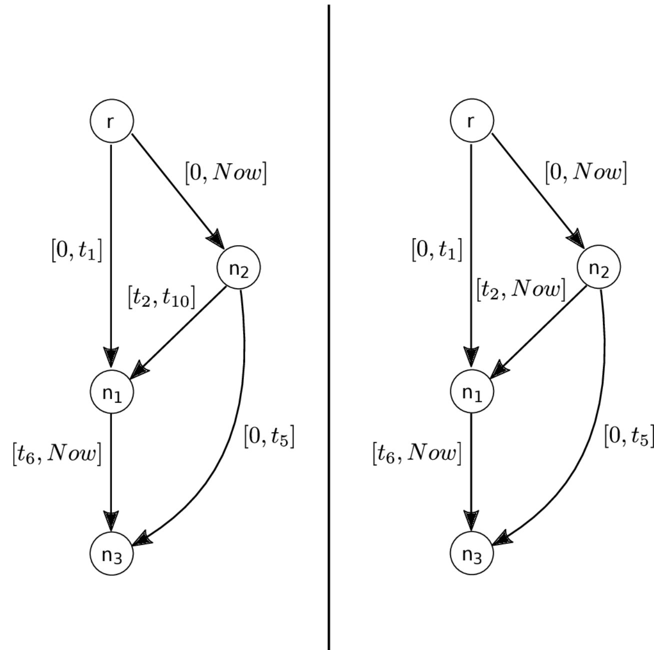
Detekce této nekonzistence je poměrně snadná, stačí ověřit, zda jsou všechny intervaly odchozích obsahových hran zahrnuty v platnosti uzlu.

Pro opravu této nekonzistence je nejjednodušší provést redukci intervalu obsahové odchozí hrany, které se nekonzistence týká. Tuto redukci může být nutné propagovat dále, protože může způsobit nekonzistenci typu I a II v koncovém uzlu. To je ukázáno na obrázku 7.9. Mezi uzly n_2 a n_3 je nekonzistence. Pokud na abstraktní model v levé části aplikujeme redukci, tedy nahradíme interval $[20, 60]$ za $[20, 50]$, zavedeme nekonzistenci typu II v uzlu n_3, protože v jeho platnosti vznikne mezera. Aplikujeme-li stejnou redukci na abstraktní model v pravé části, zavedeme nekonzistenci typu I u hrany mezi uzly n_3 a n_4.



Obrázek 7.9: Příklad problému opravy nekonzistence typu I redukcí intervalu.

Alternativou opravy může být rozšíření intervalu příchozí obsahové hrany, které může být nutné také provagovat dále ve směru ke kořeni, a navíc hrozí, že bude v dokumentu zavedena nekonzistence typu III. Tato možnost je ilustrována na obrázku 7.10. V levé části je



Obrázek 7.10: Příklad opravy nekonzistence typu I rozšířením intervalu.

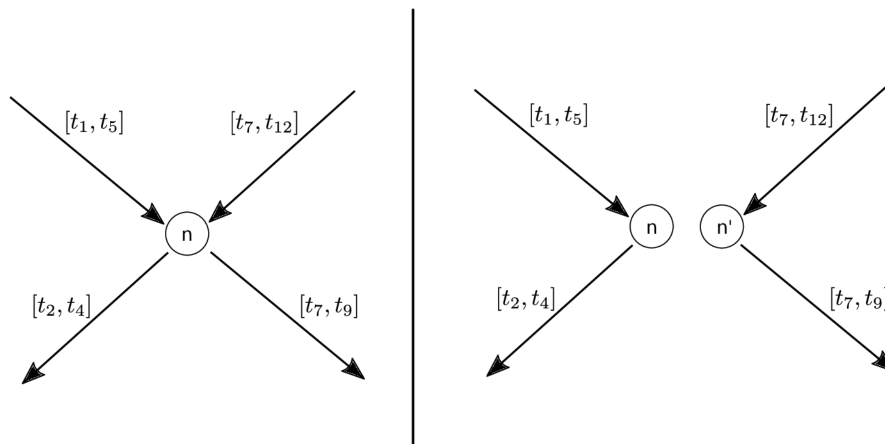
nekonzistence způsobená hranou s intervalem $[t_6, Now]$ mezi uzly n_1 a n_3. Namísto opravy intervalu této hrany jsme se rozhodli rozšířit interval $[t_2, t_{10}]$ hrany mezi uzly n_2 a n_1 na $[t_2, Now]$, jak je ukázáno v pravé části obrázku 7.10. Interval jsme rozšířili o interval $[t_{11}, Now]$, v kterém nastala nekonzistence. Otázkou ovšem je, zda v tomto intervalu vztah mezi uzly n_2 a n_1 skutečně existoval. Také zde nedocházelo k žádným propagacím směrem ke kořeni. K těm by mohlo dojít, kdyby například interval mezi uzly r a n_2 byl $[0, t_{10}]$ namísto $[0, Now]$.

7.5.2 Nekonzistence typu II

Tato nekonzistence říká, že intervaly platnosti příchozích obsahových hran do uzlu na sebe nenavazují. To znamená, že buď je v platnosti uzlu mezera anebo se intervaly překrývají. V případě mezery je možné řešit nekonzistenci vytvořením duplicitního uzlu. To je popsáno na obrázku 7.11. Na něm jsou zobrazeny intervaly, pro jejichž počáteční a koncové časové okamžiky platí $t_1 < t_2 < t_4 < t_5 < t_7 < t_9 < t_{12}$. V platnosti uzlu n je mezera $[t_5+1, t_7-1]$. Tu je možné odstranit, vytvoříme-li duplikovaný uzel n' tak, jak je ukázáno v pravé části obrázku 7.11.

Alternativní možnosti opravy nejsou příliš vhodné. Smazání hrany s intervalem $[t_1, t_5]$ nebo $[t_7, t_{12}]$ by vytvořilo v odchozích hranách uzlu n nekonzistenci typu I. Rozšíření intervalu jeho příchozích hran by zase mohlo zavést nekonzistenci typu I u počátečního uzlu těchto hran.

V případě překrytí vybereme jednu hranu, která se na překrytí podílí a upravíme její interval platnosti.

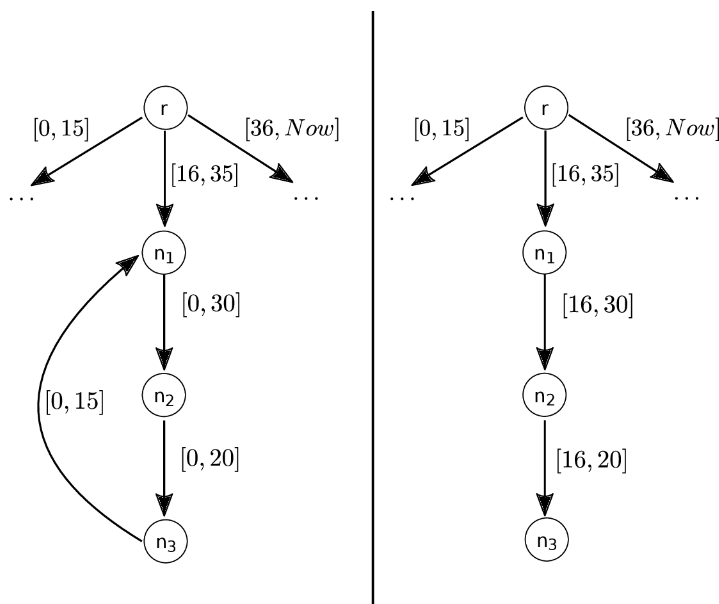


Obrázek 7.11: Oprava nekonzistence mezery v platnosti uzlu.

Pro detekci nekonzistence stačí seřadit intervaly příchozích obsahových hran v nějakém seznamu podle konce intervalu a pak seznam postupně projít a podmínku ověřit.

7.5.3 Nekonzistence typu III

Vzniká, existuje-li ve snímku abstraktního modelu cyklus. Příklad takové situace je ilustrován na obrázku 7.12 v jeho levé části, kdy je cyklus v intervalu [0, 15].



Obrázek 7.12: Cyklus v dokumentu a jeho oprava.

Pro detekci cyklu je možné využít následující vlastnost. Existuje-li v dokumentu cyklus, pak musí existovat nějaký uzel n_i, pro který platí, že jeho platnost není totožná s intervalem maximální souvislé cesty z kořene dokumentu r do n_i. Jednoduše řečeno, je to způsobeno tím, že do platnosti n_i je navíc započítaný interval hrany vytvářející cyklus. Takovým uzlem je na obrázku 7.12 uzel n_1. Interval maximální souvislé cesty je [16, 35] zatímco jeho platnost je [0, 35].

Oprava vyžaduje znalost intervalu nekonzistence I_I, který je příčinou cyklu. Poté je nutné smazat v tomto intervalu všechny obsahové hrany zapojené v cyklu a navíc toto mazání intervalů dále propagovat na všechny podstromy uzlů, které jsou v cyklu zapojeny. Při mazání mohou také vznikat nekonzistence typu II, které je nutné opravit. Jak taková oprava cyklu může vypadat je zobrazeno v pravé části obrázku 7.12.

7.5.4 Nekonzistence typu IV

Nastává, existuje-li více než jeden uzel se stejnou hodnotou atributu ID.

7.6 Rozšíření XPath

Výsledek dotazu v relační temporální databázi obsahuje kromě původního primárního klíče také časový interval platnosti záznamu, který je součástí primárního klíče. Abychom podobného chování dosáhli v temporální XML databázi, je nutné rozšířit výsledek dotazu o interval platnosti. Jazykem, který je určený pro dotazování nad XML dokumenty, je XPath. Konkrétně zde se budeme zabývat jeho verzí 2.0.

Dotaz jazyka XPath je cesta a výsledkem dotazu je sekvence uzlů na konci této cesty. Rozšíření tohoto jazyka bude spočívat v tom, že namísto sekvence uzlů bude výsledkem sekvence dvojic *(uzel, interval)*. Změní se taktéž sémantika výstupu. Nebude se jednat o sekvenci dvojic na konci *cesty*, ale o sekvenci dvojic na konci *souvislé cesty* tak, jak je definovaná v části 7.3. Tedy význam je takový, že výstup zaručuje kromě platnosti výsledných uzlů také platnost všech předchozích uzlů, z kterých výsledná sekvence vznikla. A samozřejmě sekvence dvojic *(uzel, interval)* je výstupem každého kroku, který se dále předává na vstup kroku následujícímu, případně je konečným výsledkem.

Podobně jako u temporálních relačních databází i zde může být nutné provádět restrukturalizaci. Ta je ovšem prováděna výhradně nad hodnotovými uzly, které patří uzlům s rozdílnými identifikátory. To je dáno abstraktním modelem a jeho podmínkami konzistence. Řekněme například, že bychom chtěli jména všech zaměstnanců ve společnosti. Pokud se jedná o větší společnost, může se stát, že dva zaměstnanci mají stejné jméno. Otázkou poté zůstává, zda je správné intervaly těchto dvou hodnotových uzlů sloučit. Pokud bychom vyžadovali chování co nejbližší k temporální relační databázi, pak by ke sloučení dojít mohlo. Záleží na vybraných sloupcích v dotazu *SELECT*. Ideálním řešením by nejspíš bylo nechat toto rozhodnutí na uživateli.

Dotazovací jazyk XPath nedokáže navíc vrátit snímek dokumentu v daný okamžik, to je nutné vyřešit jinak. Dokáže pouze obnovit v daném čase jeho část.

Pro podporu temporálních dotazů zavedeme speciální atributy *from* pro začátek intervalu platnosti, v XPath zkráceně jako *@from*, a *to* pro konec intervalu platnosti, v XPath zkráceně jako *@to*.

Pro lepší pochopení je uvedeno v této části několik příkladů nad obrázkem 7.1. Jednoduchý dotaz v příkladu 7.1 vrátí zaměstnance ve společnosti.

```
//employee
```

Příklad 7.1: Příklad jednoduchého temporálního dotazu po rozšíření XPath.

Jeho výstupem bude následující sekvence dvojic: $(6, [0, 20])$, $(10, [0, Now])$, $(14, [0, Now])$, $(16, [0, Now])$, $(24, [0, Now])$.

Složitější dotaz je pak uveden v příkladu 7.2. V něm se dotazujeme na zaměstnance, kteří pracují nepřetržitě jako vývojáři od okamžiku 21.

```
//department[name='development']//employee[@from <= 21 and @to
    = 'Now']
```

Příklad 7.2: Příklad složitějšího temporálního dotazu po rozšíření XPath.

Jeho výstupem je sekvence jedné dvojice: $(10, [0, Now])$.

7.7 Uložení datového modelu

Cílem této sekce je navrhnout takové uložení dat, aby bylo dotazování nad nimi co nejefektivnější. Protože se jedná o temporální databázi, vybíráme data platná v určitý časový interval. Konkrétně se jedná o souvislé cesty definované v části 7.3. Ty zaručují, že všechny uzly, které se zpracovávají během vyhodnocení dotazu, jsou platné v požadovaném intervalu. Jsou zde definovány dvě datové struktury. První vychází z návrhu představeného v části 5.2 a využívá se k vyhodnocování XPath dotazů. Druhá je určena pro rychlé vytváření snímků dokumentu v různých časových okamžicích a také obsahuje pořadí elementů v dokumentu v požadovaném časovém okamžiku. Jakým způsobem budou konkrétně uloženy a implementovány tyto struktury, není součástí této sekce. Navržené řešení je v tomto ohledu univerzální.

7.7.1 Uložení souvislých cest

XPath dotaz představuje, jak už bylo řečeno, cestu. V sekci 3.1 je tento jazyk podrobně popsán a je v ní ukázáno, jakými všemi směry je možné se v dokumentu pohybovat. Prakticky zde neexistují žádná omezení. Navržené uložení musí všechny tyto možné cesty obsahovat a nabízet jejich rychlé procházení. Výsledkem dotazu jsou vždy určité fragmenty dokumentu. Dokument navíc bude uložen v databázi. Předpokladem by tedy měla být také jeho velká velikost, z čehož vyplývá potřeba vyhnout se co nejvíce nutnosti jeho sekvenčního procházení. Proto je nutné jednotlivé části dokumentu rozdělit na co nejmenší části, aby při zpracování dotazu bylo nutné projít v databázi co nejmenší počet záznamů.

U první naší struktury, jak již bylo zmíněno, navážeme na sekci 5.2. Mimo jiné použijeme stejný příklad dokumentu s tím rozdílem, že jej rozšíříme o jeho historii změn. Základem pro rozdělení dokumentu je tedy relace ekvivalence. Na rozdíl ovšem od řešení v 5.2, kdy do stejné třídy patřily uzly sdílející stejnou sekvenci značek (název tagu či atributu) od kořene, je relace ekvivalence provedena nad souvislými cestami. To znamená, že v třídě bude navíc i interval platnosti uzlu konkrétně na konci souvislé cesty a že v tomto intervalu jsou platné i všechny uzly na souvislé cestě. Druhá struktura rozděluje dokument na třídy, kde do jedné třídy patří uzly, které náleží do souvislé cesty od kořene se stejnou délkou.

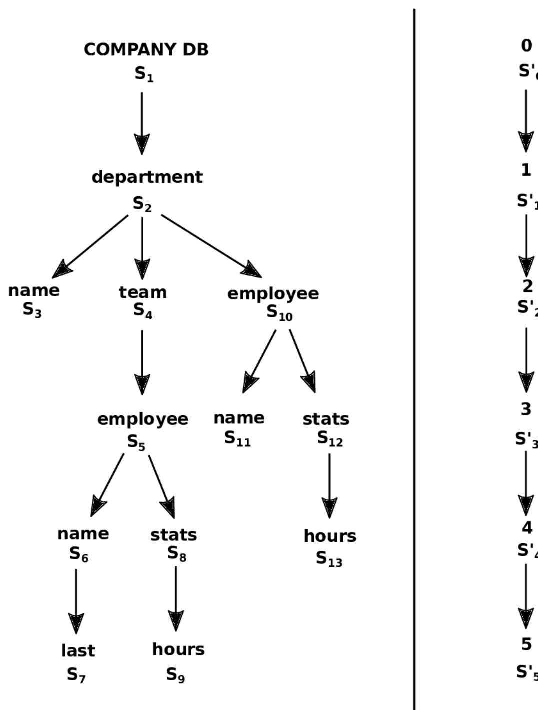
Třídy první struktury označíme zkratkou LCP, pod písmenem L ve zkratce vystupují značky uzlů (label) a pod dvojicí písmen CP souvislé cesty (continuous paths). Třídy druhé struktury pak označíme pouze zkratkou CP, neboť zařazení do relace ekvivalence je podmíněno pouze souvislou cestou.

7.7.2 Třídy LCP

Všechny uzly souvislých cest včetně jejich intervalu platnosti rozdělíme do těchto tříd na základě relace ekvivalence. Ve stejné třídě jsou uzly se stejnou sekvencí značek od kořene v abstraktním modelu (název tagu či atributu). Z relace ekvivalence vyplývá, že každý uzel

s konkrétním intervalem je právě v jedné třídě. Vytvořené třídy označíme názvem tagu či atributu uzlů, které do ní spadají. Mezi třídami taktéž definujeme vztah rodič-potomek a graficky ho znázorníme pomocí hrany. Tento vztah existuje právě tehdy, když existuje obsahová hrana mezi uzly, které do daných tříd patří.

Pro názornost je zde uveden příklad na abstraktním modelu 7.1. Třídy relace ekvivalence jsou uvedeny v levé části obrázku 7.13. Pro jednoduchost jsou označeny jako s_1 až s_{13}.



Obrázek 7.13: Rozdělení uzlů do tříd relace ekvivalence.

Rozdělení uzlů včetně jejich intervalu platnosti pro souvislou cestu do tříd popisuje tabulka 7.1.

s_1	0, [0, Now]			
s_2	1, [0, Now]	2, [0, Now]	3, [0, Now]	
s_3	4, [0, Now]	15, [0, Now]	25, [0, Now]	
s_4	5, [0, Now]			
s_5	6, [0, 20]	10, [0, Now]	14, [23, Now]	16, [0, 20]
s_6	7, [0, 20]	8, [0, Now]	17, [0, 20]	30, [23, Now]
s_7	9, [0, Now]			
s_8	13, [23, Now]	18, [0, 20]	31, [0, Now]	
s_9	11, [0, Now]	12, [23, Now]	19, [0, 20]	
s_{10}	14, [0, 22]	16, [21, Now]	24, [0, Now]	
s_{11}	17, [21, Now]	23, [0, Now]	30, [0, 22]	
s_{12}	13, [0, 22]	18, [21, Now]	22, [0, Now]	
s_{13}	12, [0, 22]	20, [0, 10]	21, [16, Now]	19, [21, Now]

Tabulka 7.1: Tabulka rozdělení uzlů do tříd LCP

V levém sloupci je uvedena třída a na stejném řádku uzly s intervaly, které do ní patří. Za povšimnutí stojí, že se některé uzly v tabulce vyskytují vícekrát v různých třídách a pokaždé s jiným intervalem. To je dáno tím, že byl element přesunut do jiného rodičovského elementu. Pak uzel náleží do jiné souvislé cesty (viz. druhá podmínka v 7.4) a také sekvence značek od kořene se liší. Například uzel 14 zastupuje zaměstnance jménem Scott. Ten pracoval nejprve v intervalu [0, 22] jako tester a pak začal pracovat jako vývojář. Třídy se liší, protože je zde navíc rodičovský element team. Častější je však situace, kdy se třídy lišit nebudou, pouze v nich bude uzel vícekrát. Například uzly 20 a 21 ve třídě s_{13}, ty představují změnu vykázaných hodin zaměstnance Lee.

7.7.3 Tabulky LCP

Popisuje uložení třídy LCP. Vzhledem k tomu, že se uzly váží ke konkrétní souvislé cestě a interval platnosti v tomto případě vyjadřuje platnost uzlu na konci této souvislé cesty, ukládají tyto tabulky ve skutečnosti hrany mezi uzly. Kromě samotného uzlu (sloupec *node*) zde bude uložen i počáteční uzel této hrany (sloupec *parent*), tedy rodičovský uzel. Dále také v tabulce bude sloupec umožňující u uzlu uložit hodnotový uzel (sloupec *value*). Samotný interval je rozdělen do sloupců *from* a *to*. Tabulka bude seřazena podle pořadí uzlů v dokumentu.

parent	**node**	**from**	**to**	**value**
hrana (s_1, s_2) - (company, department)				
0	1	0	Now	
0	2	0	Now	
0	3	0	Now	
hrana (s_2, s_{10}) - (department, employee)				
2	14	0	22	
2	16	21	Now	
2	24	0	Now	
hrana (s_4, s_5) - (team, employee)				
5	6	0	20	
5	10	0	Now	
5	14	23	Now	
5	16	0	20	
hrana (s_{12}, s_{13}) - (stats, hours)				
13	12	0	22	22
22	20	0	10	15
22	21	16	Now	27
18	19	21	Now	11
hrana (s_5, s_8) - (employee, stats)				
14	13	23	Now	
16	18	0	20	
10	31	0	Now	
hrana (s_6, s_7) - (name, last)				
8	9	0	Now	Lewis

Tabulka 7.2: Tabulky LCP část 1.

parent	**node**	**from**	**to**	**value**
hrana (s_5, s_6) - (employee, name)				
6	7	0	20	Smith
10	8	0	Now	
16	17	0	20	White
14	30	23	Now	Scott
hrana (s_8, s_9) - (stats, hours)				
31	11	0	Now	15
13	12	23	Now	22
18	19	0	20	11
hrana (s_2, s_4) - (department, team)				
1	5	0	Now	
hrana (s_2, s_3) - (department, name)				
1	4	0	Now	development
2	15	0	Now	testing
3	25	0	Now	support
hrana (s_{10}, s_{11}) - (employee, name)				
16	17	21	Now	White
24	23	0	Now	Lee
14	30	0	22	Scott
hrana (s_{10}, s_{12}) - (employee, stats)				
14	13	0	22	
16	18	21	Now	
24	22	0	Now	

Tabulka 7.3: Tabulky LCP část 2.

V 7.2 a 7.3 jsou uloženy všechny LCP tabulky pro abstraktní model 7.1. Tabulky jsou

uloženy pro přehlednost za sebou. Je možné si všimnout, že mezi tabulkami existuje prostřednictvím sloupce *parent* vztah. Tyto vztahy jsou pro každé dvě tabulky vždy v rámci dvou tříd a na obrázku 7.13 jsou znázorněny hranou. Platnost uzlu ve sloupci *node* je na dané cestě vždy podintervalem platnosti uzlu ve sloupci *parent*. Některé hodnotové uzly (obsah sloupce *value*) se objevují ve více tabulkách, ale s různými intervaly. To se děje proto, že tabulky obsahují souvislé cesty namísto uzlů. V pozdější kapitole bude popsáno, jakým způsobem se s tabulkou pracuje.

7.7.4 Třídy CP

Jedná se o rozdělení do tříd za základě délky souvislé cesty. Tedy v jedné třídě jsou uzly spolu s jejich intervalem platnosti na konci souvislé cesty, které jsou v abstraktním modelu ve stejné hloubce. Mezi třídami bude opět vztah rodič-potomek v případě, že je mezi uzly v těchto třídách. Stejně jako u LCP třídy obsahuje CP třída dvojice (uzel, interval), u které je zaručeno, že se vyskytuje právě v jedné třídě. V zásadě se jedná o sloučení několika LCP tříd do tříd CP. To je ukázáno na obrázku 7.13. V jeho pravé části jsou zobrazeny třídy CP pro abstraktní model na obrázku 7.1. Třídy jsou označeny jako s'_0 až s'_5. Také je zde uvedena délka souvislé cesty 0 až 5. Například třída LCP s'_2 odpovídá třídám s_3, s_4 a s_{10}. Konkrétní rozdělení uzlů do tříd je ukázáno v tabulce 7.4.

s'_0	0, [0, Now]			
s'_1	1, [0, Now]	2, [0, Now]	3, [0, Now]	
s'_2	4, [0, Now] 5, [0, Now] 14, [0, 22]	15, [0, Now] 16, [21, Now]	25, [0, Now] 24, [0, Now]	
s'_3	6, [0, 20] 17, [21, Now] 13, [0, 22]	10, [0, Now] 23, [0, Now] 18, [21, Now]	14, [23, Now] 30, [0, 22] 22, [0, Now]	16, [0, 20]
s'_4	7, [0, 20] 13, [23, Now] 12, [0, 22]	8, [0, Now] 18, [0, 20] 20, [0, 10]	17, [0, 20] 31, [0, Now] 21, [16, Now]	30, [23, Now] 19, [21, Now]
s'_5	9, [0, Now] 11, [0, Now]	 12, [23, Now]	 19, [0, 20]	

Tabulka 7.4: Tabulka rozdělení uzlů do tříd CP

7.7.5 Tabulky CP

Tato část pojednává o uložení tříd CP. Každý řádek bude obsahovat seznam uzlů (seřazený podle jejich relativního dokumentového pořadí ve sloupci *valid*) a interval jejich platnosti (rozdělený do sloupců *from* a *to*). Pro každou hloubku bude existovat samostatná tabulka. Problémem, který je nutné vyřešit, je převod CP tříd do tabulek, neboť ty obsahují dvojice $(uzel, interval)$.

Požadované intervaly v tabulce CP jsou získány tak, že se vezmou všechny intervaly, které označují nějakou souvislou cestu délky k, a rozdělí se podle potřeby, aby se získala nejmenší možná množina po dvou disjunktních intervalů, která obsahuje všechny intervalové úseky původních intervalů.

Tabulky lze použít při vytváření snímků dokumentu a umožňují efektivní nalezení všech uzlů platných během daného intervalu.

V 7.5 jsou uvedeny všechny tabulky pro abstraktní model 7.1. Pro přehlednost jsou sloučeny do jediné tabulky.

from	**to**	**valid**
	hloubka 0	
0	Now	(0)
	hloubka 1	
0	Now	(1, 2, 3)
	hloubka 2	
0	20	(4, 5, 14, 15, 24, 25)
21	22	(4, 5, 14, 15, 16, 24, 25)
23	Now	(4, 5, 15, 16, 24, 25)
	hloubka 3	
0	20	(6, 10, 13, 16, 22, 23, 30)
21	22	(10, 13, 17, 18, 22, 23, 30)
23	Now	(10, 14, 17, 18, 22, 23)
	hloubka 4	
0	10	(7, 8, 12, 17, 18, 20, 31)
11	15	(7, 8, 12, 17, 18, 31)
16	20	(7, 8, 12, 17, 18, 20, 21, 31)
21	22	(8, 12, 17, 19, 21, 31)
23	Now	(8, 13, 19, 21, 30, 31)
	hloubka 5	
0	20	(9, 11, 19)
21	22	(9, 11)
23	Now	(9, 11, 12)

Tabulka 7.5: Tabulky CP

7.8 Vyhodnocení XPath dotazů

Tato část popisuje, jakým způsobem jsou za pomoci tabulek LCP a CP vyhodnoceny XPath dotazy. Navazuje se zde na část 3.1, kde se popisuje obecně jazyk XPath, a na část 7.6, kde je popsáno temporální rozšíření. To spočívá v tom, že jazyk vyhodnocuje dotazy nad souvislými cestami a uzel je rozšířen na dvojici $(uzel, interval)$, kde interval představuje platnost uzlu na konci souvislé cesty. To je dáno tím, že se uzel může v různých časových intervalech nacházet na různých cestách.

Pro vyhodnocení dotazu budeme potřebovat jednak navigaci v dokumentu, kdy se budeme pohybovat různými směry v abstraktním modelu, a dále se neobejdeme bez filtrování uzlů, kdy budeme vybírat uzly splňující určitý temporální, nebo hodnotový predikát.

Pro pohyb v dokumentu je nutné rozšířit dvojici $(uzel, interval)$ o název třídy, v které se tato dvojice nachází, abychom dokázali určit, ve kterých třídách hledat její předchůdce a následníky. Obecně pracujeme proto s trojicí *(třída, uzel, interval)*. Třída ovšem bude před navrácením konečného výsledku dotazu odstraněna. Pro navigaci použijeme následující

metody. Všechny uvedené metody budou pracovat nad seznamem trojic.

- getParent(Label): Vrátí přímé předky uzlů v seznamu.
- getDescendants(Label): Vrátí potomky uzlů v seznamu (dvě lomítka).
- getChildren(Label): Vrátí přímé potomky uzlů v seznamu.
- getAncestors(Label): Vrátí předky uzlů v seznamu.

Parametr *Label* je v abstraktním modelu značka uzlu. Tedy v rámci dokumentu se jedná o název tagu či atributu a spolu se směrem určuje tabulky, v kterých se budou uzly hledat. Pro filtrování uzlů budeme potřebovat následující metody. Parametr *valPred* je hodnotový predikát a parametr *tempPred* je temporální predikát.

- valFilter(valPred): Aplikuje na uzly v seznamu predikát *valPred* a výsledek vrátí jako nový seznam.
- tempFilter(tempPred): Aplikuje na uzly v seznamu predikát *tempPred* a výsledek vrátí jako nový seznam.

Výše zmíněné metody nám dovolují vyhodnotit krok výrazu XPath. Postupným vyhodnocováním kroků bude postupně vyhodnocen celý výraz. Použití těchto metod bude demonstrováno na příkladu dotazu 7.3.

```
//department[name='development']//employee/name[@from<=21 and
   @to='Now']
```

Příklad 7.3: Dotaz pro příklad vyhodnocení XPath dotazu.

Dotaz vrací jména zaměstnanců, kteří pracují jako vývojáři nepřetržitě od okamžiku 21. Vyhodnocen bude pomocí sekvence volání metod uvedené v příkladu 7.4.

```
/* Vložíme do seznamu kořen dokumentu. */
list.add((s1, 0, [0, Now]));
list = list.getDescendants("department");
/* list obsahuje (s2, 1, [0, Now]), (s2, 2, [0, Now]), (s2, 3,
    [0, Now]) */
list = list.getChildren("name");
/* list obsahuje (s3, 4, [0, Now]), (s3, 15, [0, Now]), (s3,
   25, [0, Now]) */
list = list.valFilter("development");
/* list obsahuje (s3, 4, [0, Now]) */
list = list.getParent("department");
/* list obsahuje (s2, 1, [0, Now]) */
list = list.getDescendants("employee");
/* list obsahuje (s5, 6, [0, 20]), (s5, 10, [0, Now]), (s5,
   14, [23, Now]), (s5, 16, [0, 20]), */
list = list.getChildren("name");
/* list obsahuje (s6, 7, [0, 20]), (s6, 8, [0, Now]), (s6, 30,
    [23, Now]), (s6, 17, [0, 20]), */
list = list.tempFilter("@from<=21 and @to=Now");
```

```
/* list obsahuje (s6, 8, [0, Now]), výsledek výrazu je (8, [0,
    Now]) */
```

Příklad 7.4: Vyhodnocení XPath dotazu.

U metody *tempFilter(tempPred)* bude použita tabulka CP. Ta obsahuje správné pořadí uzlů v daný časový interval. U ostatních metod jsou vždy použity tabulky LCP.

7.9 Aktualizace s transakčním časem

V této části jsou popsány aktualizace nad temporálním XML dokumentem. Popisuje tři druhy změn nad dokumentem: vložení nového uzlu, odstranění uzlu a aktualizaci obsahových hran. Protože se zde zabýváme pouze transakčním časem, nastávají všechny aktualizace v aktuální časový okamžik. Ten bude označen jako t_c.

7.9.1 Vložení nového uzlu

Vložení nového uzlu do temporálního XML dokumentu vyžaduje specifikování nového uzlu n', který má být vložen, a aktuálního uzlu n (tj. uzel s příchozí obsahovou hranou, jejíž koncový bod intervalu je Now). Nový uzel n' a obsahová hrana z n do n' s intervalem platnosti $[t_c, Now]$ jsou přidány do abstraktního modelu. DDL (Data Definition Language) syntaxe pro vkládání je popsána v 7.5.

```
FOR variable IN PathExpression
  INSERT ChildExpression
  [VALUE value]
```

Popis syntaxe 7.5: DDL pro vložení nového uzlu.

PathExpression vrací dvojice *(node, interval)*. Pro každou dvojici takovou, že $interval.TO = Now$ je přidán nový uzel jako potomek *node* s podcestou danou *ChildExpression*. Pokud je nový uzel hodnotový uzel, klíčové slovo *VALUE* dovoluje uvedení odpovídající hodnoty. Toto klíčové slovo je vynecháno při vkládání elementového uzlu.

Aktualizace tabulek bude probíhat následovně. Vyhodnotíme *PathExpression*, což je popsáno v sekci 7.8. Vybereme pouze výsledné dvojice, které mají koncový interval *Now*. Tím získáme uzly pro sloupec *parent*. Pokud žádné nejsou, končíme. Z výrazu *ChildExpression* určíme, kam se mají uzly vložit, případně zda se mají vytvořit nové LCP tabulky. Vygenerujeme identifikátory nových uzlů. Ty se vloží u cílových LCP tabulek do sloupců *node*, do sloupce *parent* vložíme uzly získané z vyhodnocení *PathExpression*, do sloupce *from* vložíme hodnotu t_c a do sloupce *to* hodnotu *Now*. Pokud je použita část *VALUE*, nastavíme hodnotu *value* ve sloupci *value*.

Dále aktualizujeme CP tabulky. Pokud pro danou hloubku existují, ukončíme platnost jejich aktuálního záznamu v $t_c - 1$ a vložíme nový aktuální záznam s intervalem $[t_c, Now]$. Sloupec *valid* zůstane stejný, pro nový řádek ho zkopírujeme a přidáme do něj nový uzel. Pokud neexistují, vytvoříme je, přidáme do nich záznam s intervalem $[t_c, Now]$ a ve sloupci *valid* bude nový uzel.

V 7.6 je uveden příklad, na kterém je celý proces demonstrován. Pro jednoduchost předpokládejme aktuální čas $t_c = 120$.

```
FOR $p IN //employee[name/last='Lewis']/stats
  INSERT $p/minutes
```

```
VALUE '33,2'
```

Příklad 7.6: Vložení nového uzlu.

Po vyhodnocení části *PathExpression* získáme aktuálně platnou dvojici $(31, [0, Now])$. Dále vytvoříme novou LCP tabulku pro hranu $(stats, minutes)$ a vložíme do ní nový řádek. Nová tabulka je zobrazena jako tabulka 7.6.

parent	node	from	to	value
hrana (s_8, s_{14}) - (stats, minutes)				
31	32	120	Now	33,2

Tabulka 7.6: Tabulka LCP po vložení nového uzlu

Následně je nutné aktualizovat tabulku CP pro hloubku 5. Tato aktualizace bude mít dvě fáze. V první fázi upravíme hodnotu ve sloupci *to* na řádku s aktuálním intervalem na 119. V druhé přidáme nový řádek s intervalem $(120, Now)$. Obsah této tabulky před aktualizací je zobrazen v 7.7 a po aktualizaci v 7.8.

from	to	valid
hloubka 5		
0	20	(9, 11, 19)
21	22	(9, 11)
23	Now	(9, 11, 12)

Tabulka 7.7: Tabulka CP před aktualizací

from	to	valid
hloubka 5		
0	20	(9, 11, 19)
21	22	(9, 11)
23	119	(9, 11, 12)
120	Now	(9, 11, 12, 32)

Tabulka 7.8: Tabulka CP po aktualizaci

Poslední, co zbývá vyřešit, jsou možné nekonzistence, které byly definované v sekci 7.5. Předpokládáme, že aktualizace jsou provedeny nad konzistentním dokumentem a musí zanechat tento dokument v konzistentním stavu. V případě vložení uzlu n' má nový uzel pouze jednu příchozí hranu, což znamená, že nekonzistence typu II nemohou nastat. Vložený uzel nemá také žádné odchozí hrany. Z tohoto důvodu nekonzistence typů I a III (cykly) nemohou být zavedeny.

7.9.2 Odstranění uzlu

Z temporálního XML dokumentu můžeme smazat (ve smyslu temporální databáze) atributové uzly (s výjimkou atributů typu ID), elementové uzly a referenční hrany. Při mazání uzlu n v čase t_d, je *Now* nahrazeno t_d v koncovém bodu intervalu platnosti jeho příchozí obsahové hrany. Totéž platí pro všechny obsahové hrany v aktuálním podstromu n (podstrom s kořenem n, kde všechny hrany mají koncový bod intervalu platnosti *Now*). Referenční hrany jsou odstraněny nastavením jejich koncového bodu intervalu platnosti na t_d. Žádná kontrola konzistence zde není nutná. Tedy tato operace vždy zanechá dokument v konzistentním stavu.

Celý proces bude popsán pomocí příkladu v 7.7. Na prvním řádku získáme jedinou aktuální dvojici $(14, [23, Now])$. Ta obsahuje rodičovský uzel uzlu, který má být smazán.

```
FOR $e IN //employee[name='Scott']
    DELETE node $e/stats
```

Příklad 7.7: Odstranění uzlu.

Na druhém řádku zjistíme, že se má smazat uzel 13 z tabulky pro hranu (s_5, s_8) a uzel 12 v jeho podstromu v tabulce pro hranu (s_8, s_9) v čase 119. To je provedeno tak, že se nahradí ve sloupci *to* hodnota *Now* za hodnotu *119*. V 7.9 je uveden stav LCP tabulek před odstraněním uzlu a v 7.10 je uveden stav LCP tabulek po odstranění uzlu. Nyní už zbývá

parent	**node**	**from**	**to**	**value**
hrana (s_5, s_8) - (employee, stats)				
14	13	23	Now	
16	18	0	20	
10	31	0	Now	
hrana (s_8, s_9) - (stats, hours)				
31	11	0	Now	15
13	12	23	Now	22
18	19	0	20	11

Tabulka 7.9: Tabulky LCP před smazáním

parent	**node**	**from**	**to**	**value**
hrana (s_5, s_8) - (employee, stats)				
14	13	23	119	
16	18	0	20	
10	31	0	Now	
hrana (s_8, s_9) - (stats, hours)				
31	11	0	Now	15
13	12	23	119	22
18	19	0	20	11

Tabulka 7.10: Tabulky LCP po smazání

jenom provést obdobnou aktualizaci pro tabulky CP. Uzly, které mají být odstraněny, se nacházejí v hloubce 4 a 5. Aktuální stav tabulek pro tyto hloubky je uveden v 7.11. Nový stav těchto tabulek je uveden v 7.12. Odstranění uzlu z těchto tabulek bylo provedeno tak, že byla ukončena platnost aktuálního záznamu a byl přidán nový aktuální záznam, v kterém již není uveden smazaný uzel.

from	**to**	**valid**
hloubka 4		
0	10	(7, 8, 12, 17, 18, 20, 31)
11	15	(7, 8, 12, 17, 18, 31)
16	20	(7, 8, 12, 17, 18, 20, 21, 31)
21	22	(8, 12, 17, 19, 21, 31)
23	Now	(8, 13, 19, 21, 30, 31)
hloubka 5		
0	20	(9, 11, 19)
21	22	(9, 11)
23	Now	(9, 11, 12)

Tabulka 7.11: Tabulky CP před smazáním

from	**to**	**valid**
hloubka 4		
0	10	(7, 8, 12, 17, 18, 20, 31)
11	15	(7, 8, 12, 17, 18, 31)
16	20	(7, 8, 12, 17, 18, 20, 21, 31)
21	22	(8, 12, 17, 19, 21, 31)
23	119	(8, 13, 19, 21, 30, 31)
120	Now	(8, 19, 21, 30, 31)
hloubka 5		
0	20	(9, 11, 19)
21	22	(9, 11)
23	119	(9, 11, 12)
120	Now	(9, 11)

Tabulka 7.12: Tabulky CP po smazání

7.9.3 Aktualizace obsahové hrany

Tato aktualizace umožňuje změnu rodičovského uzlu. V kontextu XML dokumentu se může jednat například o přesunutí elementu na jiné místo v dokumentu. Pro její provedení bude potřeba zkopírovat všechny aktuální uzly, kterých se změna týká, včetně jejich aktuálních podstromů, do jiných tabulek a upravit původní a nové záznamy. Tento proces je ukázán

na příkladu 7.8. Ten přesune zaměstnance z oddělení, kde právě pracuje (jako tester), do týmu vývojářů.

```
FOR //employee[name='Lee']
    SET PARENT
        //department[name='development']/team
```

Příklad 7.8: Aktualizace obsahové hrany.

Vyhodnocení probíhá následovně. Na prvním řádku je výsledkem aktuální dvojice (24, [0, Now]) pocházející z LCP tabulky pro hranu (s_2, s_{10}). Ta má aktuální potomky v LCP tabulkách hran (s_{10}, s_{11}), (s_{10}, s_{12}) a (s_{12}, s_{13}). Ty budeme aktualizovat. Po vyhodnocení XPath výrazu na třetím řádku získáme dvojici $(5, [0, Now])$. Ta pochází z LCP tabulky pro hranu (s_2, s_4) a její třídy následníků, do kterých budeme přidávat záznamy, jsou (s_4, s_5), (s_5, s_6), (s_5, s_8) a (s_8, s_9). Tyto vztahy jsou zobrazeny poměrně přehledně v levé části obrázku 7.13. V tabulce 7.13 a tabulce 7.14 jsou uvedeny tabulky LCP po aktualizaci. Tabulky CP se aktualizují obdobně.

parent	**node**	**from**	**to**	**value**
hrana (s_2, s_{10}) - (department, employee)				
2	14	0	22	
2	16	21	Now	
2	24	0	119	
hrana (s_4, s_5) - (team, employee)				
5	6	0	20	
5	10	0	Now	
5	14	23	Now	
5	16	0	20	
5	24	120	Now	
hrana (s_{12}, s_{13}) - (stats, hours)				
13	12	0	22	22
22	20	0	10	15
22	21	16	119	27
18	19	21	Now	11
hrana (s_5, s_8) - (employee, stats)				
14	13	23	Now	
16	18	0	20	
10	31	0	Now	
24	22	120	Now	

Tabulka 7.13: Aktualizace hrany část 1.

parent	**node**	**from**	**to**	**value**
hrana (s_5, s_6) - (employee, name)				
6	7	0	20	Smith
10	8	0	Now	
16	17	0	20	White
14	30	23	Now	Scott
24	23	120	Now	Lee
hrana (s_8, s_9) - (stats, hours)				
31	11	0	Now	15
13	12	23	Now	22
18	19	0	20	11
22	21	120	Now	27
hrana (s_2, s_4) - (department, team)				
1	5	0	Now	
hrana (s_{10}, s_{11}) - (employee, name)				
16	17	21	Now	White
24	23	0	119	Lee
14	30	0	22	Scott
hrana (s_{10}, s_{12}) - (employee, stats)				
14	13	0	22	
16	18	21	Now	
24	22	0	119	

Tabulka 7.14: Aktualizace hrany část 2.

Kontrola konzistence se v tomto případě angažuje trochu více. Vedle ověření, že nový rodičovský uzel je aktuální, potřebujeme ověřit podmínku konzistence III, tj. že nejsou aktualizací zavedeny žádné cykly (v případě aktualizací provedených nad aktuálními uzly se to omezuje na ověření, že žádné cykly nejsou zavedeny v aktuálním okamžiku).

Kapitola 8

Analýza hotových řešení

Před návrhem vlastního řešení byla provedena analýza dvou projektů. Jedná se o diplomové práce [4] a [14]. V této kapitole jsou popsána již existující řešení podobného problému.

8.1 Temporální rozšíření objektově-relačního mapování

Práce popisuje rozšíření rozhraní Java Data Objects (JDO). Rozhraní JDO umožňuje perzistentní ukládání objektů do relačně-objektové databáze. Po rozšíření výsledný produkt nabízí možnost dotazovat se i na předchozí stavy objektů. Aby toho autor dosáhl, zavádí pro všechny třídy, jejichž instance se ukládají do databáze, nového společného předka, v kterém jsou uložena metadata potřebná pro temporální rozšíření. Atributy této třídy jsou uvedeny na obrázku 8.1.

Je zde ukládán jak čas platnosti (atribut *valid*), tak transakční čas (atribut *transaction*). Čas platnosti je ukládán ve formě seznamu instancí třídy TemporalInterval. Ta obsahuje počátek a konec platnosti záznamu, pro reprezentaci času je použita uvnitř této třídy třída Javy Calendar. Díky tomu, že se jedná o seznam, pak může být platnost objektu složena z více disjunktních intervalů. Aby mohly být atributy *valid* a *transaction* zpracovávány stejným způsobem, jsou oba reprezentovány stejným typem, ale prakticky obsahuje atribut transaction vždy jediný interval.

Při mazání objektu v temporální databázi nedochází k jeho fyzickému odstranění. Pokud má být smazána pouze část platnosti objektu, aktualizují se jeho intervaly časů platnosti. V opačném případě se nastaví v metadatech atribut *deleted*, objekt zůstane v databázi, ale není načítán v dotazech.

TemporalBase
-temporalId : String -itemId : long -temporalVersion : int -current : boolean -deleted : boolean -valid : TemporalIntervalList -transaction : TemporalInterval -validityEditable : boolean

Obrázek 8.1: Třída obsahující temporální metadata.

Atribut *itemId* je identifikátorem všech temporálních verzí jedné instance. Pomocí něho lze identifikovat všechny verze jednoho objektu. Každá verze pak má unikátní hodnotu atributu *temporalVersion*. Atribut *temporalId* slučuje oba předchozí atributy dohromady v textové podobě (*itemId* `+ "v" +` *temporalVersion*), tento atribut slouží jako primární klíč a je unikátní díky tomu, že se skládá z obou předchozích atributů.

Pro fyzické mazání (odsávání) se využívá hodnoty transakčního času. To je výhodné, neboť na rozdíl od času platnosti tento čas nese informaci o skutečném stáří dat v databázi. Spolu s transakčním časem se používá atribut *current*, jehož hodnota říká, zda se jedná o aktuálně platná data, tedy zabraňuje smazání těchto dat v případě, že jsou v databázi již delší dobu. Navíc je možné tento atribut použít i pro výběr aktuálně platných objektů.

Pro zajištění referenční integrity mezi objekty jsou ukládány v databázi všechny vazby mezi nimi ve formě *(zdroj, cíl)*. Po každé databázové operaci, která mění stav uložených záznamů, se pak pomocí těchto vazeb ověřují ukládaná nebo mazaná data. Problém ověření referenční integrity byl v projektu ze všech řešených problémů nejnáročnější.

Atribut *validityEditable* slouží pro kontrolu, zda byl objekt načten pomocí rozšířeného rozhraní JDO. V případě, že bylo použito původní rozhraní, zabraňuje změně času platnosti.

Při ukládání temporálních dat dochází, kromě již zmíněné kontrole referenční integrity, k načtení předchozí verze objektu a k jeho porovnání s potenciálně novou verzí. Pokud se objekty liší jenom v platnosti, objekt se pouze aktualizuje (rozšíří se jeho platnost, tento způsob shlukování je jediný implementovaný), jinak se vytvoří kopie. Ta reprezentuje starší verzi objektu. Její koncový čas transakčního intervalu se nastaví na aktuální časový okamžik a atribut *current* se nastaví na false. U původního objektu je nastaven počáteční bod transakčního intervalu na aktuální časový okamžik (koncový zůstává nastaven na nekonečno) a inkrementuje se jeho verze.

Vzhledem k tomu, že je možné nastavovat aktuálně platnému objektu libovolný čas platnosti, je prováděna při každém ukládání dat kontrola prázdnosti průniků časů platnosti dat. Pokud je nalezen objekt s neprázdným průnikem, jsou odečteny od jeho intervalů platnosti intervaly platnosti aktuálně ukládaného objektu. Není zde dovoleno editovat jiný než aktuálně platný objekt (detekuje se pomocí atributu *current*) z důvodu, že by to vedlo k větvení historie.

8.2 TSQL2 interpret nad relační databází

Práce popisuje implementaci interpretu podmnožiny jazyka TSQL2 pro překlad do SQL. Implementace je provedena v jazyce Java jako nadstavba nad ovladačem JDBC.

Je zde vytvořena pomocná tabulka uchovávající metadata o tabulkách v databázi. V každém jejím záznamu je uloženo jméno tabulky, zda tabulka obsahuje transakční čas, čas platnosti (zda jde o intervaly nebo časové okamžiky) a zda má být automaticky prováděno odsávání. Pokud má, je zde uvedena konkrétní časová značka, pokud nemá, je uvedena časová značka vytvoření tabulky. To zajistí, že žádný záznam nebude staršího času. Odsávání lze nastavit i relativně. Dále záznam obsahuje měřítko a přesnost platnosti času. Také jsou v ní uvedeny výchozí hodnoty pro měřítko a přesnost platnosti času. Ty se použijí, pokud uživatel tento údaj nespecifikuje. Pro odmazávání starých dat je použit transakční čas. Pokud ho tabulka neobsahuje, není ho možné u tabulky nastavit.

Další pomocnou tabulku bylo nutné vytvořit pro uchovávání naposledy vygenerovaných hodnot typu *SURROGATE*. Atributy záznamů jsou název tabulky, název sloupce (tyto dva atributy tvoří primární klíč) a volná hodnota (ta se postupně inkrementuje).

Pro transakční čas jsou použita dvě časová razítka s unixovým časem (udává počet sekund od data 1.1.1970). Každé razítko má velikost 64 bitů. Pro čas platnosti je pro volbu časového okamžiku použito jedno časové razítko o velikosti 64 bitů a pro reprezentaci pomocí intervalu dvě časová razítka, každé taktéž o velikosti 64 bitů. V obou případech hodnota udává počet jednotek od počátku dané časové osy.

Pokud uživatel požaduje, aby se v dotazu provedla restrukturalizace podle zadaných sloupců, postupuje se následovně. Vyberou se záznamy z tabulky, seřadí se podle vybraných sloupců a časů platnosti, záznamy se projdou a ty, u kterých je to možné, se sloučí. Výsledek pak uloží do nové dočasné tabulky a s tou poté pracuje namísto původní tabulky.

K odmazávání starých záznamů (vacuuming) v tabulce dochází při každém jejím použití. Zajištění referenční integrity zde není implementováno.

Kapitola 9

Návrh

V kapitole 7 bylo popsáno řešení, jak rozšířit XML dokument o časovou dimenzi a dále s ním pracovat. Toto obecné řešení bylo převzato z cizího zdroje a tato kapitola má za cíl doplnit jeho chybějící části tak, aby výsledný produkt byl použitelný v praxi.

Konečným cílem je rozšíření umožňující ukládat v databázi XML dokumenty, a zároveň zaznamenávat jejich historii. Omezíme se zde pouze na použití transakčního času, nicméně informace obsažené v kapitole 7 by měly být dostatečné i pro návrh databáze s časem platnosti.

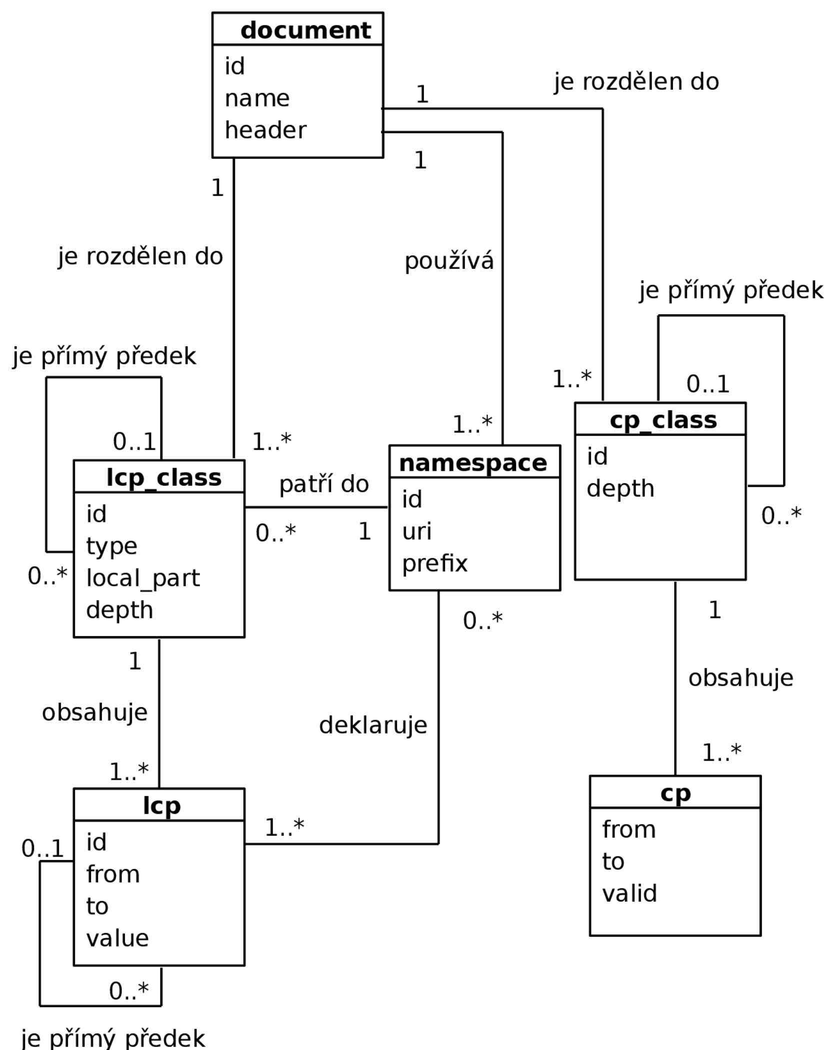
9.1 Uložení dat v databázi

V této části je navrženo uložení dat do relační databáze. Hlavní důvod pro výběr relační databáze byl její velká rozšířenost, což by mělo potencionálnímu uživateli usnadnit použití celé knihovny.

Dvěma klíčovými entitami, na kterých je návrh na obrázku 9.1 postaven, jsou entity *cp* a *lcp*. Zjednodušeně, entita *lcp* ukládá data o všech uzlech v dokumentu a entita *cp* data o jejich uspořádání. Jak již bylo popsáno v kapitole 7, uzly jsou rozděleny do tříd ekvivalence. Ty jsou reprezentovány entitami *cp_class* a *lcp_class*, v nichž jsou uložena navíc data, která uzly ve třídách sdílí. Pro možnost uchovávat v databázi více dokumentů slouží pak entita *document*. Poslední entitou je *namespace*. Ta obsahuje prefixy a URI jmenných prostorů.

Z formálního hlediska je v relační databázi uložen orientovaný graf s jediným kořenovým uzlem. To se projevuje v návrhu tak, že entity *lcp*, *lcp_class* a *cp_class* mají asociace unárního stupně. Na úrovni tabulek pak řádek v těchto tabulkách reprezentující kořenový uzel je identifikován pomocí nulové hodnoty ve sloupci *depth*. Za povšimnutí dále stojí vztah mezi entitami *lcp* a *lcp_class* respektive *cp* a *cp_class*, kde je minimální kardinalita rovna jedné. To je dáno definicí XML dokumentu. Ta říká, že každý XML dokument musí mít kořenový elementový uzel. Poměrně složité jsou vztahy u entity *namespace*. Vazba s entitou *lcp* ukazuje elementový uzel, jenž jmenný prostor deklaruje. Vazba s entitou *lcp_class* říká, že uzly patřící do stejné třídy musí patřit i ke stejnému jmennému prostoru. To lze později využít pro rychlé vyhledání uzlů XML dokumentu podle jmenného prostoru a vztah s entitou *document* umožňuje v budoucnosti odstranit z databáze dokument i s jeho jmennými prostory, aniž by byla narušena referenční integrita.

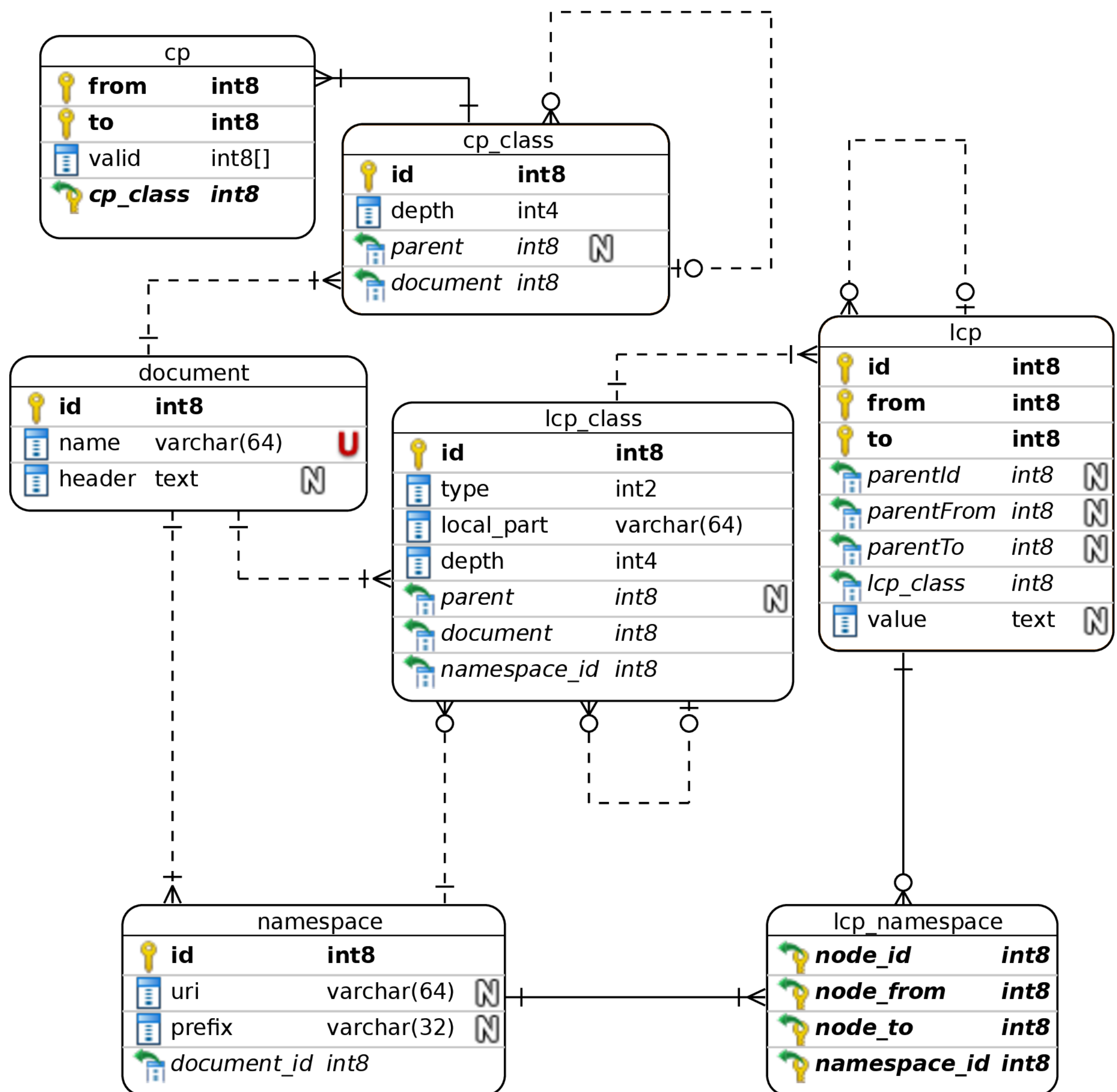
Návrh počítá i s možností seskupovat dokumenty do skupin. V jedné skupině tak mohou být uloženy dokumenty určitého druhu a nad každou skupinou mohou být definována práva.



Obrázek 9.1: Konceptuální diagram databáze.

Toho je docíleno pomocí schémat relační databáze, kdy je umožněno uživateli vytvořit si v databázi schéma, v němž je vytvořen potřebný soubor tabulek. Nastavení práv u schémat je pak ponecháno v režii uživatele. Dalším účelem schémat je oddělení XML dokumentů od zbytku databáze.

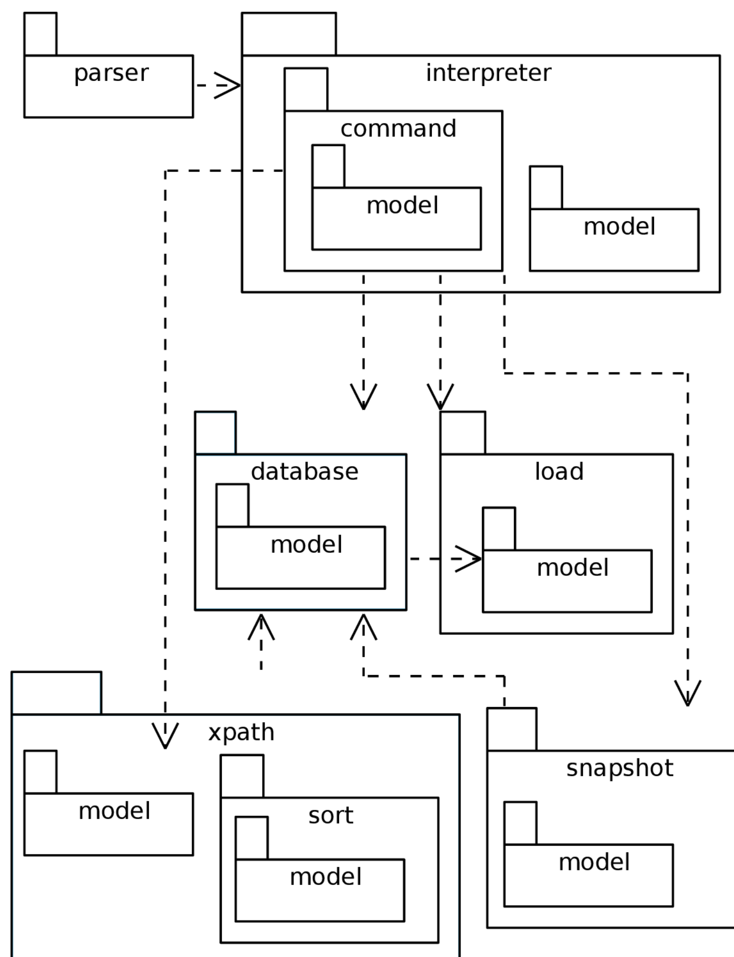
Na obrázku 9.2 je zobrazena databáze na úrovni tabulek. Zde je vhodné upozornit na dvě věci. První z nich je uložení pořadí uzlů, které je v relační databázi poměrně komplikované. K tomu je použito v tabulce *cp* pole *valid*, které ukládá vždy pořadí všech uzlů, jenž mají v XML stromu stejnou hloubku a stejný interval platnosti v rámci jednoho dokumentu. Tou druhou je minimální kardinalita mezi tabulkami *namespace* a *document*. Ta je nastavena na hodnotu 1. Z důvodu zjednodušení implementace se ukládá u každého dokumentu prázdný jmenný prostor.



Obrázek 9.2: Databázové schéma.

9.2 Návrh aplikace

Tato část popisuje objektový návrh knihovny. Celá knihovna, která je dále označována v této práci jako knihovna *TXML*, je rozdělena do několika balíčků. Zjednodušený pohled na celou aplikaci je znázorněn na obrázku 9.3.



Obrázek 9.3: Diagram balíčků.

Tento návrh vychází z návrhu rozhraní TXML knihovny. To umožňuje uživateli pracovat s XML dokumenty jak pomocí interpretovaného jazyka popsaného v části 9.3, tak pomocí rozhraní třídy *TXml*.

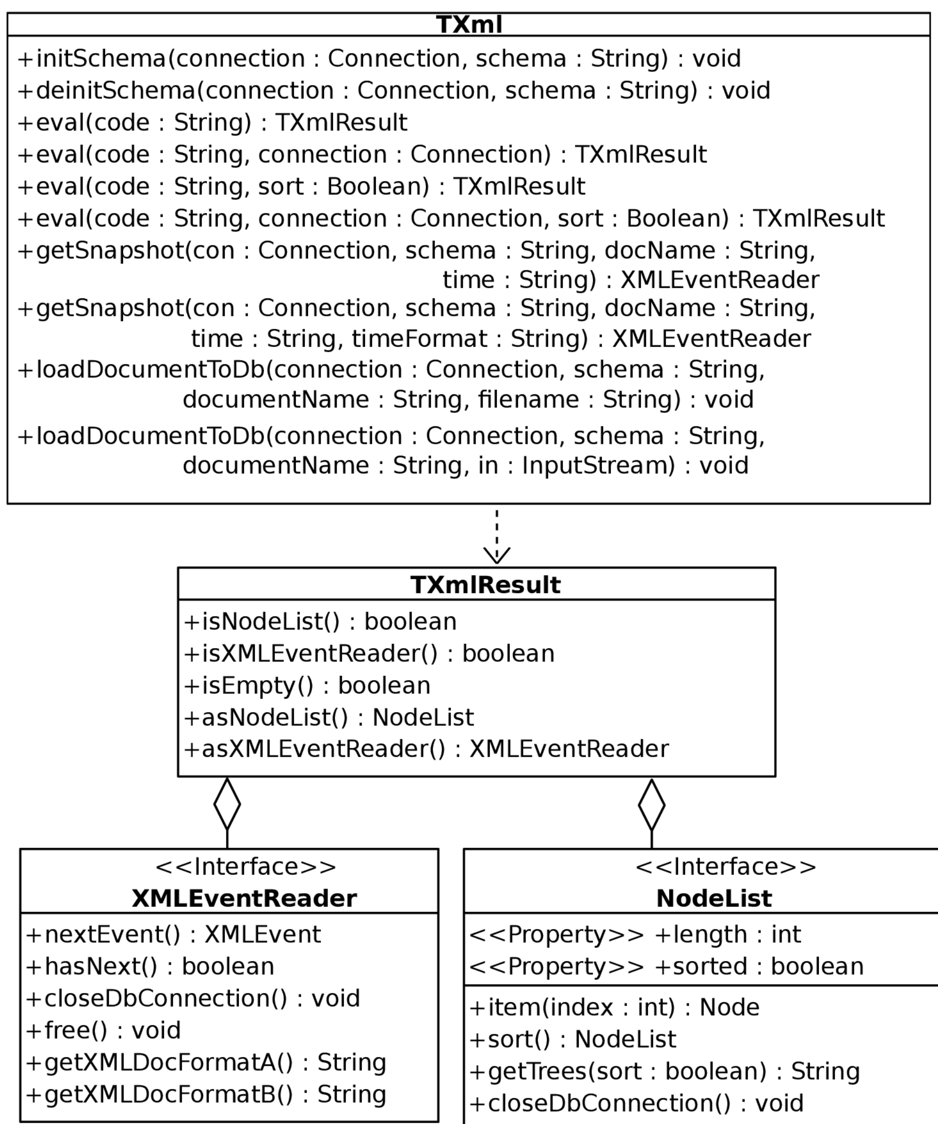
Na obrázku 9.3 je na nejvyšší úrovni balíček *parser*. Jeho vstupem je zdrojový kód (například v jazyce XPath), který parsuje a převádí na pseudoinstrukce. Tyto pseudoinstrukce jsou předány balíčku *interpreter*. Zde je použit návrhový vzor *Command* [9]. Jednotlivé instrukce jsou převedeny na základě jejich názvu na příkazy v balíčku *command*, které jsou následně vykonány nad tabulkou symbolů. Tyto příkazy ovládají celou knihovnu, což je na obrázku 9.3 znázorněno pomocí závislostí mezi balíčky. Příkaz může provést následující operace.

- vyhodnocení kroku XPath výrazu - závislost na balíčku *xpath*
- vytvoření snímku dokumentu - závislost na balíčku *snapshot*

- načtení dokumentu - závislost na balíčcích *load* a *database*, kdy příkaz pomocí balíčku *load* načte dokument a výsledný načtený model předá balíčku *database*, což popisuje závislost mezi balíčky *database* a *load*.

Každý balíček obsahuje navíc balíček *model*. V tom jsou třídy, jenž jsou hlavními nositeli dat rodičovského balíčku.

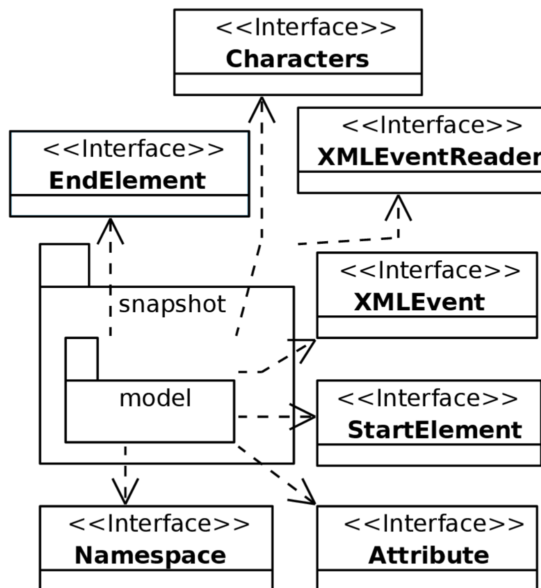
Hlavní třída rozhraní knihovny je popsána na obrázku 9.4. Návratovou hodnotou metod této třídy může být objekt pro proudové čtení snímku, seznam uzlů, jedná-li se o výsledek XPath výrazu, nebo prázdný objekt, jedná-li se o změnu dokumentu. Kromě snímků a interpretace dotazu či příkazu třída *TXml* umožňuje inicializovat a odstranit schéma v databázi a načíst nový XML dokument do databáze.



Obrázek 9.4: Rozhraní knihovny.

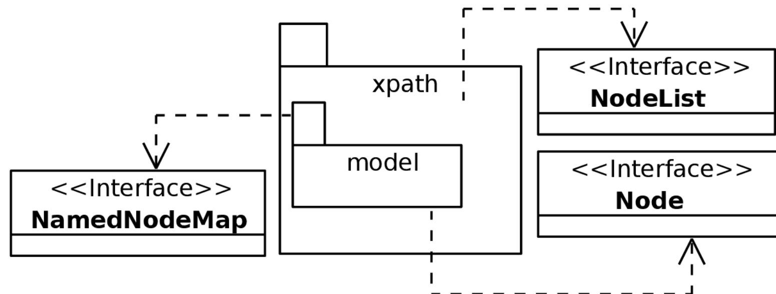
Na obrázku 9.5 je zobrazen vztah mezi balíčkem *snapshot* a rozhraním, pomocí něhož uživatel knihovny *TXML* čte snímek temporálního XML dokumentu z databáze. Návrh rozhraní je podobný rozhraním tříd v balíčku *javax.xml.stream*, které se standardně použí-

vají v jazyce Java pro načítání XML dokumentů. Dokument se zde načítá ve formě proudu událostí, které popisují jednotlivé jeho části.



Obrázek 9.5: Rozhraní pro práci se snímky temporálního XML dokumentu.

Obrázek 9.6 ukazuje vztah mezi balíčkem *xpath* a rozhraními pro zpracování výsledku XPath dotazu, která jsou uživateli k dispozici.

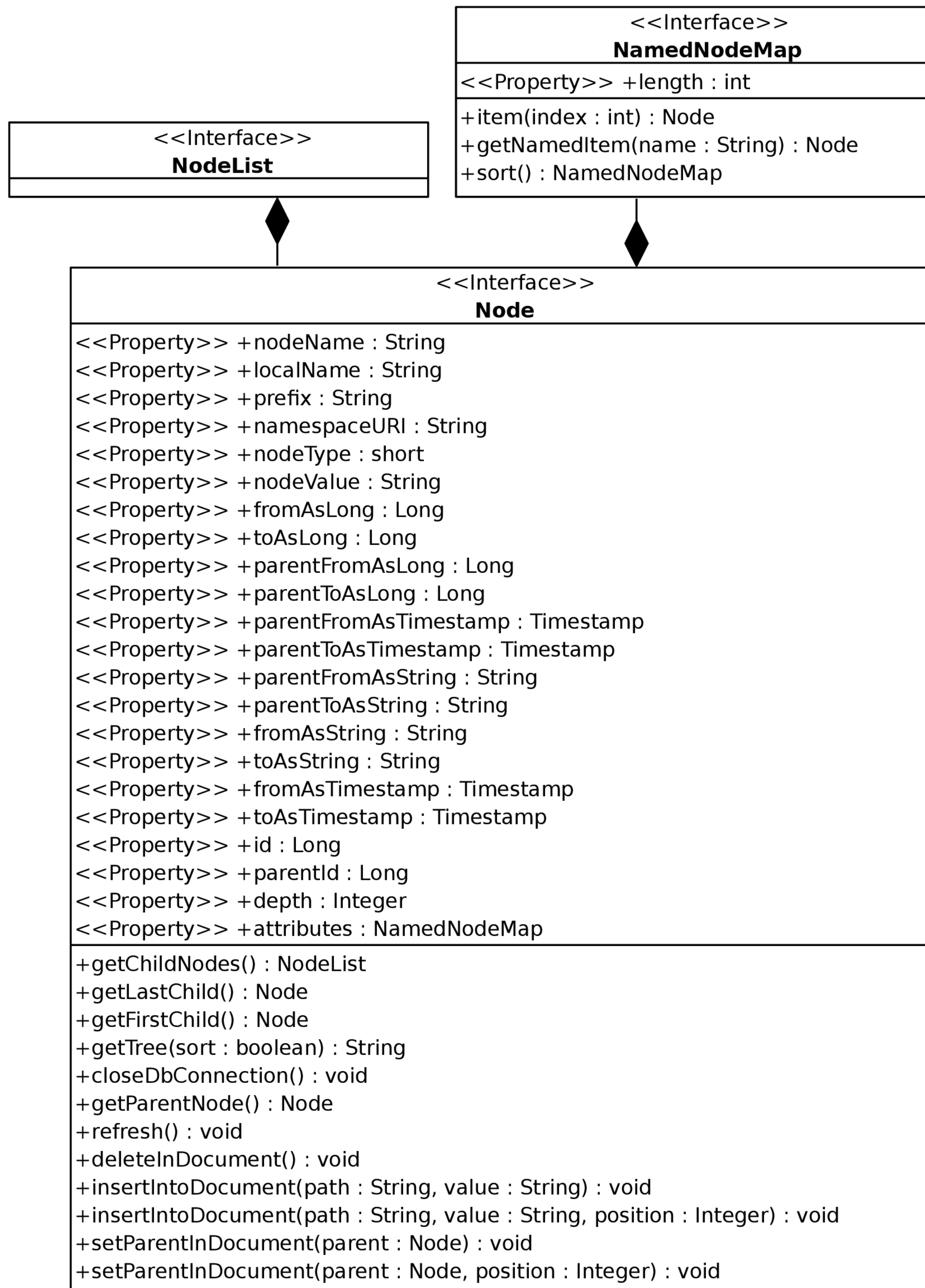


Obrázek 9.6: Rozhraní pro zpracování výsledku XPath dotazu.

Podrobněji jsou tato rozhraní popsána na obrázku 9.7. Ústředním rozhraním je zde rozhraní *Node*. Jeho návrh vychází z rozhraní *org.w3c.dom.Node*, jenž se běžně v jazyce Java používá s XPath dotazy. Rozšířeno bylo o temporální položky a navíc umožňuje provádět nad výsledkem dotazu následující operace.

- temporální smazání tohoto uzlu z dokumentu
- vložení nového elementového, atributového a textového uzlu do podstromu tohoto uzlu na určitou pozici
- temporální přesun uzlu pod nový rodičovský uzel
- textové vypsání podstromu uzlu včetně historie změn

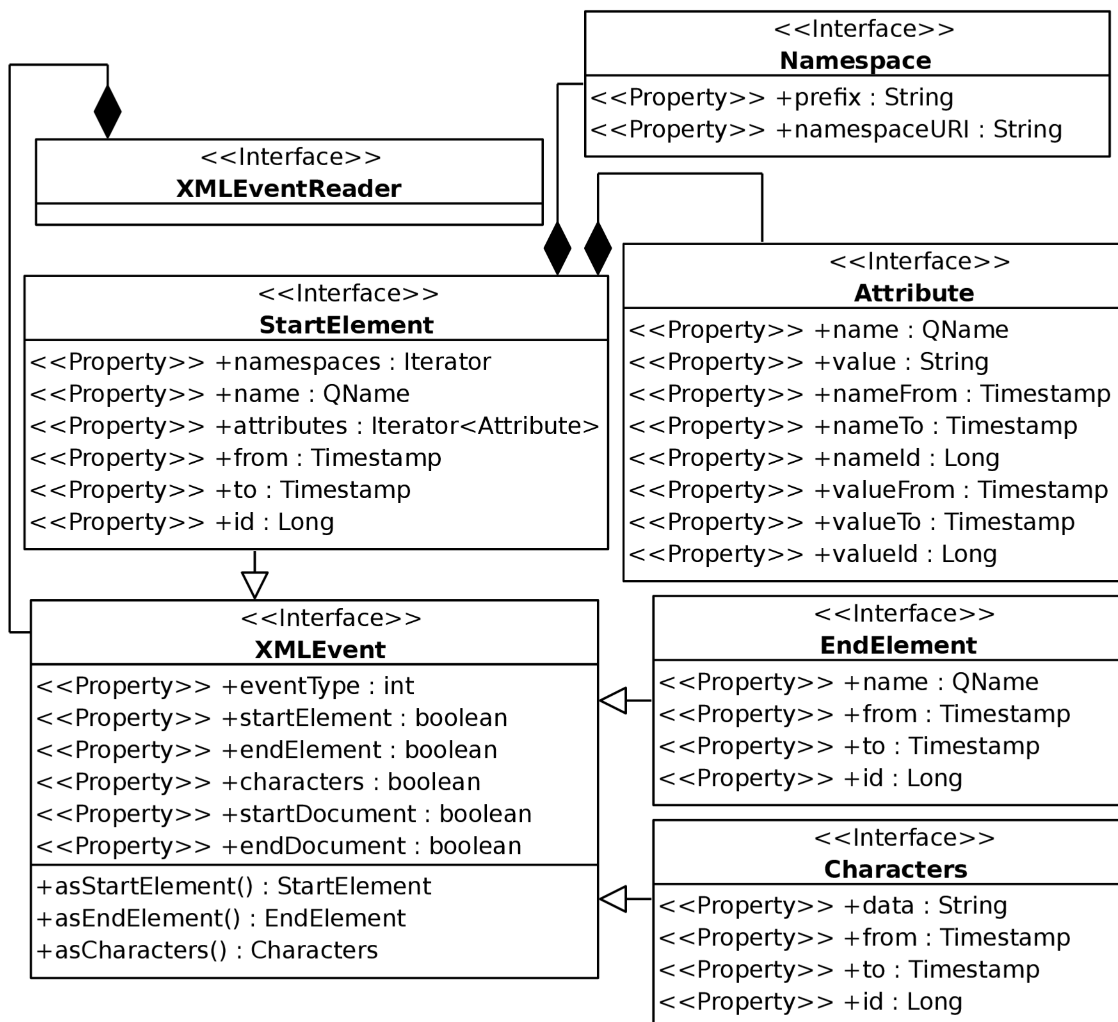
- uzavření použitého databázového spojení
- znovu načtení uzlu z databáze (metoda *refresh*)



Obrázek 9.7: Výsledek XPath dotazu.

Rozhraní *NamedNodeMap* je použito pro práci s atributy elementu. Rozhraní *NodeList* pak navíc obsahuje operace pro řazení uzlů a textové vypsání všech podstromů včetně historie změn.

Poslední část rozhraní knihovny *TXML* je zobrazena na obrázku 9.8. Jak už bylo napsáno, jedná se o podmnožinu rozhraní tříd z balíčku *javax.xml.stream* stejně jako u XPath uzlů rozšířenou o temporální položky. Navíc je třeba ještě upozornit na položku *id*. Jedná se o jednoznačný identifikátor uzlu, který je stejný pro celou jeho historii změn a který lze používat i v dotazech.



Obrázek 9.8: Zpracování snímku XML dokumentu.

9.3 Návrh gramatiky jazyka

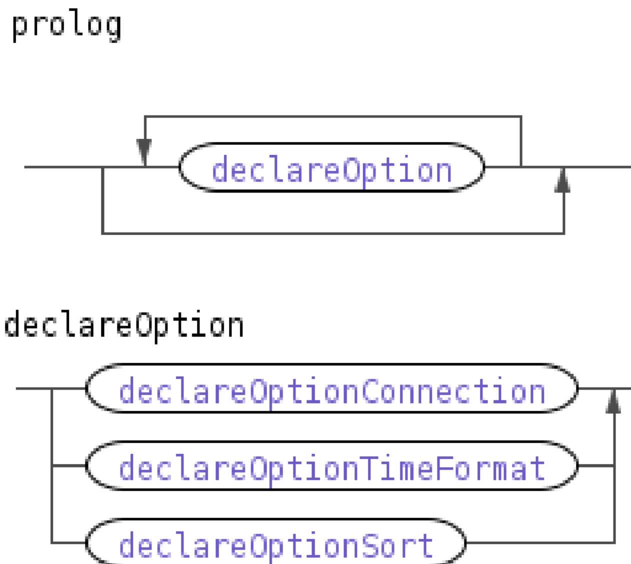
Tato část rozšiřuje a doplňuje jazyk popsaný v kapitole 7. Konkrétně se jedná o jazyk pro dotazování nad temporálním XML dokumentem a pro jeho modifikace. Z důvodu přehlednosti bude popsán celý jazyk po částech a doplněn o grafickou vizualizaci pomocí diagramu syntaxe. Také je zde použita BNF. U té je užita specifická notace pro terminální a neterminální symboly, kde terminální symbol začíná velkým písmenem a neterminální symbol

malým písmenem. Jazyk není popsán celý úplně do detailu, nicméně jeho definici lze dohledat jako součást zdrojových kódů.

Na nejvyšší úrovni jazyka je počáteční neterminál *main*. Jak ukazuje obrázek 9.9, jazyk se skládá z části *prolog* a *body* podobně jako jazyk *XQuery*. Interpretr jazyka se ukončí, jakmile narazí na konec souboru. V případě sémantické chyby je generována výjimka *TXmlException*.

Obrázek 9.9: Rozdělení zdrojového kódu.

V části prolog se deklarují volitelná nastavení, konkrétně připojení pro databázi, formát času a řazení. Tuto fázi jazyka zobrazuje obrázek 9.10. Jakmile se přejde do části *body*, tato nastavení již není možné měnit.



Obrázek 9.10: Nastavení prostředí.

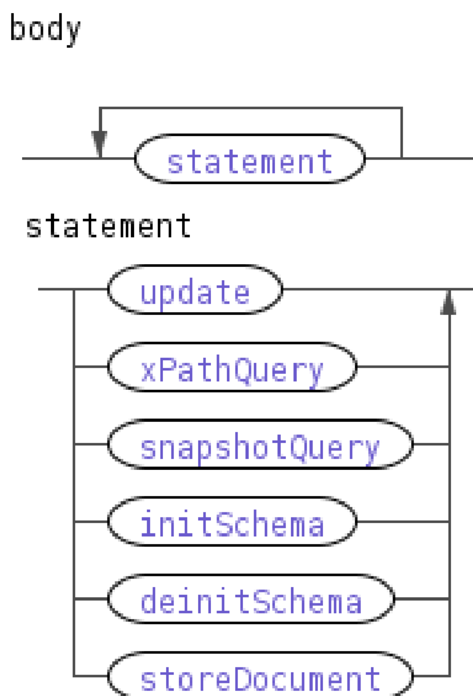
Gramatika pro deklarace je konkrétněji popsána pomocí BNF v popisu jazyka 9.1. Z důvodu malého množství možných deklarací jsou tyto deklarace napevno zakomponovány v gramatice.

```
declareOptionConnection
  ::=  'declare' 'option' 'txml:connection' connectionUrl ';'
declareOptionTimeFormat
  ::=  'declare' 'option' 'txml:time-format' timeFormat ';'
declareOptionSort
  ::= 'declare' 'option' 'txml:sort' sort ';'
```

Popis jazyka 9.1: Volitelná nastavení.

Obrázek 9.11 popisuje hlavní část celého jazyka. Ta je uvozená neterminálem s názvem *body*. Zde je neomezeně dlouhá sekvence neterminálů *statement*. Některé z nich se mohou překrývat v případě, že jde o dotazy. Poté platí, že se provedou všechny, ale je navrácen pouze výsledek posledního z nich. Neterminál *statement* může zastupovat následující příkazy.

- cyklus provádějící změnu v XML dokumentu
- XPath dotaz
- požadavek na vrácení snímku dokumentu
- inicializaci schématu včetně tabulek
- smazání schématu
- načtení XML dokumentu ze souboru do databáze



Obrázek 9.11: Tělo jazyka.

Gramatika pro snímek dokumentu, inicializaci a deinicializaci schématu a pro vložení dokumentu do databáze je poměrně jednoduchá. Konkrétně se jedná o jednořádkové příkazy, jak ukazuje popis jazyka 9.2.

```
initSchema     ::=   'txml:init-schema' '(' schemaName ')'
deinitSchema   ::=   'txml:deinit-schema' '(' schemaName ')'
storeDocument  ::=   storeDocumentWithName
                |    storeDocumentWithoutName
```

```
storeDocumentWithName
                  ::=  'txml:store' '(' schemaName ','
                       fileFullName ',' documentName ')'
storeDocumentWithoutName
                  ::=  'txml:store' '(' schemaName ','
                       fileFullName ')'
snapshotQuery ::=  'txml:doc-snapshot' '(' schemaName ','
   documentName ',' time ')'
```

Popis jazyka 9.2: Základní příkazy.

Aktualizace dokumentu jsou z důvodu obecnosti vždy prováděny pomocí cyklu. Cyklus může obsahovat jednu nebo více aktualizací a je tu i několik volitelných položek. Tato gramatika je popsána v popisu jazyka 9.3.

```
update        ::=   forClause updateExpr
               |    forClause '{' updateExpr+ '}'
forClause     ::=   'for' variable 'in' absPathExprWithDoc
updateExpr    ::=   insertExpr
               |    deleteExpr
               |    parentExpr
insertExpr    ::=   'insert' insertXPathExpression ('value'
   value)? ('position' insert_pos)?
deleteExpr    ::=   'delete' 'node' childXPathExpression
parentExpr    ::=   'set' 'parent' parentXPathExpression ('
   position' insert_pos)?
parentXPathExpression
              ::=   variable 'as' steps
childXPathExpression
              ::=   variable steps
simpleXPathExpression
              ::=   ('/' elementName)* ('/' attributeName)?
insertXPathExpression
              ::=   variable simpleXPathExpression
absPathExpr   ::=   steps
doc           ::=   'txml:doc' '(' schemaName ',' documentName
   ')'
absPathExprWithDoc
              ::=   doc  absPathExpr
```

Popis jazyka 9.3: Gramatika pro aktualizaci dokumentu.

Gramatika pro XPath výrazy je z časových důvodů obecně zjednodušená a samozřejmě byla rozšířená o temporální operátory a název schématu. Tuto gramatiku popisuje popis jazyka 9.4.

```
xPathQuery      ::=  absPathExprWithDoc
absPathExprWithDoc
                ::=  doc  absPathExpr
doc             ::=  'txml:doc' '(' schemaName ',' documentName
   ')'
```

```
absPathExpr    ::=  steps
steps          ::=  (directStep|undirectStep) steps*
undirectStep   ::=  '//' nodeGenerator predicate*
directStep     ::=  '/' nodeGenerator predicate*
predicate      ::=  '[' expr ']'
expr           ::=  xPathInExpr operator value
                |   value operator xPathInExpr
                |   position
xPathInExpr    ::=  nodeGenerator predicate* steps?
position       ::=  integerNumber
                |   'last()'
operator       ::=  '='
                |   '!='
                |   '<'
                |   '>'
                |   '<='
                |   '>='
                |   'PRECEDES'
                |   'FOLLOWS'
                |   'MEETS' | 'LMEETS'
                |   'RMEETS'
                |   'OVERLAPS'
                |   'CONTAINS'
                |   'IN'

nodeGenerator  ::=  elementName
                |   attributeName
                |   'text()'
                |   '.'
                |   '..'
                |   '*'
elementName    ::=  (Name ':')? Name
attributeName  ::=  ('@' | 'attribute' ':') Name
                |   ('@' | 'attribute' ':') Name ':' Name
```

Popis jazyka 9.4: Gramatika pro XPath dotazy.

9.4 Sémantika jazyka

Abychom popsali také sémantiku, jednotlivé části jazyka a jeho možnosti demonstrujeme v následujících sekcích na příkladech. Deklarace, snímky a inicializace a deinicializace schémat zde popsány nejsou, neboť ty jsou primitivní.

9.4.1 Vložení nového uzlu

Příklad 9.5 ukazuje vložení elementového uzlu *available* s hodnotou *10* na první pozici v podstromu elementu *book*, v kterém se nachází jméno autora zadané ve filtru. Název schématu je *txml* a název dokumentu je *doc.xml*.

```
for $n in txml:doc("txml", "doc.xml")//book[author = "Cach"]
{
  insert $n/available value "10" position "1"
}
```

Příklad 9.5: Ukázka vložení nového elementu s hodnotou.

Vložení uzlu na určitou pozici zadanou číselnou hodnotou nemusí být pro uživatele příjemné. Jako rozšíření by bylo vhodné cílovou pozici fixovat na jiný existující element.

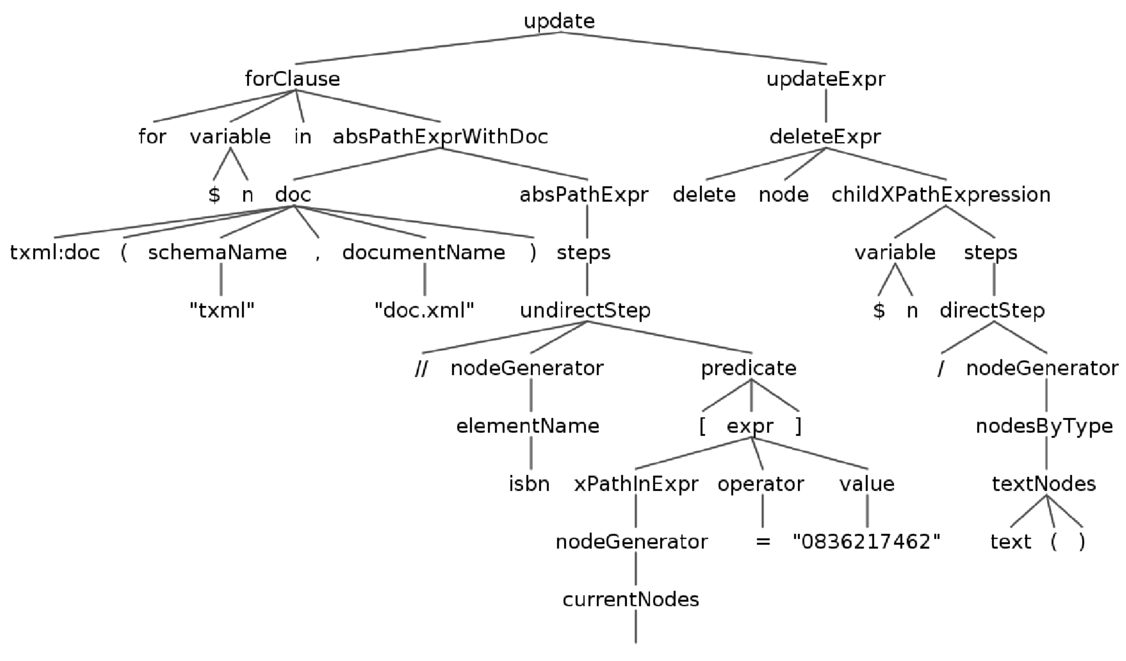
9.4.2 Odstranění uzlu

Cyklus v příkladu 9.6 odstraní z dokumentu *doc.xml* ve schématu *txml* textový uzel se zadanou hodnotou *isbn*.

```
for $n in txml:doc("txml", "doc.xml")//isbn[. = "0836217462"]
  delete node $n/text()
```

Příklad 9.6: Odstranění hodnoty.

Obrázek 9.12 ukazuje syntaktický strom cyklu pro smazání textového uzlu. V tomto stromě je také zobrazeno parsování XPath výrazu.



Obrázek 9.12: Syntaktický strom pro odstranění hodnoty.

9.4.3 Změna umístění elementu nebo atributu

Příklad 9.7 ukazuje přesun autora jménem *Linn* pod titul *Learning XML*. Po přesunu bude element *author* na druhé pozici v podstromě knihy

```
for $n in txml:doc("txml", "books.xml")//author[name = "Linn"]
{
  SET PARENT $n AS //book[name = "Learning XML"] POSITION "2"
}
```

Příklad 9.7: Přesun elementu.

9.4.4 Temporální XPath dotaz

Zde se nabízí kromě atributů *txml:from* a *txml:to*, s kterými lze pracovat v XPath jako s normálními atributy, poměrně velká množina temporálních operátorů. Mimo těch dobře známých z relačních databází, jenž jsou popsány na obrázku 6.3, byly doplněny následující operátory: *FOLLOWS*, *RMEETS* a *IN*. Tyto operátory nějak nezvyšují vyjadřovací sílu a jsou pouze syntaktickým cukrem, což vyplývá z jejich definice, která je následující.

$$a\ FOLLOWS\ b = b\ PRECEDES\ a$$

$$a\ RMEETS\ b = b\ MEETS\ a$$

$$a\ IN\ b = b\ CONTAINS\ a$$

Temporální dotaz v příkladu 9.8 vrací knihy, které jsou v databázi platné pouze v roce 2014.

```
declare option txml:time-format "yyyy";
txml:doc("txml", "books.xml")//book[. IN "<2014; 2015>"]
```

Příklad 9.8: Temporální dotaz.

Kapitola 10

Implementace

Cílem této kapitoly je popsat klíčové a problémové části implementace. V některých případech je zde uvedeno i navržené rozšíření do budoucna. Logicky tato kapitola navazuje na kapitolu 9, takže popisovaná implementace nemusí být použitelná v dalších možných rozšířeních knihovny *TXML*.

Jako jazyk byl pro implementaci vybrán jazyk Java a to ze dvou důvodů. V první řadě je to přenositelnost tohoto jazyka, která je velice výhodná ve spojení s databázemi. Tato knihovna tak může být použitelná dokonce i na mobilních zařízeních. Druhým důvodem je velká komunita programátorů, což by mělo zvýšit pravděpodobnost, že někdo aplikaci využije. Zajímavé by taktéž mohlo být použití s Java EE.

10.1 Filtry

Filtrování uzlů ve výrazu XPath je provedeno na základě predikátu, v němž na jedné straně vystupuje hodnota a na druhé straně XPath cesta. Z pohledu implementace to znamená, že nejprve se vyhodnotí cesta, následně se výsledné uzly cesty zamění za jejich hodnoty, nad nimi se aplikuje filtr a nakonec se u zbývajících uzlů provede návrat k původním uzlům. Tento návrat je jednoduše implementován tak, že se předává při každém kroku XPath výrazu novému uzlu původní uzel. Ten se na konci jednoduše použije jako výsledek po filtrování. Aby bylo možné filtry neomezeně zanořovat, předává se namísto jednoho uzlu zásobník uzlů.

10.2 Referenční integrita

Operace smazání uzlu a přesun uzlu jsou problémové, neboť nepracují pouze s jediným uzlem. Při smazání uzlu je temporálně odstraněn celý jeho podstrom, a protože je součástí primárního klíče i interval platnosti uzlu, který je modifikován, musí mazání podstromu v databázi probíhat od uzlů v největší hloubce nahoru. Proto bylo nutné implementovat příslušnou metodu rekurzivně.

U přesunu uzlu je problém ještě složitější, neboť je nutné zajistit, že se po přesunu podstromu nezmění vzájemné pořadí všech jeho uzlů. Implementace je rozdělena do následujících fází.

1. Pomocí průchodu preorder se načte do seznamu celý podstrom.

2. Uzly se seřadí nejprve podle dokumentového pořadí, poté podle jejich hloubky. Zde se využívá toho, že je řadící algoritmus v jazyce Java stabilní [8].
3. Seznam uzlů se v databázi temporálně smaže (uzly se mažou od konce).
4. Seznam uzlů se do databáze vloží s aktuálním časem začátku intervalu platnosti.

10.3 Přenositelnost mezi databázemi

Přestože existuje standard SQL a použité příkazy jsou vesměs triviální, objevilo se zde několik problémů, jenž způsobily v některých případech nefunkčnost s určitými databázovými systémy. Například použitá databáze *H2* se liší v DDL u deklarace pole. Databáze *MySql* zase nepodporuje schémata a databáze *PostgreSQL* jiným způsobem pracuje se sekvencemi. Tyto odlišnosti byly zanalyzovány a pro tyto databáze ošetřeny. Přenositelnost je zaručena pro databáze *H2* a *PostgreSQL*.

10.4 Podmínky konzistence

První podmínka konzistence je zajištěna na úrovni databáze pomocí sql klauzule 10.1 nad tabulkou *lcp*.

```
CHECK (parentFrom <= "from" AND to <= parentTo)
```

Sql klauzule 10.1: První podmínka konzistence dokumentu.

Druhá podmínka konzistence je implementována na úrovni Javy automaticky jako důsledek použití transakčního času. Pro zajištění třetí podmínky je explicitně proveden dotaz, jenž kontroluje při přesunu uzlu, zda nový rodičovský uzel není současně potomkem uzlu, který se přesouvá. Čtvrtá, pátá a šestá podmínka nejsou implementovány a jsou ponechány jako rozšíření.

10.5 Jmenné prostory

U jmenných prostorů bylo implementováno omezení, které dovoluje deklarovat jmenné prostory pouze u kořenového elementu. To zjednodušuje operace vkládání, mazání a přesun elementu, neboť se nemůže v rámci jednoho dokumentu vyskytovat více výchozích jmenných prostorů a v rámci každého prefixu existuje pouze jedna URI, jenž je k němu přiřazená v celém dokumentu.

Problémem bylo zajistit, že se po těchto operacích nezmění jmenný prostor žádného uzlu v dokumentu, případně, že se vkládané uzly vloží se správným jmenným prostorem. Toho by bylo možné dosáhnout například tak, že se explicitně po přesunu uzlu u něj deklaruje jeho jmenný prostor. Tato implementace je ponechána jako možné rozšíření.

Dalším problémem mohou být atributy z jmenného prostoru *xml*, ty by po přesunu uzlu například pro atribut *xml:lang* měly být také explicitně deklarované.

10.6 Textové uzly

Obrázek 3.1 ukazuje příklad, kdy jeden elementový uzel může obsahovat více textových uzlů. Tabulky představené v kapitole 7 jsou navržené tak, že každý elementový uzel má

přiřazenou maximálně jednu hodnotu. Proto bylo provedeno rozšíření, kdy byla hodnota povýšena na samostatný uzel s vlastním identifikátorem a platností.

10.7 Aktualizace

Změna tagu uzlu není dovolena. Taková operace je složitá, protože změní kromě názvu tagu i třídu uzlu a potencionálně třídu i všech jeho následovníků. Proto je ponechána jako rozšíření.

Změnu textového uzlu lze provést vzhledem k 10.6 kombinací operací smazání a vložení uzlu. Jako rozšíření je ponechána lepší implementace.

10.8 Řazení

V databázi je uloženo pořadí uzlů v polích *valid*. Tyto pole obsahují pořadí uzlů ve stejné hloubce. Nutno dodat, že součásti pořadí je i interval platnosti. Jinak řečeno je možné provést operaci, která mění vzájemné pořadí uzlů, a tato změna bude temporálně uložena. Po každé operaci měnící dokument se tato pole duplikují pro modifikované uzly.

Cílem operace řazení je použít známé pořadí uzlů v určité hloubce a určitém časovém intervalu k seřazení uzlů s potenciálně rozdílnou platností a hloubkou. Implementovaný algoritmus pro porovnání uzlů funguje následovně.

1. Pokud se intervaly platnosti uzlů překrývají a jsou ve stejné hloubce, provede se průnik intervalů, pro ten se vyhledá pole *valid* a podle něj se uzly porovnají. To, že pole existuje pro daný průnik intervalů, je zaručeno podmínkami konzistence a způsobem, jakým se pracuje s transakčním časem a duplikují pole *valid*.

2. Pokud se intervaly platnosti uzlů překrývají a jsou v různé hloubce, porovná se jeden z uzlů s předchůdcem druhého uzlu ve stejné hloubce. Je-li mezi uzly vztah předchůdce-následník, tak se vyskytuje v dokumentu dříve uzel v menší hloubce.

3. Pokud jsou intervaly platnosti disjunktní a hloubka uzlů není stejná, porovnají se uzly podle hloubky.

4. Pokud jsou intervaly platnosti disjunktní a hloubka uzlů je stejná, porovnají se uzly podle pořadí ve svých polích *valid*. Je-li stejné, porovnají se podle začátku intervalu platnosti.

Popsaný algoritmus je kompromisem mezi vysokou výpočetní náročností a kvalitou řazení uzlů. Pro snížení počtu databázových dotazů je implementovaná navíc cache. Algoritmus dokáže dobře seřadit uzly, u kterých bylo pouze prohozeno pořadí.

10.9 Validace

Validace není implementovaná a je ponechána jako rozšíření. Pro její naivní implementaci by bylo možné využít faktu, že změny v dokumentu mohou být provedeny pouze u aktuálních XML uzlů (konec intervalu platnosti je *Now*). Stačilo by tedy, kdyby se po každé změně v dokumentu provedla validace jeho aktuálního snímku pomocí knihovny třetí strany. V případě nevalidního snímku lze poté jednoduše provést rollback databáze.

10.10 Zamykání tabulek

Každá operace modifikující XML dokument má dvě fáze. V první fázi se provede XPath dotaz. Ten obecně vybírá z databáze uzly, jenž se budou měnit. V druhé fázi je poté aplikována konkrétní operace. Nutno dodat, že obě fáze se zpracovávají na úrovni Javy. Může tak dojít k situaci, kdy data získaná v první fázi již nejsou platná ve fázi druhé. Pro ošetření takových situací je nutné v databázových dotazech uvést klauzuli *for update*. To ovšem může být zase spojeno s rizikem uváznutí (deadlock). Implementace zamykání řádků tabulek je ponechána jako rozšíření. [2]

Kapitola 11

Testování

Testování je rozděleno do dvou fází. V první fázi je testována správnost výstupu. Druhá fáze je zaměřena na výkonnostní testování a snaží se identifikovat výpočetně a paměťově nejnáročnější části programu. Také je zde provedeno porovnání na netemporální úrovni s konkurenčním produktem eXist-db.

Testy popisují závislost doby provedení operace na velikosti dokumentu. Aby byl graf přehlednější, měří se z osy X pouze každá stá hodnota. Zde testované operace jsou vložení nového elementu, smazání elementu, přesun elementu na jiné místo v dokumentu, řazení elementů a tři XPath dotazy. Testy byly provedeny nad databází PostgreSQL 9.3.

11.1 Ověření správnosti výstupu

Pro ověření správnosti výstupu bylo napsáno 38 jednotkových testů. Ty lze rozdělit do následujícím skupin. Některé testy patří do více skupin.

- XPath filtr - 11 testů
- XPath výrazy pracující s uzly předchůdců - 6 testů
- XPath výraz pro získání kořenového uzlu - 1 test
- XPath výrazy používající nepřímý krok - 7 testů
- XPath výrazy pracující s textovými uzly - 3 testy
- XPath výrazy pro libovolný elementový a textový uzel - 5 testů
- XPath výrazy pracující s vnitřními uzly knihovny TXML - 2 testy
- operace smazání uzlu - 2 testy
- operace vložení uzlu - 3 testy
- operace změna rodičovského uzlu - 8 testů
- získání snímku dokumentu z databáze - 5 testů

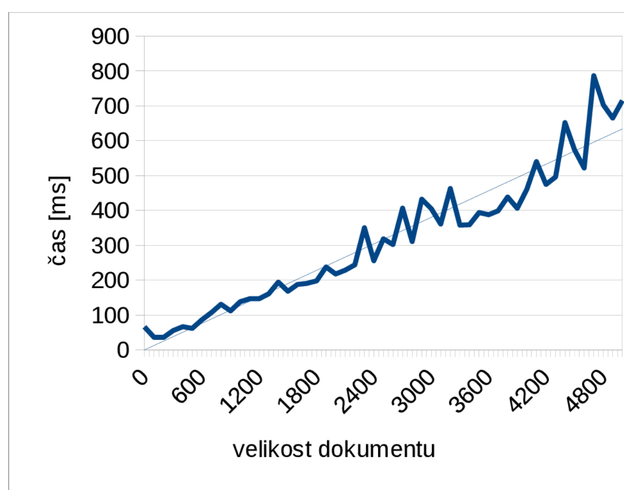
11.2 Vložení nového elementu

Na původně prázdný dokument je opakovaně aplikována operace vkládání. Ta vloží do dokumentu obsah uvedený v testovacím vzorku 11.1, kde za X je doplněno pořadové číslo vzorku.

```
<book>
  <name>
    name X
  </name>
  <author>
    <forename>
      forename X
    <forename>
    <surname>
      surname X
    </surname>
  </author>
</book>
```

Testovací vzorek 11.1: Testování vkládání.

Výsledek testu je zobrazen na obrázku 11.1. Na ose X je pořadové číslo vkládaného vzorku a na ose Y doba v milisekundách potřebná pro jeho vložení. Navíc je zde regresní přímka. V zobrazeném intervalu na ose X se jedná o lineární závislost. Velikost 5000 vzorků odpovídá XML dokumentu o velikosti 976,7 kB.

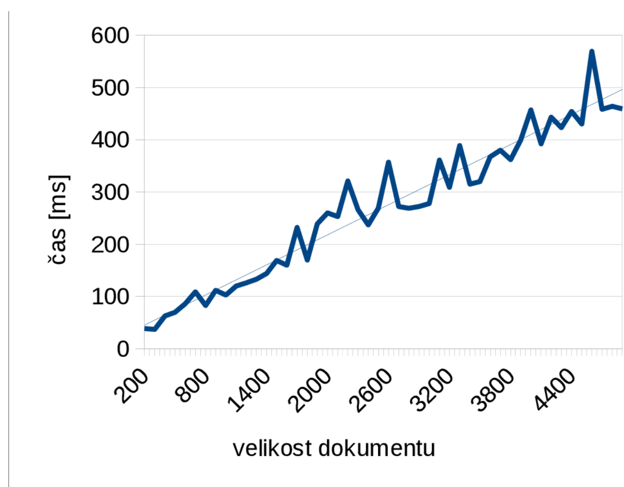


Obrázek 11.1: Výsledek měření času vložení nového elementu.

11.3 Smazání elementu

Test funguje na stejném principu jako test v části 11.2 s tím rozdílem, že se po každém stém vložení smaže element *book* nacházející se v dokumentu přibližně o sto elementů *book* dříve. Doba operace smazání elementu *book* je zobrazena na obrázku 11.2. Na ose X je

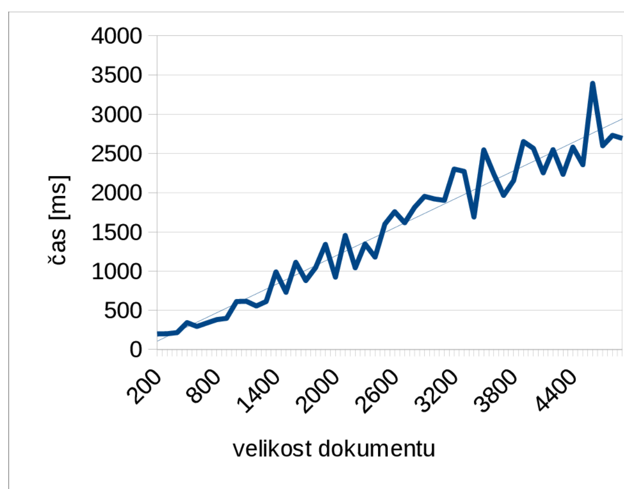
uvedeno, kolik vzorků bylo v dokumentu, když proběhlo mazání. Z grafu je patrné, že smazání elementu je jenom o něco málo rychlejší než jeho vložení.



Obrázek 11.2: Výsledek měření času smazání elementu.

11.4 Přesun elementu

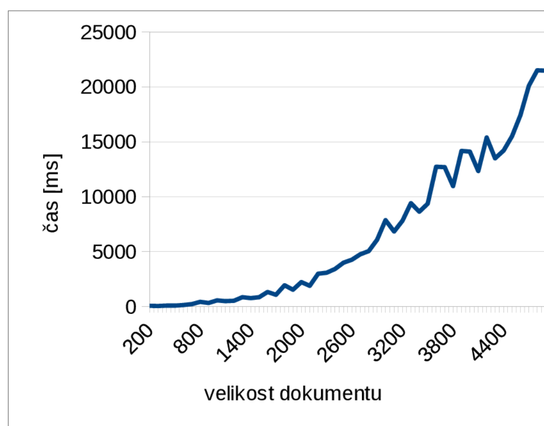
Test funguje na stejném principu jako test v části 11.3 s tím rozdílem, že se po každém stém vložení přesune element *author* nacházející se v dokumentu přibližně o padesát elementů dříve pod element *book*, který se nachází přibližně dalších 50 elementů *book* před ním. Doba operace přesunu elementu *book* je zobrazena na obrázku 11.3. Na ose X je opět uvedeno, kolik vzorků bylo v dokumentu, když došlo k přesunu. Z grafu je patrné, že přesun elementu je výpočetně náročnější.



Obrázek 11.3: Výsledek měření času přesunu elementu.

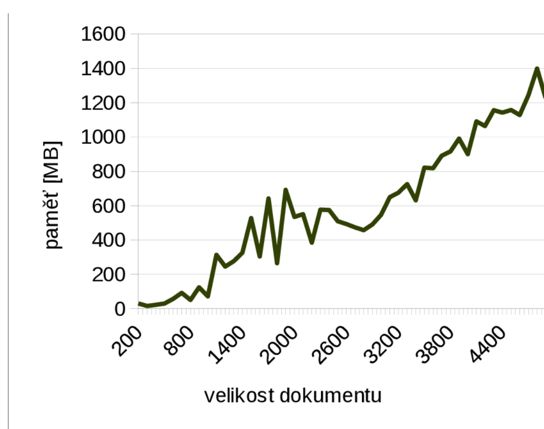
11.5 Řazení elementů

Testováno je řazení elementů *book*, ty jsou vytvořené ze vzorku 11.1. Navíc je v každé stovce těchto elementů jeden smazán, aby bylo testováno i řazení elementů, jejichž intervaly platnosti jsou disjunktní, a sleduje se i paměťová náročnost. Obrázek 11.4 ukazuje výpočetní náročnost řazení. Na ose X je uveden počet řazených elementů *book*.



Obrázek 11.4: Výsledek měření času řazení elementů.

Paměťovou náročnost řazení těchto elementů ukazuje obrázek 11.5. Takto velké množství spotřebované paměti je způsobeno použitím datové struktury *HashMap*.



Obrázek 11.5: Výsledek měření paměťové náročnosti řazení elementů.

11.6 XPath dotazy

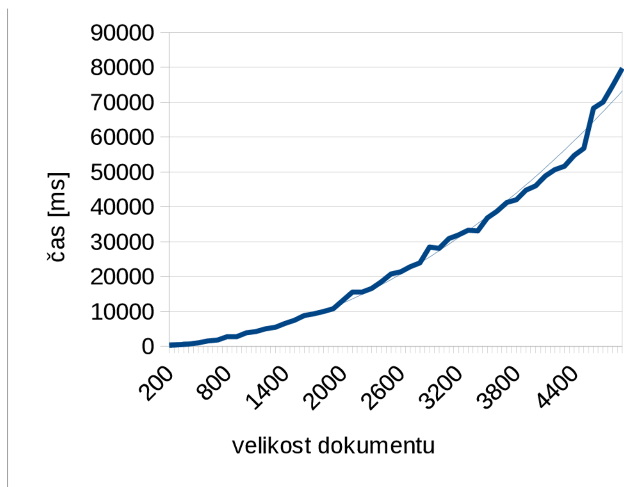
Tento test zkoumá časovou náročnost XPath dotazů. Test je prováděn nad dokumentem s přibývajícím počtem vzorků 11.1, kde na každých sto vzorků připadá jeden smazaný vzorek.

První testovací dotaz je ukázán v dotazu 11.2, kde číslo S odpovídá vždy polovině aktuálního počtu vzorků. V tomto případě jsou zkoumány časové nároky na dotaz sestavený pouze z přímých kroků.

```
/books/book/author[surname = "surname S"]
```

Dotaz 11.2: Dotaz pro výkonnostní test složený z přímých kroků.

Výsledek testu zobrazuje obrázek 11.6. Z grafu je zřejmé, že zde není lineární závislost a pro větší počet elementů je doba zpracování dotazu příliš vysoká.



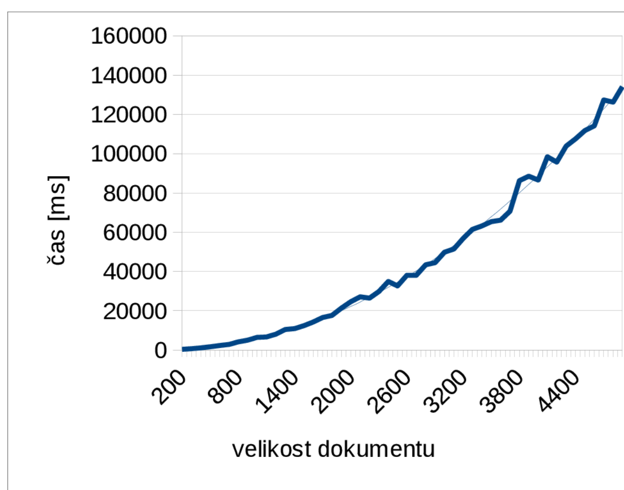
Obrázek 11.6: Výsledek měření doby XPath dotazu s přímými kroky.

Druhý testovaný XPath dotaz, popsaný jako dotaz 11.3, obsahuje nepřímý krok. Ten je potencionálně výpočetně náročnější, protože se pracuje s větším množstvím uzlů. Znak *S* opět zastupuje hodnotu odpovídající polovině aktuálního počtu vzorků.

```
//author[surname = "surname S"]
```

Dotaz 11.3: Dotaz pro výkonnostní test s použitím nepřímého kroku.

Výsledek testu zachycuje graf na obrázku 11.7. Rozdíl oproti grafu 11.6 je téměř dvojnásobný.



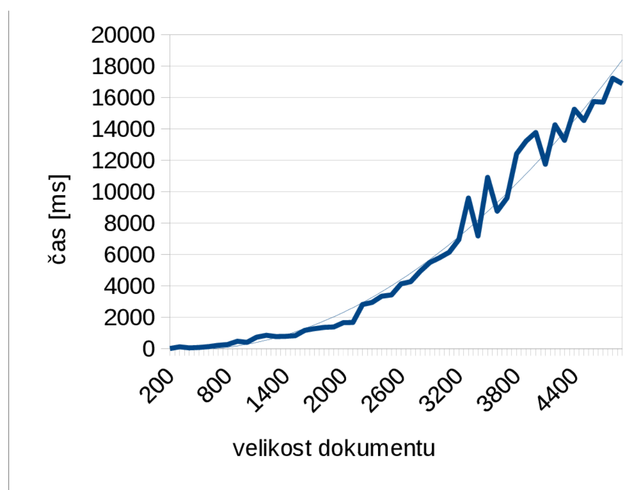
Obrázek 11.7: Výsledek měření doby XPath dotazu s nepřímými kroky.

V třetím testu je použit XPath výraz s temporálním filtrem. Výraz je zobrazen jako dotaz 11.4. Tento dotaz vybere z dokumentu v databázi všechny elementy *book*, které nejsou aktuálně již platné.

```
/books/book[txml:to != 'Now']
```

Dotaz 11.4: Dotaz pro výkonnostní test s použitím temporálního filtru.

Výsledek testu je zobrazen v grafu na obrázku 11.8. Dotaz pracuje s menším počtem uzlů, proto je výsledek testu příznivější.



Obrázek 11.8: Výsledek měření času XPath dotazu s temporálním filtrem.

11.7 Zhodnocení testů

Aby bylo možné výsledky testů zhodnotit, byly provedeny dva stejné testy nad nativní XML databází eXist-db. Tato databáze není temporální, proto byly prováděny pouze netemporální testy. Konkrétně nad XML dokumentem s 5000 vzorky 11.1 byl proveden dotaz číslo 11.2, jehož vyhodnocení trvalo 20 ms, a dotaz 11.3, u kterého byla doba vyhodnocení 24 ms. Databáze eXist-db tedy byla ve vyhodnocení více jak 4000krát rychlejší.

Pomocí profilování bylo zjištěno, že většinu výpočetního času stráví knihovna TXML u dotazu 11.2 vyhodnocením jednotlivých kroků výrazu. Zde se pro jeden krok 5000krát (pro každý element jeden) volá dotaz 11.5.

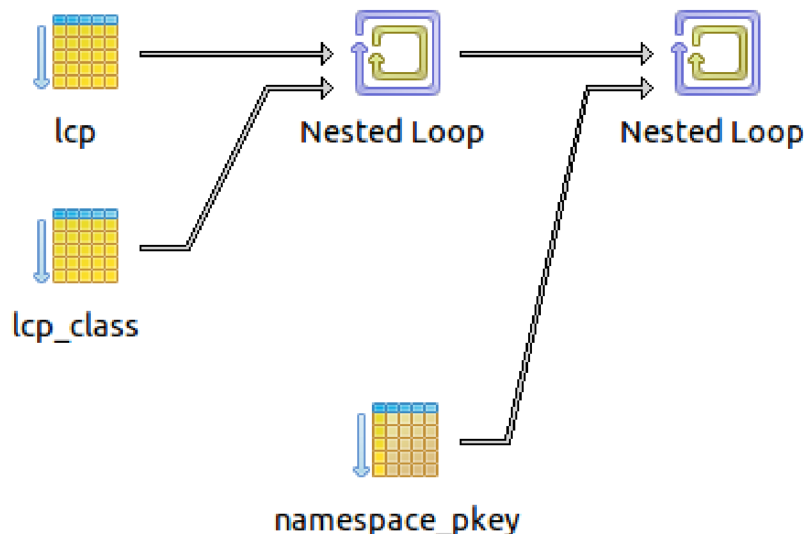
```
SELECT lcp.id as id, lcp.from as from, lcp.to as to, lcp.
   parentId as parentId, lcp.parentFrom as parentFrom, lcp.
   parentTo as parentTo, lcp.value as value, lcp_class.depth
   as depth, lcp.lcp_class as lcp_class, lcp_class.type as
   type, lcp_class.local_part as local_part, namespace.prefix
   as prefix, namespace.uri as uri, namespace.id as
   namespace_id
FROM %s.LCP as lcp, %s.LCP_CLASS as lcp_class, %s.namespace as
    namespace
WHERE namespace.id = lcp_class.namespace_id and lcp.parentId =
    ? and lcp.parentFrom = ? and lcp.parentTo = ? and lcp.
```

```
   LCP_CLASS = lcp_class.ID and lcp_class.type = ? and
   lcp_class.local_part = ? and lcp_class.namespace_id = ?;
```

Dotaz 11.5: SQL dotaz pro získání potomka uzlu.

Což zabere 26 s na jeden krok (přepočteno na jeden dotaz 5 ms). Nabízejí se tři možné optimalizace. První optimalizací by mohlo být sloučení dotazů tak, aby získal plánovač PostgreSQL větší prostor pro optimalizaci. Druhá se zaměřuje na omezení množství zpracovávaných uzlů. V tomto konkrétním případě by stačilo vyhodnocovat XPath výraz od konce namísto od začátku ([12]). Pokud by nad textovými uzly existoval databázový index, který by se použil pro vyhodnocení filtru v prvním kroku, pak by se v dalších krocích v tomto případě pracovalo pouze s jediným uzlem. Třetí navrhovanou optimalizací je provést analýzu dotazu a vytvořit na jejím základě v databázi nový index.

Z pohledu implementace je nejjednodušší varianta číslo jedna, proto je zkoumána jako první. Na obrázku 11.9 je znázorněno vyhodnocení dotazu 11.5.



Obrázek 11.9: Vyhodnocení SQL dotazu plánovačem PostgreSQL.

Celková doba vyhodnocení dotazu je 7,7 ms. Nejvíce času, konkrétně 6,3 ms, bylo stráveno filtrováním tabulky *lcp* (na obrázku vlevo nahoře), kterým bylo odstraněno 39999 řádků. Pro toto filtrování byly použity sloupce *parentTo*, *parentFrom* a *parentId*. Ty jsou v tabulce cizím klíčem, a prakticky proto zde není žádný prostor pro rychlejší vyhodnocení.

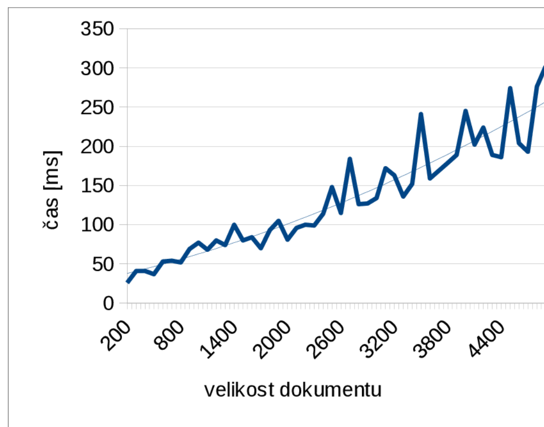
Druhá varianta zde vzhledem k časové náročnosti její implementace není uvedena. Třetí variantou je dotazy sloučit. Naivně by se to dalo provést tak, že zřetězíme obsah v klauzuli *where*. Takové řešení ale není ideální, neboť by vedlo k příliš dlouhým dotazům a některé databáze mohou mít v tomhle směru omezení. Z toho důvodu bylo provedeno naopak zobecnění dotazu. To je popsáno v dotazu 11.6.

```
SELECT lcp.id as id, lcp.from as from, lcp.to as to, lcp.
   parentId as parentId, lcp.parentFrom as parentFrom, lcp.
   parentTo as parentTo, lcp.value as value, lcp_class.depth
```

```
    as depth, lcp.lcp_class as lcp_class, lcp_class.type as
    type, lcp_class.local_part as local_part, namespace.prefix
    as prefix, namespace.uri as uri, namespace.id as
    namespace_id
FROM %s.LCP as lcp, %s.LCP_CLASS as lcp_class, %s.namespace as
     namespace where namespace.id = lcp_class.namespace_id and
    lcp.LCP_CLASS = lcp_class.ID and lcp_class.type = ? and
    lcp_class.local_part = ? and lcp_class.namespace_id = ?
    order by lcp.parentId asc, lcp.parentFrom asc
```

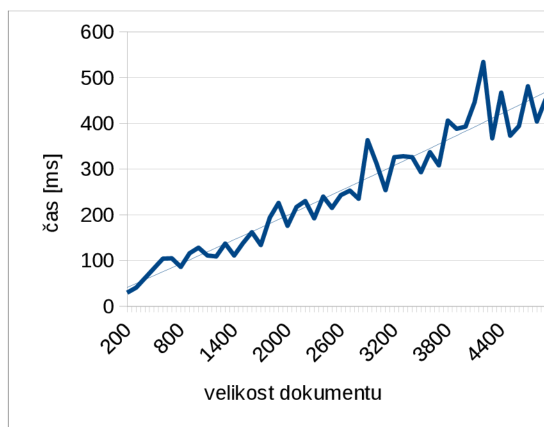
Dotaz 11.6: SQL dotaz po optimalizaci.

Dotaz již není omezen na jeden uzel a navíc v něm přibylo řazení. Namísto dotazování se databáze na potomky každého rodičovského uzlu zvlášť jsou z databáze načteny všechny možné uzly potomků. Poté je seřazen seznam rodičovských uzlů a na úrovni Javy se oba seřazené seznamy v lineárním čase projdou a platní potomci dohledají. Výsledek testu je zobrazen na obrázku 11.10.



Obrázek 11.10: Výsledek měření doby XPath dotazu s přímými kroky po optimalizaci.

Podobná optimalizace byla provedena i pro dotaz s nepřímými kroky. Její výsledek je zobrazen na obrázku 11.11.



Obrázek 11.11: Výsledek měření doby XPath dotazu s nepřímými kroky po optimalizaci.

Optimalizace výrazně pomohla. Oproti databázi *eXist-db* je ale rychlost vyhodnocení stále horší. Pro dotaz skládající se pouze s přímých kroků je to více jak desetinásobek, pro dotaz s použitím nepřímého kroku více jak dvacetinásobek. Tato optimalizace by mohla být použita i na řazení uzlů.

Kapitola 12

Závěr

V kapitole 4 bylo popsáno, jakým způsobem a s jakými omezeními lze v současné době pracovat s XML dokumenty v relačních databázích. Práce se také zmiňuje o jejich alternativě. Tou je nativní XML databáze. Dále popisuje principy temporálního uložení dat a podrobně se zabývá konkrétním modelem určeným pro temporální XML databázi. Tyto poznatky jsou později využity při návrhu vlastní temporální XML databáze v kapitole 9. Stejně jako bylo nutné u temporálního rozšíření relační databáze *TSQL2* upravit dotazovací jazyk, tak bylo nutné v této kapitole rozšířit jazyk pro práci s XML dokumenty o temporální prvky. Konkrétně je implementováno temporální rozšíření pro podmnožinu jazyka *XPath* a navržen vlastní *DDL* a *DML* jazyk pro modifikování XML dokumentů v databázi.

Jako úložiště dat zde byla použita relační databáze, protože se jedná o nejrozšířenější databázi v současné době. Ovšem její použití není nezbytné a naopak by mohlo být efektivnější použít jako rozšíření jinou databázi. Tou by mohla být například databáze *Berkeley DB*.

Prostřednictvím kapitoly 10 zprostředkovává práce postřehy a problémy z fáze implementace a také navrhuje další rozšíření. Na tuto kapitolu navazuje kapitola 11, kde je popsáno výkonnostní testování, včetně porovnání výsledků s nativní XML databází *eXist-db*. V těchto testech dopadla databáze *eXist-db* mnohem lépe, nicméně jsou zde navržená rozšíření, která by měla rozdíl podstatně zmenšit.

Do archivu *JAR* byla integrována databáze *H2*. Data tak lze ukládat i do souboru. Mimo této databáze byla úspěšně otestována i podpora databáze *PostgreSQL*.

Hlavní přínosy práce oproti použitým zdrojům jsou následující. Je zde vyřešen problém s textovými uzly. Také byl navržen způsob, jak ukládat XML data do relační databáze a jak s těmito daty pracovat. V uvedených zdrojích například vůbec nebyly řešeny jmenné prostory. A v neposlední řadě je zde popsána implementace podmínek konzistence a vyřešen byl taktéž problém řazení a filtrování.

Další rozšíření by se mělo ubírat hlavně cestou optimalizace. Také je nutné obohatit rozhraní knihovny včetně navržené gramatiky.

Zdrojové kódy jsou zveřejněny pod licencí *Apache 2.0* spolu s popisem a ukázkami použití na `https://github.com/tkunovsky/TXML`.

Literatura

[1] Bourret, R.: rpbourret.com - XML consulting, writing, and research [online]. http://www.rpbourret.com, 2010 [cit. 2014-10-21].

[2] Bílek, P.: Transakce prakticky [online]. http://www.sallyx.org/sally/psql/transakce-prakticky.php, 2015 [cit. 2016-05-09].

[3] Bříza, P.: Základy jazyka XPath [online]. http://interval.cz/clanky/zaklady-jazyka-xpath, 2004 [cit. 2014-12-30].

[4] Horčička, J.: *Temporální rozšíření pro Java Data Objects, diplomová práce*. FIT VUT v Brně, 2012.

[5] Kolář, D.: Pokročilé databázové systémy [online]. www.fit.vutbr.cz/study/courses/PDB/private/lectures/prednesyPRD-IV-noBG.pdf, 2002 [cit. 2015-01-08].

[6] Lorentz, D.: *Oracle Database SQL Reference, 10g Release 2 (10.2)*. Oracle, 2005, 1428 s., b14200-02.

[7] Lysák, J.: XQuery: dotazovací jazyk nad XML [online]. http://jls.webz.cz/mff/xquery.pdf, 2002 [cit. 2015-01-02].

[8] Oracle: Lesson: Algorithms (The Java™ Tutorials > Collections) [online]. https://docs.oracle.com/javase/tutorial/collections/algorithms/, 2015 [cit. 2016-05-08].

[9] Pecinovský, R.: *Návrhové vzory - 33 vzorových postupů pro objektové programování*. Computer Press, 2007, ISBN 978-80-251-1582-4.

[10] Pokorný, J.: XML databáze: současný stav a perspektivy [online]. http://www.ksi.mff.cuni.cz/~pokorny/papers/DATAKON04-ready.pdf, 2004 [cit. 2014-10-23].

[11] Richta, K.: Jazyky XQuery a XPath [online]. https://www.ksi.mff.cuni.cz/~richta/publications/RichtaMD2006.pdf, 2006 [cit. 2014-12-26].

[12] Rizzolo, F.; Vaisma, A. A.: Temporal XML: Modeling, Indexing, and Query Processing. *The VLDB Journal*, ročník 17, č. 5, 2008: s. 1179–1212.

[13] Rychlý, M.: Temporální a deduktivní databáze [online].
www.fit.vutbr.cz/~rychly/public/docs/slides-demo3-tmp_and_deductive/slides-demo3-tmp_and_deductive.print.pdf, 2014 [cit. 2015-01-08].

[14] Tomek, J.: *TSQL2 interpret nad relační databází, diplomová práce.* FIT VUT v Brně, 2009.

[15] Zendulka, J.: Podpora XML v databázích [online].
wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/PDB-IT/lectures/XMLDB.pdf?cid=9505, 2010 [cit. 2014-10-15].

Přílohy

Seznam příloh

A Obsah CD **85**

Příloha A

Obsah CD

- technicka_zprava – zdrojové kódy v LaTeXu a finální PDF soubor
- java_archiv – finální Java archiv použitelný jako externí knihovna
- ukazky_pouziti – popis v angličtině s příklady v HTML
- demo – pár příkladů pro rychlé vyzkoušení knihovny
- programova_dokumentace – vygenerovaná JavaDoc HTML dokumentace
- zdrojove_kody – zdrojové kódy v GIT repozitáři, ze kterých byl vytvořen finální Java archiv