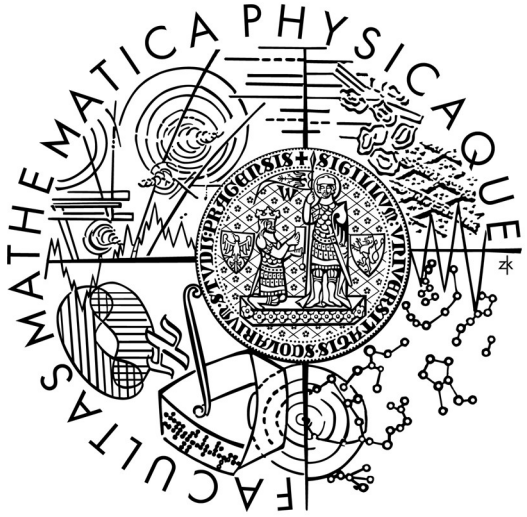
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Petr Švec

Datové úložiště pro rozvolněné objekty

Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.

Studijní program: Informatika
Studijní obor: Softwarové systémy

Rád bych poděkoval vedoucímu mé diplomové práce RNDr. Michalovi Kopeckému, Ph.D. za pomoc při výběru tohoto zajímavého tématu, umožnění jeho temporálního rozšíření, podnětné připomínky k návrhu úložiště i následné požadavky na implementované úložiště. Podpora RNDr. Michala Kopeckého, Ph.D. mně ve fázi implementace umožnila využití všech dostupných možností a znalostí. Ve fázi psaní textu diplomové práce děkuji za jeho odbornou pomoc a cenné rady.

Dále bych rád poděkoval RNDr. Michalovi Žemličkovi, Ph.D. za pomoc při výběru vhodné struktury indexu.

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

V Praze dne 5. 8. 2009

Petr Švec

Obsah

Abstrakt

Název práce: Datové úložiště pro rozvolněné objekty
Autor: Petr Švec
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.
e-mail vedoucího: Michal.Kopecky@mff.cuni.cz

Abstrakt:

Rozvolněné objekty jsou do databáze ukládány po jednotlivých atributech. Každý atribut je mapován na právě jednu tabulku. V každé tabulce jsou hodnoty pro všechny objekty, na kterých je příslušný atribut definován. Zatímco je takto navržené úložiště vhodné pro ukládání heterogenních dat či dat, jejichž struktura není předem známa, neumožňuje standardně efektivní vyhledávání přes více atributů, protože v současných databázových systémech nelze vytvořit index nad více tabulkami najednou.

V práci je navrženo a implementováno úložiště pro rozvolněné objekty rozšířené o temporální složku. Primárním cílem bylo umožnit efektivní vyhledávání. Navržená struktura indexových struktur a katalogu umožňuje jak vyhledávání datového obsahu daného objektu tak i vyhledávání vyhovujících objektů podle hodnot atributů v určitý časový okamžik. Implementace úložiště je vytvořena pro databázový server Oracle.

Klíčová slova: Rozvolněné objekty, index, relační databáze, Oracle

Title: Data store for relaxed objects
Author: Petr Švec
Department: Department of Software Engineering
Supervisor: RNDr. Michal Kopecký, Ph.D.
Supervisor's e-mail address: Michal.Kopecky@mff.cuni.cz

Abstract:

Individual relaxed object attributes are stored in the database one by one. Each attribute is mapped in a unique table. All values of given attribute are saved in the same table for all object instances, where the attribute is defined. While the relaxed object store is an outstanding instrument for storing heterogenic data and/or data without previously known structure, it does not support effective searching using more attributes simultaneously as the current databases do not support indexes over more tables.

The objective of the thesis is to design and implement a relaxed object store with a temporal extension. The primary objective is an effective searching. For this purpose a new structure for data indexing over multiple tables is proposed. A newly defined index structure can be used for searching for any object by attribute values. The relaxed store catalog makes it possible to search for attribute data values of given objects as well as searching for objects according to their attribute values. Implementation of the relaxed object store is designed for Oracle Database server.

Keywords: Relaxed objects, index, relational database, Oracle

1 Úvod

V klasickém objektově orientovaném datovém modelu bývají objekty rozděleny do předem definovaných tříd. Každá instance objektu potom náleží do právě jedné třídy. Třída má pevně definovanou množinu atributů, rozsah domén pro dané atributy a metody, které umožňují manipulaci s obsahem instancí objektů dané třídy. Tento takzvaný *class-based* model je použit v řadě vyšších programovacích jazyků jako například C++, C#, Java, Pascal, Delphi a další.

V reálném světě je většinou velmi problematické během analýzy správně definovat všechny třídy objektů. Důvodů je několik: velký rozsah projektu, kdy s rozsahem roste i počet tříd objektů či nejasná specifikace od zákazníka. Dalším důvodem může být slučování dat ze dvou a více heterogenních zdrojů, kdy se kromě hierarchií tříd a domén jednotlivých atributů mohou lišit i množiny atributů evidovaných pro každou třídu. I v případě homogenních zdrojů mohou být příčinou problematického definování tříd špatně definovaná integritní omezení.

Objekty lze rozdělit do předem stanovených tříd někdy jen s problémy. Navíc mohou objekty reálného světa během svého života některé vlastnosti získávat či ztrácet, a tak mezi třídami v čase migrovat.

Migraci mezi třídami objektů lze již dnes v class-based modelu částečně řešit tím, že se navrhne nejprve velká množina všech potenciálních atributů, které by někdy objekty mohly mít a poté se k danému objektu vlastnosti z této množiny přidávají. Nevýhoda tohoto přístupu je zřejmá, neboť přidávat lze pouze ty vlastnosti, na které bylo předem pamatováno.

Jinak je nutno při přidání či odebrání atributů třídu objektu změnit. Pokud v aplikaci třída ještě definována není, je nutné ji nadefinovat. Elegantněji lze model v praxi realizovat pomocí middlewarových[1] nástrojů jako je Hibernate[2] či NHibernate[3].

Opakem class-based modelu je méně rozšířený *object-based* model, kde jsou základním prvkem aplikace jednotlivé instance. Každé instanci lze přiřadit libovolné atributy a někdy i libovolné metody. Tento přístup je použit například v jazyce PHP.

Tato práce se zabývá datovým modelem pro rozvolněné objekty, nastíněném v práci Kopeckého a Žemličky 9, navrhovaným pro objekty, jejichž třída není explicitně předem definovaná. Tento model lze zařadit doprostřed mezi *class-based* a *object-based* modely.

[1] Middleware je software pro integraci více různých aplikací. Popisuje software, který spojuje dvě nebo více softwarových aplikací, které jim umožní výměnu dat.
[2] Hibernate je knihovna objektově-relačního mapování pro jazyk Java, která mapuje objekty mezi objektově-orientovaným modelem a tradičním relačním modelem databáze. Framework Hibernate 2.1 vyhrál v rámci soutěže Jolt Award v roce 2005.
[3] NHibernate je řešení objektově-relačního mapování pro Microsoft .Net. NHibernate je od verze 3.0 přímo součástí Microsoft .Net 3.5.

V tomto modelu mohou mít instance libovolné atributy, podobně jako v *object-based* modelu, ale na základě okamžitého datového obsahu jim může být přiřazena určitá třída. Stanovení třídy určité instance objektu se děje implicitně za běhu, nikoliv explicitně při návrhu jako v *class-based* modelu. Podle typu (třídy) objektu lze nad instancí vykonávat metody, které se k aktuální třídě objektu vážou. Cílem této práce je analyzovat typické operace nad temporálním rozšířením nastíněného modelu a navrhnout vhodné a efektivní databázové úložiště s požadovanou funkcionalitou.

Následující sekce na příkladu použití uvádí základní myšlenky modelu a porovnává jej s modely existujícími.

1.1 Motivační příklad státní správy

Státní správa je vzorovým prostředím pro existenci výše popsaných problémů. Každá instituce eviduje informace o občanech. Každému člověku jsou po narození přiřazeny a dále evidovány základní informace o jeho osobě, jako je jméno, příjmení, rodiče atd. Poté se v průběhu jeho života tyto informace obohacují o další specifická data. Například vzdělávací instituce udržují informace o dosaženém vzdělání, prospěchu či o obhajobě postupových prací. Odbor sociálního zabezpečení si uchovává informace o sociálním pojištění, důchodech a nemocenských dávkách. Finanční úřad eviduje odvedené daně. Zdravotní pojišťovny uchovávají informace o platbách zdravotního pojištění a provedené léčbě. Dopravní inspektorát udržuje informace o řidičských oprávněních a evidovaných vozidlech. Výsledkem je, že každý jedinec má evidovanou specifickou množinu informací, ukládanou leckdy částečně duplicitně v mnoha různých institucích či úřadech, přičemž každý úřad si data uchovává jinak.

Ve společnosti dochází k decentralizaci nebo naopak k centralizaci různých institucí a tak mohou vznikat různě závažné problémy s přenosem dat. V případě provádění centralizace dvou institucí není vhodné nadále nechat vést obě databáze souběžně, jelikož je potom nutné uchovávat data redundantně. Sloučení dat se zachováním veškeré informace rovněž vede k problémům, neboť dodefinování a převzetí všech atributů z druhé databáze povede k tomu, že mnohé atributy instancí evidovaných v původní databázi zůstanou prázdné. Nově vzniklá databáze tedy bude obsahovat příliš mnoho nedefinovaných hodnot. Metody pracující s těmito údaji musí mít tento fakt na zřeteli a být připraveny na to, že v řadě případů jsou zpracovávaná data neúplná. Je rovněž pravděpodobné, že při manuálním sjednocování obou databází vznikne mnoho chyb. Plně automatické sjednocení je přitom u rozsáhlých projektů

díky mnoha výjimkám téměř nemožné.

Úřady a státní aparát pro svůj správný chod potřebují určité informace. Pokud se jedná o informace, které lze získat dotazem pouze z jednoho úložiště, je získávaní dat snadné. Může to být například přehled počtu osob podle dosaženého vzdělání, zjištění průměrného příjmu, průměrně vynaložené náklady při léčbě pacienta atd.

Mnohem zajímavější je situace u složitějších dotazů, kdy je nutné zjistit požadovanou informaci z více různých zdrojů, protože to zpravidla vede k práci s objekty s různou vnitřní strukturou.

Například pro odpověď na dotaz *„zjisti seznam oprávněných voličů v daném volebním regionu“* je nutné projít instance všech potenciálních voličů v databázi. Při řešení otázky potenciálního voliče vzniká problém jak správně modelovat osoby, protože každý člověk může mít v podstatě libovolnou kombinaci zaměstnání a dalších zařazení, což vede k různé struktuře každé osoby v databázi. Problému modelování třídy osob se přitom nelze vyhnout, neboť je to nutná podmínka pro potenciální voliče. Řešení pomocí obecné podtřídy volič není ideální, protože tato podtřída je nezávislá na vlastnostech osob jako je například řidič, pacient, lékař atd. Konkrétní řešení dotazu by v praxi mohlo nejdříve zjistit všechny instance osoby starších 18 let s trvalým bydlištěm v regionu. To jsou právě ty objekty v databázi, které mají rodné číslo, jsou starší 18 let a jejich trvalé bydliště je v daném regionu. Navíc je nutné z dané množiny vyřadit lidi, kteří mají přiřazenu vlastnost označující, že jsou cizinci anebo že jsou nesvéprávní.

Další zajímavou ukázkou může být dotaz *„kdo všechno může pracovat jako řidič autobusu“*. V databázi je nutné vyhledat instance osob starších 21 let vlastnících řidičský průkaz skupiny D a majících dostatečnou praxi s řízením motorových vozidel. Navíc osoby nesmějí být trestně stíhané.

Pro správné fungování je nutné provádět s objekty různých tříd i manipulující operace. Například pokud člověk přijde o život, je nutné definovat jeho úmrtí, což v řeči temporálního modelu rozvolněných objektů může znamenat smazání a nebo zneplatnění všech atributů daného objektu.

1.2 Přehled práce

Předkládaná práce se zabývá modelem úložiště, ve kterém lze efektivně řešit problémy z motivačního příkladu ze státní zprávy od návrhu, přes implementaci až po měření výkonu. Vše je diskutováno v kontextu s klasickým objektovým modelem, který je v současných

programovacích jazycích nejrozšířenější.

Objektovému modelu je v předkládané práci věnována následující samostatná kapitola, protože je v dnešní době asi nejznámějším přístupem pro práci s objekty. Navíc je nutné čtenáře seznámit se všemi úskalími, které tento model skrývá, aby bylo možné v následujících kapitolách diskutovat o rozdílech mezi tímto modelem a modelem rozvolněných objektů.

Třetí kapitola popisuje používané techniky temporálního rozšíření klasického class-based modelu a reprezentaci temporální složky v současných úložištích.

Poté následuje formální definice rozšířeného modelu rozvolněných objektů s temporální složkou, kde lze zmíněné problémové situace efektivně řešit. Navržený model primárně vychází z práce autorů Kopeckého a Žemličky 9. Cílem je dostatečně formálně definovat úložiště pro rozvolněné objekty, které lze použít pro ukládání objektů skutečného světa. Objekty v něm mohou být v jednom okamžiku instancemi více tříd najednou a navíc mohou během života náležení do tříd libovolně měnit podle svého okamžitého datového obsahu.

V další kapitole je popsána reálná implementace modelu rozvolněných objektů s temporální složkou v databázovém systému Oracle.

V posledních dvou kapitolách je testování a měření výkonu předkládané implementace v kontextu reálně používaných úložišť. Naměřené hodnoty jsou diskutovány podle různých požadavků kladených na konkrétní úložiště.

Práce je doplněna popisem instalace implementovaného úložiště a textovou dokumentací všech funkcí implementovaného modelu.

2 Objektový model

K reprezentaci světa a ukládání informací o něm jsou uchovávané informace v tzv. class-based modelu obvykle omezené na konečnou množinu určitých tříd objektů. Každá třída si o objektu reálného světa pamatuje jenom několik předem definovaných typů informací (atributů).

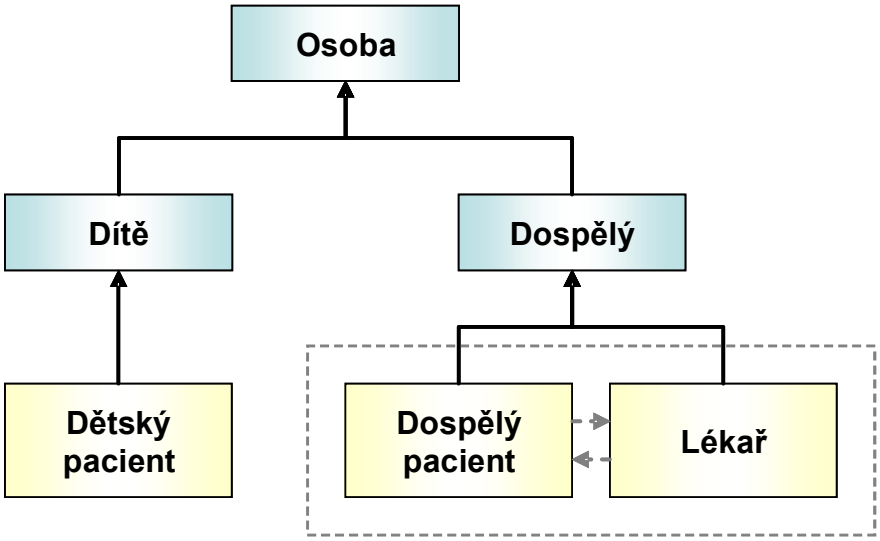
2.1 Objektový model na straně aplikace

V době designu databázové aplikace s použitím klasických databázových systémů jsou programátoři nuceni nejprve určit povolené typy objektů a potom do nich tyto datové objekty rozdělit. Principiální nedostatky tohoto modelování jsou uvedeny ve výše zmíněném motivačním příkladu zpracování dat ve státní správě. Problémem je právě explicitní rozdělení datových objektů do předem stanovených tříd. V reálném světě však objekty během svého života migrují mezi více třídami, proto je vhodné, aby byla třída definována implicitně až na základě vlastního datového obsahu instance. To může vést ke zjednodušenému návrhu aplikace i k vlastnímu ukládání dat.

Bez rozšíření class-based modelu je problémem nemožnost uchování té informace o objektu samotném, na kterou se v době návrhu tříd nepamatovalo. S tímto problémem se lze částečně vypořádat pomocí dědičnosti tříd objektů.

Například v aplikaci je již definovaná třída *Osoba* a od ní odvozené podtřídy *Dítě* a *Dospělý*. Pokud se budeme snažit rozšířit aplikaci o třídu pacient je to možné provést odvozením podtříd *Dětský pacient* ze třídy *Dítě* a *Dospělý pacient* ze třídy *Dospělý*. Odvození dvou nových tříd může mít praktické opodstatnění, protože některé typy onemocnění se vyskytují pouze u dětí a jiné pouze u dospělých. Vhodné rozšíření při dodefinování tříd pro pacienty je i vznik další nové třídy *Lékař,* neboť každý pacient lékaře navštěvuje. Podmínku, že lékař musí být dospělý lze splnit snadno vhodným odvozením. Zde již vzniká prostor pro znázornění nedostatků typického class-based modelu, protože každá instance objektu má odpovídat právě jedné třídě. V příkladu však dospělý může být *Dospělým pacientem*, *Lékařem* a nebo *Dospělým pacientem* a *Lékařem zároveň*. Řešení existuje v podobě odvození obecnější třídy pro dospělé. Na příkladě je vidět, že reálně objekty mohou odpovídat i více různým třídám najednou v určitý čas. Navíc - pokud se do databáze přidá časová složka - mohou objekty mezi třídami migrovat na základě aktuálního času a je nutné podle aktuální třídy přistupovat i k objektům. Například, pokud bylo dítě *Dětským pacientem* a nyní dospělo, je nutné některé informace o jeho chorobách přenést do instance objektu třídy *Dospělý*

pacient. V praxi jsou však třídy pro *Dětské pacienty* a *Dospělé pacienty* jiné nejen svoji strukturou, ale i počtem a typem definovaných atributů. Navíc problémem může být i neefektivita, protože pokud se nemají žádná data ztratit, tak je nutné některé informace uchovávat redundantně.



Obrázek 1 - příklad hierarchie tříd v aplikaci

Na obrázku (Obrázek 1) je zachycena modelová hierarchie tříd. V prvních dvou vrstvách jsou třídy *Osoba*, *Dítě* a *Dospělý*, které byly definovány v původní aplikaci. Nově dodefinované třídy jsou třídy *Dětský pacient*, *Dospělý pacient* a *Lékař*, které jsou ve spodní vrstvě. Šipky značí hierarchii tříd dědičnosti. Přerušovanými čarami je označena problémová situace v modelovém příkladu.

2.2 Frameworky pro objektový model

Nutnost náležení instance objektu právě jedné třídě v class-based modelu lze v aplikaci vyřešit tak, že v úložišti je jedna velká tabulka, která obsahuje atributy pro všechny typy objektů. Tedy instance všech objektů (s různými třídami) jsou mapovány do jediné tabulky. Pokud je daný atribut vyplněn hodnotou, znamená to, že objekt má nějakou vlastnost definovanou, jinak objekt vlastnost definovanou nemá. V aplikaci lze poté hledat objekty pouze podle daných vlastností. Dotazy tohoto typu lze snadno klást například v SQL, nevýhodou je však to, že programátor je při práci nucen starat se i o kontrolu, zda instance objektu vyhovuje dané třídě. V dnešní době se k snadnému mapování instancí objektů v aplikaci a v databázi používají frameworky jako Hibernate nebo NHibernate. Dotazy se v těchto frameworcích kladou v dialektu SQL zvaném HQL.

Oba dva výše zmíněné přístupy pomocí SQL či HQL umožňují pracovat s objekty různých tříd a objekty, které náleží více třídám najednou.

Například v případě dotazů *„vyhledej všechny lékaře“* a *„vyhledej všechny dospělé pacienty“* je možné, že osoba je jak lékařem, tak i pacientem. V případě hierarchie tříd znázorněné černými šipkami na obrázku (Obrázek 1) může být potom jedna osoba reprezentována dvěma instancemi objektů, kdy každá instance odpovídá právě jedné z obou tříd.

Modelové hodnoty jsou v tabulce (Tabulka 1). Atributy a náležení jednotlivým třídám je asociováno tečkovou notací, kdy jako prefix je jméno třídy a sufix je jméno atributu.

Osoba.Jmeno	Osoba.Prijmeni	Lekar.Identif_cislo	Pacient.Praktik_lekar	Pacient.Posledni_nemoc
Karel	*Novotný*	1	2	Angína
Jiřina	Srkalová	*null*	1	Alergie
Josef	Omáčka	*null*	*null*	*null*
Martina	*Nesmělá*	2	1	Zlomená noha

Tabulka 1 - hodnoty pro Hibernate

Toto na první pohled velmi elegantní řešení skrývá další úskalí, protože v databázi je objekt pouze jednou, ale v aplikaci je objekt v tomto případě dvakrát. Právě ony dvě instance jednoho objektu v aplikaci vedou k redundanci společných dat a v některých případech i k neaktuálnosti atributů namapovaných objektů. Množina objektů, které budou v aplikaci reprezentovány více třídami bude {*Karel Novotný*, *Martina Nesmělá*}. Například pokud jsou některé společné údaje modifikovány v jedné instanci, ve druhé instanci zůstanou původní, již neaktuální data. To může být problém, a proto je nutné uložení, vymazání objektu z cache a jeho znovunačtení z databáze již s aktuálními atributy.

Na tento problém musí pamatovat programátor sám, neboť v Hibernate není a z principiálních důvodů ani nemůže být nástroj, jak mapování jedné databázové instance na více aplikačních instancí omezit. Navíc v současné verzi Hibernate není žádná podpora pro temporální objekty.

V případě používání samotného SQL je problémem nutná znalost tříd instancí objektů a v některých případech nešikovná manipulace s daty. Temporální rozšíření, které je pro zachycení vývoje skutečných objektů v čase, je hodně těžkopádné.

2.3 Objektový model v databázi

Současné objektově orientované databáze (dále jen „OODB“), stejně jako objektově relační databáze (dále jen „ORDB“), vyžadují předem explicitně definovanou hierarchii tříd. Každý objekt musí po celou dobu života náležet právě jedné třídě. Případná změna třídy instance objektu je provedena externě aplikací, a to obvykle za cenu ztráty její původní identity. Toto řešení je neefektivní, protože je nutné překopírovat všechen požadovaný obsah z původního objektu do nového objektu. Navíc struktura instancí objektů různých tříd je zpravidla jiná.

2.4 Procedurální rozhraní

Na základě třídy objektu lze nad instancemi objektu provádět určité akce nazývané obvykle metody, které jsou se třídou objektu úzce svázané. Tyto metody (procedury či funkce) v aplikaci realizují určité procesy. Správná třída instance objektu je obvykle kontrolována už během překladu aplikace na základě deklarované hierarchie tříd. V případě zavolání funkce nad špatným typem objektu může aplikace nečekaně havarovat a to je nežádoucí. Bylo by tedy bezpečnější a efektivnější provést kontrolu správného typu objektu aplikací v době aktivace metody či ještě lépe vybrat objekty daného typu na základě definované omezující podmínky v databázovém systému. Toto řešení umožňuje mít všechna integritní omezení objektů a všechna omezení na prováděné procesy na jednom místě.

Pro usnadnění provádění skutečných aktivit se zavádějí zástupné procedury podle Žemličky a kolektivu 9, 9. Vhodná procedura pro realizaci dané aktivity se sama zvolí v závislosti na konkrétní vstupní množině datových objektů.

Konzistenci dat lze zabezpečit i způsoby známými z aktivních databází (viz. např. 9), kde se kontrola provádí ihned po splnění podmínek proveditelnosti.

2.5 Druhy uložení objektového modelu

Normální formy doporučují jak vhodně dekomponovat datové objekty a přímo tak ovlivňují výsledný návrh datového modelu. Smyslem normálních forem je navržení efektivního modelu pro dané úložiště a minimalizace redundancí. Proto se při reprezentaci dat v databázi používají tři základní způsoby mapování objektů 9 označované jako horizontální, vertikální a filtrované 9.

Horizontální metoda ukládá všechny atributy konkrétní třídy do samostatné relace, tj.

počet relací v databázi je roven počtu tříd objektů. Při migraci objektu mezi dvěmi třídami je nutné nejprve smazat řádek v původní relaci a do nové relace přidat záznam pro celý objekt. Například při rozšíření třídy s *k* atributy o 1 atribut se bude zbytečně kopírovat *k* atributů do nové relace. Výhoda této metody je snadná možnost přidání temporální složky, stačí přidat pouze dva časové sloupce do každé tabulky. Na druhou stranu, pokud se například u temporálního objektu s *k* atributy změní jeden atribut, bude objekt uložen jako nový záznam. V databázi tedy budou dva záznamy pro daný modifikovaný objekt s *k*-1 redundantními atributy.

Při vertikálním způsobu ukládání datových objektů jsou všechny nově definované atributy v dané třídě uloženy do samostatné relace. Pokud má třída například 3 předky, bude výsledný objekt uložen do 4 relací. Při vyhledávání je poté nutné procházet a spojovat velké množství tabulek. V krajním případě, kdy každá nová třída definuje pouze 1 nový atribut je třída s *k* atributy uložena v *k* relacích. V takovém případě by každá tabulka obsahovala pouze 1 sloupec atributu a 1 sloupec s vazbou na daný objekt. Toto uložení je v takto krajním případě hodně neefektivní, neboť odkaz na objekt, který má *k* atributů je v databázi uložen redundantně s *k* opakováními. V patologickém případě může dokonce existovat třída s *k* atributy, která má více něž *k* předků, protože některé třídy nemění datový záznam, ale jen chování. Přesto však bude potřeba nová tabulka s ID, aby bylo možné poznat, zda daná instance je nebo není instancí dané třídy. Na druhou stranu při získání kompletních dat lze spojování tabulky jen s ID sloupcem vynechávat (kromě té, která zároveň určuje danou třídu).

Filtrované mapování bylo demonstrováno výše (viz. Tabulka 1). Ukládá všechna data o všech objektech vždy do jedné a té samé relace. Výsledkem je tedy jedna velká tabulka, obsahující instance všech objektů. Pro každou nově definovanou třídu se do tabulky relace přidají nové sloupce příslušné k přidaným atributům nové třídy. Hodnota těchto sloupců je pro všechny předky dané třídy *null*. Naopak pro aktuální třídu a její potomky jsou hodnoty vyplněny určitou hodnotou. Tento způsob ukládání vede k tomu, že v databázi může postupem času vzniknout natolik velká tabulka, že je vhodné či nutné ji rozdělit. Důvodem nutnosti dělení tabulky může být její faktická velikost, protože rychle rostoucí počet hodnot (i *null* hodnot) v tabulce je úměrný součinu počtu instancí objektů a definovaných atributů všech tříd. Také počet sloupců jedné tabulky bývá v databázích omezen. Naštěstí se v reálném světě ukazuje, že nově přibývá méně nových tříd s daty oproti počtu uložených záznamů v databázi, což je pro filtrovaný způsob ukládání výhoda.

3 Temporální databáze

Pro zachycení skutečných objektů, které se v čase dynamicky mění, je nutné do databázového úložiště přidat temporalitu. V této kapitole je popsáno několik technik, které se v dnešní době používají. Primárně je čerpáno z prezentací 9, 9 a 9. Jednotlivá zde popsaná temporální úložiště se liší především podle použité složky času, uspořádání a hustoty.

3.1 Uspořádání

Nejznámější uspořádání jsou lineární, větvící se a cyklické prezentované v práci 9. Lineární uspořádání je klasický pohled na svět, kdy čas plyne lineárně dopředu. Tedy kdy každý časový okamžik má právě jeden bezprostředně předchozí a bezprostředně následující časový okamžik. Oproti tomu větvící uspořádání je navrženo tak, že je více možných budoucích okamžiků, vždy však alespoň jeden. Jedná se o případ větvícího příkazu. Například student střední školy si podává přihlášku a skládá zkoušku na vysokou školu. V době úspěšného absolvování přijímacích zkoušek bude jeho stav opět student, jinak nikoliv. Cyklické uspořádání je charakterizováno lineárním omezením na konečnou množinu časových okamžiků tvořících konečné těleso. Přičítání času je modulováno velikostí tělesa, a čas se tedy neustále cyklicky opakuje.

3.2 Hustota

Hustota časového modelu může být diskrétní, hustá anebo spojitá. Obvykle závisí na datovém typu pro čas. V závislosti na použitém typu se počítá buď s posloupností časových okamžiků nebo s časovým úsekem.

V současných databázových systémech je nejrozšířenější hustota diskrétní, protože i při použití velmi přesných časových typů jako například časové razítko je typ stále definován konečnou množinou hodnot. Zahrnutí skutečných reálných čísel do databázových systémů by bylo neefektivní.

3.3 Vztah událostí a času

Klasický pohled na stávající databáze bez temporality je označován jako snapshot. Tento datový model umožňuje zaznamenávat aktuální stav modelované reality. Každá n-tice představuje potom platný fakt v reálném světě v daný časový okamžik.

Pokud databáze podporuje *transakční čas,* nabízí možnost ukládání uspořádaných snapshotů za sebou. To znamená, že je vždy zachována kauzální posloupnost akcí v čase. Výhodou transakčního času je nízká režie nutná k pamatování si posloupnosti událostí, neboť každé změně stačí přiřadit jednoznačný identifikátor z vhodné uspořádané množiny. Transakční čas má tedy diskrétní hustotu. Nevýhodou modelu je to, že uživatel nemá možnost ptát se, jak dlouhý byl časový interval mezi událostí *A* a událostí *B*, protože tato informace v databázi uložena není. Lze zjistit pouze to, zda událost *A* nastala před událostí *B* či nikoliv.

Databáze může dále podporovat ukládání informací s *časem platnosti*. U každé n-tice je uloženo časové razítko s údajem, kdy byla informace do databáze vložena. Změna n-tice se provádí doplněním druhého časového razítka s dobou zneplatnění původní n-tice a vložením n-tice nové. Mazání se provádí opět pomocí zneplatnění původního údaje. Databáze si tak pamatuje nejen aktuální stav, ale rovněž informaci o všech předchozích hodnotách. To může být důležité nebo dokonce nezbytné pro řadu aplikací. Například lékařské informace by měly být vždy ukládány s časem platnosti. Ukládá se tak nejen aktuální zdravotní stav pacienta, ale i jeho kompletní historie.

3.4 Temporální relace

Temporální databázový systém, který podporuje časovou složku při ukládání dat, zjednodušuje dotazy, ulehčuje údržbu aplikace a řeší problém klasických databázových systémů jak naložit se starými daty.

Asi nejčastěji se jedná o podporu času platnosti s diskrétní hustotou času, kdy je temporální rozšíření aplikováno na celý objekt, a nikoli na jeho atributy. Ukázka je uvedena v tabulce (Tabulka 2).

Jméno	Plat	Funkce	Datum narození	Platí_od	Platí_do
Josef	12000	Vrátný	1945-04-09	1995-01-01	1995-06-01
Josef	14000	Vrátný	1945-04-09	1995-06-01	1995-09-01
Josef	18000	Vrchní vrátný	1945-04-09	1995-09-01	1996-02-01
Josef	25000	Ředitel bezpečnosti	1945-04-09	1996-02-01	1997-01-01

Tabulka 2 - příklad temporální relace

V příkladu jsou zahrnuty jak informace, které se mohou měnit - např. výše *Platu* - tak i fakta, které se měnit nemohou - např. *Jméno* a *Datum narození*. Výsledkem je obvykle velká

redundance dat, neboť v jednom okamžiku se mění jenom malá podmnožina atributů objektu. V řadě případů je mnohem efektivnější neměnící se atributy ukládat zvlášť do samostatné relace. V uvedeném příkladu by bylo výhodnější ukládat samostatně atributy *Jméno* a *Datum narození*.

Pokud je do aplikace nutné zahrnout i časovou složku, jsou temporální databáze velmi často používané.

3.5 Bitemporální relace

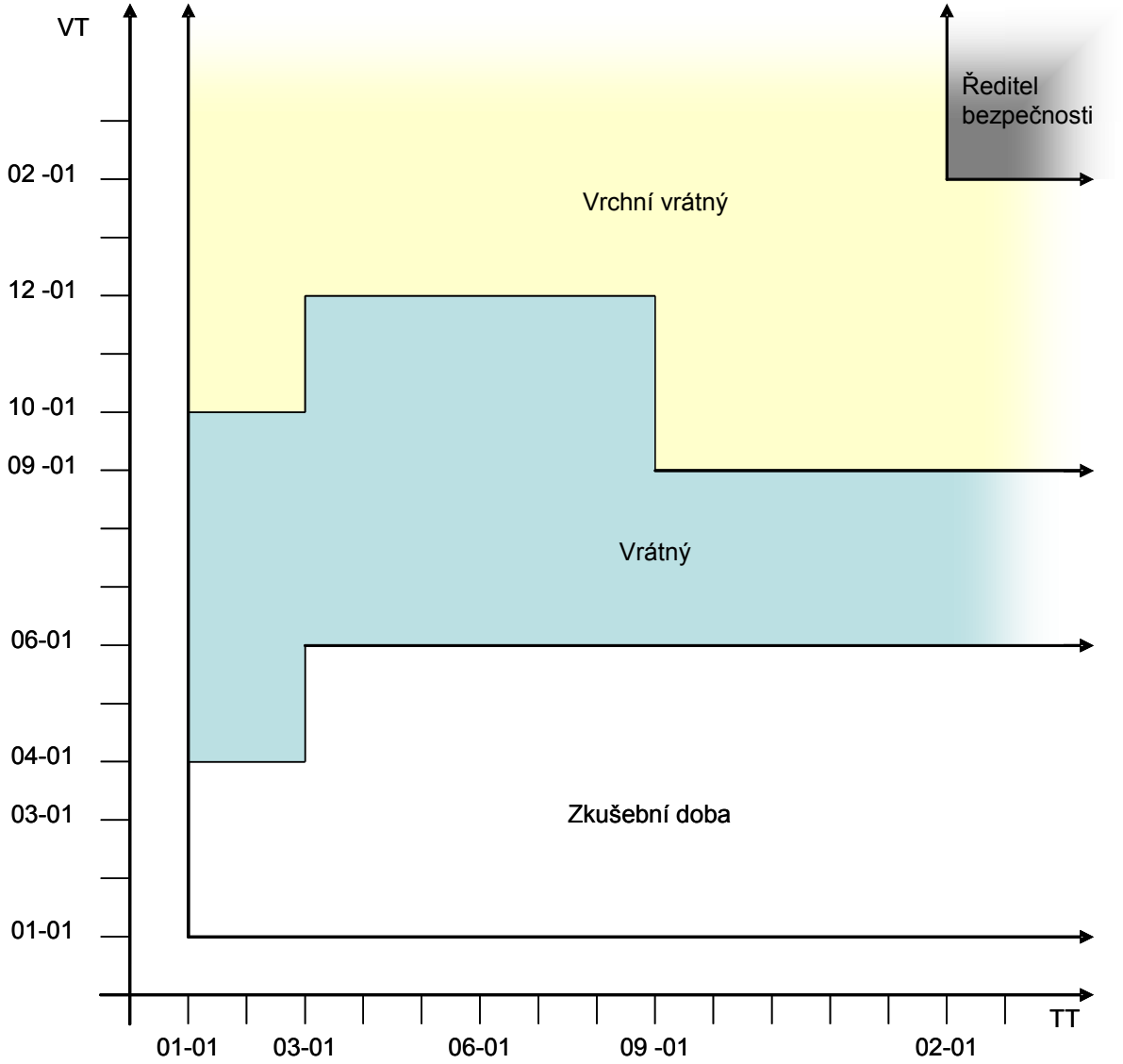
Jedná se o nejobecnější reálně používaný model uchovávání dat s časovou složkou. Kromě klasické temporality od kdy do kdy určitá informace platí je přidaná i další informace, a to interval, kdy je tato informace známa. Formálně je každá relace orazítkována množinou tzv. *bitemporálních chronomů*. Bitemporální chronom je dvojice transakčního času a času platnosti.

V příkladě uvedeném na obrázku (Obrázek 2) osa TT označuje skutečný čas a VT čas očekávané platnosti. Na obrázku je znázorněn kariérní postup vrátného Josefa z předchozího příkladu uvedeného v tabulce (Tabulka 2). Na obrázku je spodní plocha (bílá) zkušební doba na pozici vrátného, plocha uprostřed (modrá) je pozice vrátného, horní plocha je pozice vrchního vrátného a plocha vpravo nahoře (šedá) je pozice ředitele bezpečnosti. Josef nastoupí do práce dne 1.1.1995 s tím, že nejprve musí být vrátným alespoň 6 měsíců, než bude možný jeho další kariérní postup, konkrétně na pozici vrchního vrátného. Při nástupu do firmy pan Josef předpokládá, že první 3 měsíce bude ve zkušební době (interval od 1.1. do 1.4.), poté bude 6 měsíců pracovat jako vrátný (interval od 1.4. do 1.10.) a nakonec se od 1.10. stane vrchním vrátným.

Pan Josef nastoupil do práce a čas postupně plynul. Bohužel 1.3. se zjistí, že Josef podepsal špatnou nástupní smlouvu a je tedy nutné podepsat novou s tím, že jeho zkušební doba bude trvat do 1.6. místo původně předpokládaného termínu 1.4. Z důvodu posunutí zkušební doby až do 1.6. je nutné posunout i odhady, kdy se Josef stane vrátným (tzv. nový interval od 1.6. do 1.12.1995).

Jak čas postupně plyne dále, Josefovi skončí 1.6. zkušební doba a začne pracovat jako vrátný, jak předpokládal. Od 1.9. se nečekaně uvolní místo vrchního vrátného a firma nemá na tuto pozici žádného vhodnějšího kandidáta než pana Josefa a protože si Josef vedl dobře, vedení firmy mu odpustí nutných 6 měsíců na pozici vrátného. Josef se tedy, již 1.9. stane vrchním vrátným na dobu neurčitou.

1.2.1996 se opět nečekaně uvolní místo ředitele bezpečnosti a tak Josef od tohoto data pracuje na této nové pozici.



Obrázek 2 - ukázka bitemporálního záznamu

V demonstrovaném příkladě pan Josef hned od začátku předpokládá určitý růst v pracovních pozicích a postupem času se termíny pouze mění. Nicméně je důležité vědět, že do bitemporálních databází lze ukládat jakékoliv informace nepredikovaně. Pokud by se pan Josef dozvěděl, že od 1.10. se má uvolnit pozice vrchního vrátného až 1.3., byla by horní (žlutá) plocha, znázorňující tuto informaci, začínající v ose TT až v čase, kdy se informaci dozvěděl tedy 1.3. a ne 1.1. jak je uvedeno na obrázku.

Ve výše uvedeném příkladě je ukázáno obohacení klasického temporálního modelu o možnost zjistit, kdy která informace měla platit a možnost porovnat tento předpoklad se

skutečností.

Datový model podporující tyto relace se nazývá *Bitemporal Conceptual Data Model* (BCDM). V reálných aplikacích se v současné době vyskytuje jen zřídka.

4 Rozvolněné objekty

V této kapitole je popsán model rozvolněných objektů od formální definice a stanovení tříd pro takové objekty přes možnosti manipulace s daty až po procedurální rozhraní. Tento model primárně vychází z práce autorů Kopeckého a Žemličky 9, který je v předkládané práci rozšířen o lineární temporalitu. Je zde popsán i způsob ukládání dat včetně implementace úložiště pro takové objekty.

Cílem této kapitoly je dostatečně formálně definovat úložiště pro rozvolněné objekty, které lze použít pro ukládání objektů skutečného světa. Objekty v něm mohou být v jednom okamžiku instancemi více tříd najednou a navíc mohou během života náležení do tříd libovolně měnit podle svého okamžitého datového obsahu. Navržené a implementované úložiště by mělo umožnit ukládání objektů tohoto modelu a zajišťovat co nejefektivnější manipulaci s těmito objekty a jejich vyhledání.

Rozvolněný objekt je navenek velmi podobný objektu jakéhokoliv objektově orientovaného programovacího jazyka. Informace o objektu jsou ukládané v atributech. Jméno každého atributu v rámci datového modelu musí být unikátní. Termín rozvolněný objekt vznikl ze způsobu, jakým jsou jednotlivé objekty ukládány do perzistentního úložiště. Pro každý atribut existuje právě jedna relace, kde jsou informace o všech objektech, které mají daný atribut definován. Pokud tedy instance objektu má určenou nějakou vlastnost, je její hodnota uložena v příslušné relaci.

4.1 Formální definice

Návrh datového modelu je ovlivněn snahou o snadnou implementaci ve stávajících databázových systémech. Každý atribut a_i bude reprezentován právě jednou tabulkou. Atributy a_i jsou temporální, takže hodnota a_i se může v čase měnit. Množina hodnot pro atribut a_i je definována v právě jedné doméně D_i. Předpokládá se, že každá tato doména D_i bude reprezentovat atomický typ (v současných databázových systémech například číslo, řetězec, binární typ atd.), ale může to být i složitější struktura. Kvůli efektivnímu vyhledávání objektů v úložišti podle atributů je v případě použití vlastních struktur pro doménu nutné definovat kromě množiny hodnot i uspořádání prvků množiny. Navíc D_i není závislá na aktuálním čase, takže atribut a_i může nabývat obecně libovolnou hodnotu z D_i v jakémkoliv časovém okamžiku.

4.1.1 Datový model

- $D = \{ D_1, D_2, D_3, \ldots, D_k \}$ je konečná množina domén.
- $A = \{ a_1, a_2, a_3 \ldots, a_k \}$ je konečná množina atributů. Povolené hodnoty každého atributu a_i jsou definované odpovídající doménou $D_i \supseteq \{null\}$.
- I je množina identifikátorů instancí objektů. V předkládané práci se stejně jako v práci 9 používá identifikace objektů pomocí přirozených čísel.
- $F = \{ f_1, f_2, f_3, \ldots, f_l \}$ je konečná množina funkcí pro manipulaci s datovým obsahem objektů. Každá funkce f_j má standardním způsobem deklarované parametry a návratový typ.
- $T = \{ t_1, t_2, t_3, \ldots, t_m \}$ je množina hodnot času pro temporální složku definovaná jako lineární. Je nutné, aby všechny t_l splňovaly kauzální závislost. T může být jak spojitá tak i diskrétní.

Datový model schématu S je potom jednoznačně určen pěticí $S=(D,A,I,F,T)$.

4.1.2 Instance datového modelu

- $O = \{o_1, o_2, o_3, \ldots, o_n\}$ je konečná množina objektů.
- $id: O \leftrightarrow I$ je bijekce mezi množinou objektů O a množinou identifikátorů I.
- $val: O \times A \times T \rightarrow I \times D$ je zobrazení, které každému objektu $o \in O$ a atributu a_i přiřazuje v libovolný časový okamžik t odpovídající hodnotu z D_i. Tedy každý objekt je v jakýkoliv časový okamžik prvkem kartézského součinu $I \times D_1 \times D_2 \times D_3 \times \ldots \times D_k$.
- $C = \{c_1, c_2, c_3, \ldots, c_p\}$ je konečná množina booleovských podmínek nad atributy.

Instancí datového modelu M je potom jakýkoliv stav relační databáze, který odpovídá S. Ten je reprezentován pěticí $M=(S,O,id,val,C)$.

4.1.3 Množiny atributů objektů

O objektu $o \in O$ můžeme říci, že definuje atribut a_i v čase $t \in T$, pokud zobrazení $val(o, a_i)^t$!= *null*. Množina atributů definovaných na objektu o v čase t je potom označována jako $def(o)^{(t)}$, formálně ve vztahu (R 1). Stavem objektu o v čase t se rozumí obsah celé množiny právě definovaných atributů na instanci objektu.

$$def(o)^{(t)} = \{a_i \mid a_i \in A \,\&\, val(o, a_i)^t != null\} \qquad \textbf{R 1}$$

Pro označení množiny definovaných atributů v aktuální časový okamžik se používá zkrácený zápis $def(o)$.

Množina všech atributů, které kdy byly definovány na objektu *o* se označuje $\overline{def}(o)$ podle vztahu (R 2).

$$\overline{def}(o) = \bigcup_{\forall t \in T} def(o)^{(t)} \qquad \textbf{R 2}$$

Jádrem objektu se myslí množina atributů, které má objekt definované po celou dobu své existence, značí se $\underline{def}(o)$ dle vztahu (R 3). $T_D \subseteq T$ je množina časových okamžiků, kdy měl objekt *o* alespoň jeden atribut. Formálně řečeno pro dvojici [*o,t*] objektu $o \in O$ a množinu $T_D \subseteq T$ je $| def(o)^{(t)} | \geq 1$ pro všechna $t \in T_D$.

$$\underline{def}(o) = \bigcap_{\forall t \in T_D} def(o)^{(t)} \qquad \textbf{R 3}$$

Pro označování jednotlivých složek (atributů) objektu $o \in O$ v čase $t \in T$ je v této práci použita klasická tečková notace jako $o.id$, $o.a_1^{(t)}$, $o.a_2^{(t)}$, … $o.a_k^{(t)}$. Pro označení aktuálních hodnot bude identifikace vypadat následovně $o.id$, $o.a_1$, $o.a_2$, … $o.a_k$,

Díky temporalitě tedy může existovat objekt, který v žádném časovém okamžiku nedefinuje všechny své atributy. Tato vlastnost ztěžuje pozdější procedurální rozhraní (viz. 4.3) nad rozvolněnými objekty, neboť třída objektu se mění v závislosti na aktuálním časovém okamžiku.

4.2 Typy

V klasických programovacích jazycích založených na class-based modelu jsou třídy objektů předem dané. Každá třída je potom explicitně definovaná, a proto je i jasná hierarchie dědičnosti od obecného objektu až ke konkrétnímu objektu dané třídy. U rozvolněných objektů lze definovat třídy dvěma základními přístupy. Prvním, jednodušším způsobem je stanovení implicitní třídy objektu na základě aktuální množiny atributů dané instance. Druhým, obecnějším přístupem se třídy definují libovolnou booleovskou podmínkou. Díky temporalitě a potenciálním změnám třídy v čase nelze u objektu určit třídu globálně, ale pouze v nějakém konkrétním časovém okamžiku.

4.2.1 Třídy definované množinou atributů

Třídou objektu $o \in O$ v libovolném časovém okamžiku $t \in T$ se u rozvolněných objektů myslí množina atributů objektu definovaných v daném čase, tedy množina $def(o)^{(t)}$. Třída objektu *o* v čase *t* se značí jako $[o]^{(t)}$. Aktuální třída objektu *o* se zkráceně zapisuje [*o*]. Každá třída odpovídá právě jedné podmnožině atributů, a proto nezáleží na pořadí přidávání atributů či odebírání jednotlivých atributů. Pro pohodlí programátora lze třídy pojmenovat. Na práci úložiště to však nebude mít žádný efekt. U pojmenovaných tříd se může stát, že v úložišti neexistuje v určitý časový okamžik ani jedna instance objektu, jejíž atributy by odpovídaly nějaké pojmenované třídě.

$\underline{[o]}$ je třída jádra objektu a rozumí se jí minimální množina atributů deklarovaných po celou dobu života na objektu, tedy $\underline{def}(o)$. $\overline{[o]}$ je maximální třída - maximální množina atributů definovaných na objektu - tedy $\overline{def}(o)$. Formální definice $\underline{def}(o)$ je ve vztahu (R 2) a $\overline{def}(o)$ ve vztahu (R 3). Třídy $\underline{[o]}$ a $\overline{[o]}$ nejsou závislé na zvoleném časovém okamžiku, proto jsou dále v textu označovány jako globální na rozdíl od $[o]^{(t)}$ a [*o*], které jsou temporální.

Náležení objektu $o \in O$ třídě [*o*] se značí $o \in [o]$.

O třídě $[o_1]$ určené množinou $def(o_1) \subseteq A$ a třídě $[o_2]$ určené množinou $def(o_2) \subseteq A$ lze říci, že $[o_1]$ je potomkem (podtřídou) $[o_2]$, pokud platí, že $def(o_1) \supseteq def(o_2)$. formálně zapsáno ve vztahu (R 4a).

$$[o_1] \text{ je potomkem } [o_2] \Leftrightarrow def(o_1) \supseteq def(o_2) \qquad \textbf{R 4a}$$

Třída $[o_1]$ je přímým potomkem třídy $[o_2]$ právě tehdy, když $[o_1]$ má o jeden atribut více. Formální definice je uvedena v (R 5).

$$[o_1] \text{ je přímým potomkem } [o_2] \Leftrightarrow def(o_1) \supset def(o_2) \,\&\, |\, def(o_1) \,|=|\, def(o_2) \,| + 1 \qquad \textbf{R 5}$$

Dále lze definovat potomky nepřímé. $[o_1]$ je nepřímým potomkem $[o_2]$, jestliže $[o_1]$ je potomkem $[o_2]$ a $[o_1]$ má alespoň o dva atributy více než $[o_2]$. Tedy $[o_1]$ je nepřímým potomkem $[o_2]$ právě tehdy, když existuje více než dvou prvková množina $B \subset A$ taková, že všechny prvky $b \in B$ jsou i prvky $def(o_1)$ a přitom nejsou obsaženy v $def(o_2)$. Formálně zapsáno ve vztahu (R 6).

$[o_1]$ je nepřímým potomkem $[o_2] \Leftrightarrow \exists B \subseteq A : B \subseteq def(o_1) \,\&\, B \not\subset def(o_2) \,\&\, |B| \geq 2$ **R 6**

Analogicky se definují i předci v hierarchii tříd. Formální zápisy jsou ve vztazích (R 7a), (R 8) a (R 9). Přímý předek je v textu dále označován jako rodič.

$[o_2]$ je předkem $[o_1] \Leftrightarrow def(o_1) \supseteq def(o_2)$ **R 7a**

$[o_2]$ je přímým předkem $[o_1] \Leftrightarrow def(o_1) \supset def(o_2) \,\&\, |def(o_1)| = |def(o_2)| + 1$ **R 8**

$[o_2]$ je nepřímým předkem $[o_1] \Leftrightarrow \exists B \subseteq A : B \subseteq def(o_1) \,\&\, B \not\subset def(o_2) \,\&\, |B| \geq 2$ **R 9**

Při definici třídy pomocí množiny atributů je tedy ekvivalence tříd $[o_1]$ a $[o_2]$ v časový okamžik $t \in T$ definována podle vztahu (R 10a).

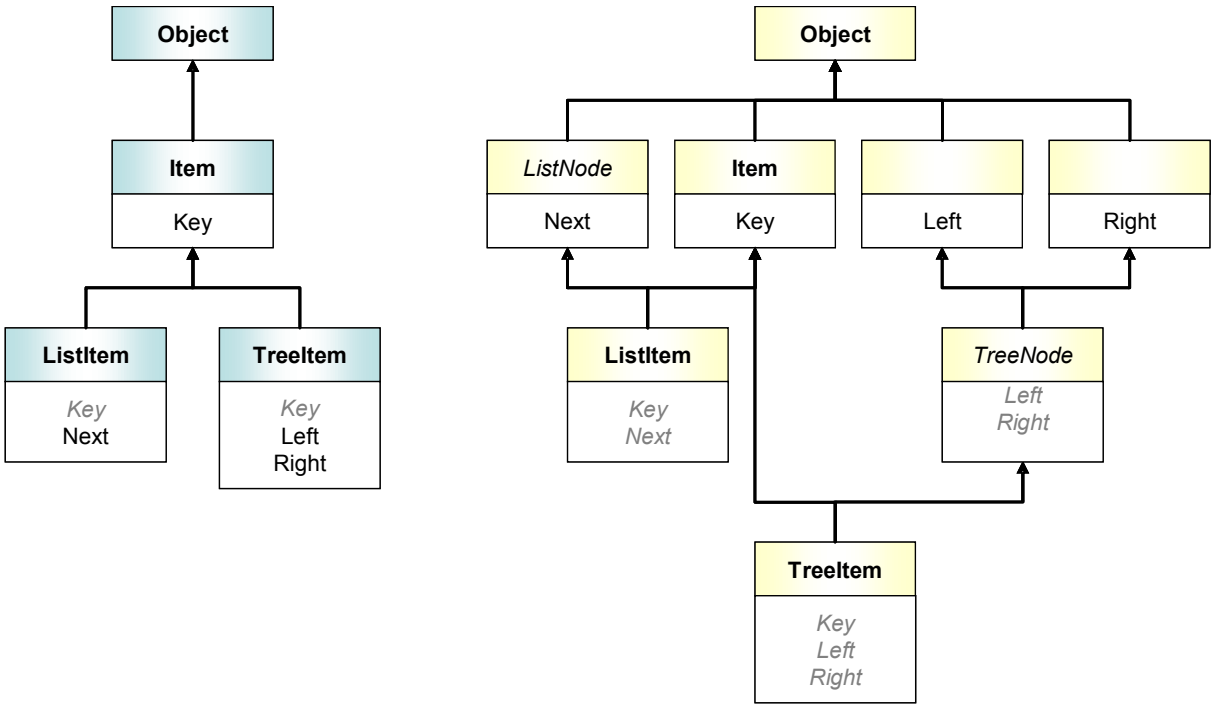
$$[o_1]^{(t)} \equiv [o_2]^{(t)} \Leftrightarrow def(o_1)^{(t)} = def(o_2)^{(t)} \quad \textbf{R 10a}$$

Tyto třídy jsou implicitními třídami objektů. Pokud nebude v textu uvedeno jinak, bude se pod pojmem třída skrývat implicitní třída. Tato definice tříd je použita i v implementační části.

Ztotožněním tříd s množinami atributů je teoretický počet všech tříd roven počtu různých podmnožin atributů A ve schématu, tedy hodnotě $2^{|A|}$. K jedné třídě vede v hierarchii od kořenové třídy $\varnothing$ více cest. Konkrétně ke vzniku třídy s k atributy vede až $k!$ cest.

Na obrázku (Obrázek 3) je na abstrakci definice položek spojového seznam *ListItem* a binárního vyhledávacího stromu *TreeItem* znázorněn rozdíl mezi explicitním definováním tříd v objektově orientovaném jazyku a částí hierarchie implicitních tříd rozvolněných objektů. V klasických modelech jako jsou objektově orientovaný a objektově relační by bylo nezbytné zavést obecnou třídu *Item* s atributy, které mají položky společné a z té by bylo provedeno další odvození. Dále lze definovat třídu *Item* jako abstraktní a tím zamezit vytváření instancí objektů dané třídy.

Naproti tomu třída *TreeItem* u rozvolněných objektů v příkladu vznikne implicitně jako množina požadovaných atributů {*Key, Left, Right*}. Obdobně je třída *ListItem* totožná s množinou atributů {*Key, Next*}. Instance objektů mohou podle svého okamžitého datového obsahu patřit do libovolné z nich. Aplikace však obvykle nepracuje se všemi třídami, ale pro svoji činnost využívá jen jejich omezenou podmnožinu. V tomto příkladě může teoreticky vzniknout až 16 různých tříd. Implicitních tříd definovaných podle aktuálního datového obsahu však vzniklo pouze 8.



Obrázek 3 - explicitní model tříd vs. hierarchie implicitních tříd

Na obrázku (Obrázek 3) vpravo jsou implicitní třídy znázorněny ve vrstvách podle počtu atributů. V první vrstvě je nejobecnější třída *Object*, odpovídající prázdné množině definovaných atributů. Ve vrstvě níže jsou zobrazeny všechny třídy s právě jedním atributem, v další vrstvě to jsou některé ze tříd se dvěma atributy atd. Kromě tříd z levé části obrázku jsou vyznačeny ještě dvě třídy *ListNode* a *TreeNode*. Ne všechny třídy modelu musí mít jméno. Ve skutečnosti slouží jména tříd pouze jako mnemotechnická pomůcka pro snazší orientaci programátora.

Šipky na obrázku (Obrázek 3) v pravé části, tj.v hierarchii implicitních tříd, znázorňují možné cesty při dědění atributů pro konečné třídy, které svou logickou strukturou odpovídají explicitnímu modelu z pohledu programátora. Třída *TreeItem* však nemusí vzniknout ze dvou tříd *Item* a *TreeNode*, jak je uvedeno na obrázku. V grafu $(P(A),\subseteq)$ může třída $\{Key, Left, Right\}$ vzniknout až ze sedmi různých nadtříd $\{Key, Left\}$, $\{Key, Right\}$, $\{Left, Right\}$, $\{Key\}$, $\{Left\}$, $\{Right\}$, $\varnothing$. V kontextu přímých potomků by třída $\{Key, Left, Right\}$ z příkladu mohla vzniknout pouze ze tří přímých nadtříd $\{Key, Left\}$, $\{Key, Right\}$, $\{Left, Right\}$. Pokud by bylo povolené odvozovat nové třídy jen takto (jako přímé potomky), potom by tvořil graf hierarchie $|A|$-rozměrnou krychli.

Vzhledem k tomu, že model dovoluje definovat pro libovolnou instanci libovolnou množinu atributů, nemá v tomto modelu smysl hovořit o pojmu abstraktní třídy. Teoreticky by sice bylo možné zakázat některé konkrétní kombinace atributů, například $\{Next\}$, $\{Left\}$,

{*Right*} a {*Left, Right*} za předpokladu, že od kořene hierarchie ke každé z používaných tříd bude vést alespoň jedna povolená cesta v grafu dědičnosti ($P(A), \subseteq$). Na druhou stranu by však test na náležení do zakázané třídy trval v modelu s mnoha atributy příliš dlouho, protože by se musely otestovat všechny atributy. Model je určený pro situace, kdy pojem zakázané třídy nemá smysl. Maximálně se může stát, že objekt dané třídy bude v rámci procedurálního rozhraní považován za instanci některé své nadtřídy až do doby, než získá nějaké další atributy a stane se použitelný pro další definované metody.

I při vytváření tak malého počtu tříd jako je na obrázku (Obrázek 3) a přes zakreslení všech nutných tříd z celé hierarchie vznikají vedlejší třídy *ListNode*, *TreeNode* a dvě nepojmenované. I v aplikaci, kde by se pracovalo pouze se dvěma třídami *ListItem* a *TreeItem,* by v normálním class-based modelu musely být definovány dvě třídy navíc. U rozvolněných objektů se nemusí definovat žádná třída navíc. Programátor není nucen si implicitně vznikajících tříd všímat a pracovat s nimi. Celá aplikace může ve svých definicích používat jen třídy {*Key, Left, Right*} a {*Key, Next*}, které lze pro větší pohodlí programátora symbolicky pojmenovat. Pojmenování tříd je volitelná záležitost uživatele, nikoliv nutná.

Databázové úložiště pro rozvolněné objekty musí být schopné pro objekt v určitý časový okamžik rozhodnout, zda objekt náleží dané třídě či nikoliv a nalézt všechny instance objektů požadované třídy. Tyto požadavky vyplývají z používání metod nad instancemi objektů. Podrobně bude tato problematika diskutována v sekci procedurální rozhraní (4.3).

4.2.2 Třídy definované podmínkou

Definování implicitních tříd rozvolněných objektů pomocí množiny atributů umožňuje rozlišovat třídy jen na základě odlišných kombinací definovaných atributů.

Pro obecnější definování tříd rozvolněných objektů lze využít podmínek nad atributy $c \in C$, které umožňují mnohem širší specifikaci deklarované třídy. Každá třída je potom definována jednou podmínkou $c \in C, c : O \times T \rightarrow \{true, false\}$. Objekt $o \in O$ v časovém okamžiku $t \in T$ náleží třídě definované podmínkou $c \in C$ právě tehdy, když $c(o)^{(t)} = true$. Použitím logických spojek *and, or, not, xor, nor* atd. a závorek lze vytvářet libovolné podmínky na obsah atributů.

Nechť objekty $o_1, o_2 \in O$ náleží podle shodných indexů třídám $[c_1]$, respektive $[c_2]$, definovaným podmínkami $c_1, c_2 \in C$ v časový okamžik $t \in T$. Tedy $c_1(o_1)^{(t)} = true$ a $c_2(o_2)^{(t)} = true$. Definovat globální třídu jádra a maximální třídu při použití podmínek obecně nelze.

O třídě $[c_1]$ lze říci, že je potomkem třídy $[c_2]$, respektive o třídě $[c_2]$ lze říci, že je předkem třídy $[c_1]$, právě tehdy, když $c_1 \Rightarrow c_2$. Formálně předpis je uveden ve vztazích (R 4b) a (R 7b). Ekvivalence tříd (R 10b) je důsledkem implikací ze vztahů (R 4b) a (R7b).

$$[c_1] \text{ je potomkem (podtřídou) } [c_2] \Leftrightarrow (c_1 \Rightarrow c_2) \qquad \textbf{R 4b}$$
$$[c_2] \text{ je předkem (nadtřídou) } [c_1] \Leftrightarrow (c_1 \Rightarrow c_2) \qquad \textbf{R 7b}$$
$$[c_1] \text{ je ekvivalentní } [c_2] \Leftrightarrow (c_1 \Leftrightarrow c_2) \qquad \textbf{R 10b}$$

Definice hierarchie tříd $[c_1]$ určené podmínkou $c_1 \in C$ a tříd $[c_2]$ určené podmínkou $c_2 \in C$ je tedy reflexivní, stejně jako u tříd definovaných pomocí množin atributů. V obecném případě může být rozhodnutelnost implikace podmínek značně netriviální a určení správně hierarchie tříd tak může být neefektivní.

V tomto rozšířeném modelu příkladu z (Obrázek 3) je třída *Item* definovaná podmínkou (*Key*≠*null*). Je však možné definovat i třídu *SmallItem* pomocí podmínky (*Key*<256). Protože (*Key*<256) implikuje (*Key*≠*null*), je třída *SmallItem* potomkem třídy *Item* a třída *Item* je předkem třídy *SmallItem*.

V kontextu tříd definovaných podmínkami nelze rozlišovat přímé či nepřímé potomky ani předky.

4.2.3 Život objektu

Objekt je považován v datovém úložišti za *živý*, pokud má v daném čase definován alespoň jeden atribut. V případě, že objekt nemá definován žádný atribut, je označován za *neživý*. Neživou je tedy každá instance objektu, která v určitý časový okamžik náleží právě třídě *Object*. Naopak každá instance, která v určitý časový okamžik odpovídá jakémukoliv potomku třídy *Object* je považována za živou.

Objekty, které po celou dobu nikdy neměly definovaný žádný atribut, nejsou v modelu nijak zvlášť označovány.

Časem narození objektu se myslí zlomový okamžik $t \in T$, kdy mrtvý objekt získal alespoň jeden atribut (stal se instancí třídy různé od třídy *Object*). *Časem úmrtí* se myslí časový okamžik *t*, kdy objekt ztratil poslední atribut (stal se instancí třídy *Object*). Každý objekt může mít více časů narození i úmrtí. Život objektu tedy může být v patologických případech definován více nespojitými intervaly. Někdy proto budeme explicitně hovořit o *čase prvního narození* a *čase finálního úmrtí*.

4.3 Procedurální rozhraní

Aplikace využívající rozvolněné objekty potřebuje nad daty realizovat procesy určené během analýzy a specifikace dané aplikace. Aby bylo možné s daty uloženými v podobě rozvolněných objektů pracovat, je nad rozvolněnými objekty možné definovat procedurální rozhraní jak pro manipulaci s atributy objektů, tak i pro manipulaci s objekty jako celky. Kromě vlastních procesů jsou během analýzy stanoveny podmínky, za kterých lze procesy nad objekty vykonávat. Většinou jsou tyto podmínky pro splnitelnost provádění procesu nutné a zároveň i postačující. Problém splnitelnosti může být v některých případech natolik složitý, že by jeho řešení nešlo dostatečně efektivně najít. V takovém případě se musíme spokojit pouze se stanovením nutných podmínek.

Procesy jsou reprezentovány funkcemi. Funkce, které nevrací žádnou hodnotu, jsou dále označovány, stejně jako v class-based modelu, za procedury. V závislosti na kontextu procesu lze v modelu rozlišovat *běžné funkce* a *procedury, metody, triggery* a *agregační funkce*.

- *Běžné funkce* a *procedury* nejsou vázané na konkrétní instanci objektu a lze je tedy volat bez kontextu konkrétního objektu. Funkce bez návratové hodnoty jsou nazývány procedurami. Běžnými funkcemi jsou například funkce matematické, funkce pro manipulaci s textem, souborové funkce, kryptografické funkce, systémové funkce, funkce pro síťovou komunikaci atd.
- *Metodami* budeme označovat funkce a procedury, které lze vykonávat pouze v závislosti na konkrétní instanci objektu. Vykonatelnost takových funkcí úzce závisí na třídě daného objektu, což představuje část omezení pro jejich volání. Zpravidla se jedná o funkce modifikující datový obsah instance objektu, nad kterou jsou volané. *Metody* uživatel programátor implementuje sám. Může se však jednat i o systémové metody, definované globálně pro třídu *Object* tedy buďto třídu ∅ nebo třídu (*true*) podle použitého způsobu definice tříd. Takovými funkcemi mohou být například funkce vracející typ objektu, jednoznačný identifikátor instance v úložišti, počet atributů objektu, velikost objektu atd.
- *Triggery* jsou speciálním případem metod, které jsou vázané na modifikaci atributů objektu - jejich přidání, změnu či odebrání. *Triggery* mohou být volané před provedením modifikace nebo až po ní. V širším významu může *trigger* představovat metodu navázanou i na samotnou instanci objektu. Například pokud se celý objekt

posune v čase, není efektivní postupně odchytávat všechny *triggery* na změnu jeho atributů, ale je lepší provést pouze jeden *globální trigger* pro objekt. *Triggery* lze tedy definovat jak nad atributy, tak i nad objekty samotnými, přičemž provádění *globálního triggeru* zastíní provádění *triggeru* pro atribut.

- *Agregační funkcí* se rozumí funkce, která je volaná nad množinou instancí objektů (množinou referencí na objekty). Návratová hodnota agregační funkce může být atomická, může se jednat o referenci na instanci objektu a nebo o množinu referencí. Agregačními funkcemi, které vrací atomickou hodnotu, mohou být obecně známé funkce ze stávajících databázových systémů, jako je počet prvků množiny, průměrná, minimální a maximální hodnota atributů v dané množině atd. Za agregační funkce jsou zde však považovány i funkce vracející referenci na vybraného zástupce množiny, vracející množinu referencí na vybrané objekty množiny nebo agregační procedury volané na množinu objektů a nevracejících žádný výstup.

Model dovoluje automaticky zobecnit volání některých metod na agregační funkce v případě, kdy jsou místo na konkrétní objekt zavolány na množinu objektů. Metody v roli agregačních funkcí zavolaných na množině referencí provedou:

- V případě procedury iterativní volání metody na všechny instance objektů vstupní množiny, na kterých metodu lze vykonat. Jedná se o metody, které zpravidla modifikují obsah instancí objektů.
- V případě funkcí vracejících referenci nebo množinu referencí navíc sjednocení návratových hodnot do jedné výsledné množiny referencí. Takto definované výstupy lze použít jako vstup pro další agregační funkce, a tak umožňují skládáním vytvářet složitější dotazy.

Implicitně není určeno v jakém pořadí se má vykonávat metoda v roli agregační funkce volaná nad množinami referencí. Z praktických důvodů by však bylo vhodné mít možnost stanovit pořadí zpracování instancí objektů explicitně.

Omezení proveditelnosti metod lze provést dvěma způsoby v závislosti na použitém modelu implicitních tříd. V prvním případě je omezení proveditelnosti metody definované množinou požadovaných atributů daného objektu. Ve druhém případě je omezení proveditelnosti definované stanovením podmínky proveditelnosti.

4.3.1 Omezení proveditelnosti množinou atributů

V případě omezení množinou atributů je pro funkci $f \in F$ množina požadovaných atributů označená jako $req_f \subseteq A$. Metodu reprezentovanou funkcí $f \in F$ lze nad objektem $o \in O$ v časovém okamžiku $t \in T$ vykonat právě tehdy, pokud $req_f \subseteq def(o)^{(t)}$. Jinými slovy objekt *o* musí náležet třídě $req_f \subseteq A$ či jejímu potomku.

V časovém okamžiku *t* lze definovat množinu všech vyhovujících instancí objektů, na které lze funkci *f* vykonávat. Tato množina se označuje $dom_f{}^{(t)}$ dle vztahu (R 11a).

$$dom_f{}^{(t)} = \{o \in O \mid def(o)^{(t)} \supseteq req_f\} \qquad \textbf{R 11a}$$

Ze vztahu (R 11a) plyne, že pro každé dvě funkce $f, g \in F$ takové, že $req_f \subseteq req_g$ platí $dom_f{}^{(t)} \supseteq dom_g{}^{(t)}$.

Podobně lze definovat i množinu funkcí $iface(o)^{(t)}$, které lze v časovém okamžiku $t \in T$ nad instancí objektu $o \in O$ provádět. Formální definice je uvedena ve vztahu (R 12a).

$$iface_f(o)^{(t)} = \{f \in F \mid req_f \subseteq def(o)^{(t)}\} \qquad \textbf{R 12a}$$

Zápis dom_f znamená aktuální množinu všech vyhovujících instancí objektů, nad kterými lze funkci *f* volat. Zápis $iface(o)$ znamená aktuální množinu funkcí vykonatelných nad objektem *o*.

Pro implicitní třídy rozvolněných objektů v daném časovém okamžiku navíc platí vztah (R 13).

$$\forall o_i, o_j \in O, t \in T : def(o_i)^{(t)} \supseteq def(o_j)^{(t)} \Rightarrow iface(o_i)^{(t)} \supseteq iface(o_j)^{(t)} \qquad \textbf{R 13}$$

Vztah (R 13) říká, že pokud je funkce proveditelná nad instancí nějaké implicitní třídy, je proveditelná i nad všemi instancemi tříd odvozenými od této třídy.

Značení omezení proveditelnosti funkce $f \in F$ pomocí konkrétní množiny atributů bude dále označeno jako {*množina*}::*f*. V případě použití pojmenovaných tříd bude označení vypadat následovně *Třída*::*f*.

4.3.2 Omezení proveditelnosti podmínkou

V rozšířeném modelu rozvolněných objektů lze proveditelnost funkce specifikovat obecněji booleovskou podmínkou. Za použití logických spojek *and, or, not, xor, nor* atd. a závorek lze specifikovat podmínky proveditelnosti mnohem přesněji. Samotná definice implicitních tříd lze v tomto směru zobecnit na booleovskou podmínkou.

Nechť $f \in F$ je funkce, která má omezenou proveditelnost podmínkou $req_f \in C$.

V případě takto obecných definic implicitních tříd je nutné redefinovat proveditelnost funkce nad daným objektem pomocí zobrazením $req_f : O \times T \rightarrow \{true, false\}$. Funkce $f \in F$ je poté proveditelná právě tehdy, když $req_f(o)^{(t)} = true$. Množina objektů $dom_f{}^{(t)}$, na které jsou proveditelné funkce $f \in F$ v časový okamžik $t \in T$ je nyní dána nikoliv předpisem (R 11a), ale jeho modifikovaným tvarem (R 11b).

$$dom_f{}^{(t)} = \{o \in O \mid req_f(o)^{(t)} = true\} \qquad \textbf{R 11b}$$

Ze vztahu (R 11b) plyne, že pro každé dvě funkce $f, g \in F$ takové, že $req_f \Leftarrow req_g$ musí platit $dom_f{}^{(t)} \supseteq dom_g{}^{(t)}$.

Obdobně je definována i množina proveditelných funkcích na objektu $o \in O$ v časový okamžik $t \in T$. Formální definice je ve vztahu (12 b).

$$iface_f(o)^{(t)} = \{f \in F \mid req_f(o)^{(t)} = true\} \qquad \textbf{R 12b}$$

Pro každé dvě třídy rozvolněných objektů s omezením proveditelnosti podmínkami obecně vztah (R 13) neplatí a nelze stanovit ani žádnou jeho obdobu.

Značení omezení proveditelnosti funkce $f \in F$ pomocí konkrétní podmínky bude dále označeno jako (*podmínka*)::*f*. V případě použití definice podmínkami tříd bude označení vypadat shodně jako při množinové definici tj. *Třída*::*f*.

4.3.3 Zástupné metody

V objektově orientovaných jazycích mohou programátoři u jednoho typu objektu definovat více funkcí se stejným názvem. Výběr správné funkce, která se bude vykonávat, potom závisí na počtu a typu jednotlivých parametrů.

Na obrázku (Obrázek 4) je kód použití virtuálních metod explicitní hierarchie

z obrázku (Obrázek 3). V klasickém objektovém modelu není těžké zajistit, aby se zavolala metoda f ze třídy *ListItem*, takže všechny objekty v *items* budou mít hodnotu klíče *Key* rovnou hodnotě 10.

```
class Item
{
    public Item(int Key)
    {
        this.key = Key;
    }

    private int key;
    public int Key
    {
        set { key = value; }
        get { return key; }
    }

    public virtual void f()
    {
        key = 0;
    }
}
```

```
class ListItem: Item
{
    public ListItem(int Key, ListItem Next)
        : base(Key)
    {
        this.next = Next;
    }

    private ListItem next;
    public ListItem Next
    {
        set { next = value; }
        get { return next; }
    }

    public virtual void f()
    {
        key = 10;
    }
}
```

```
static void Main(string[] args)
{
    List<Item> items = new List<Item>();
    for (int i = 0; i < 10; ++i)
        items.Add(new ListItem(i, null));
    for (int i = 0; i < 10; ++i)
        items[i].f();
}
```

Obrázek 4 - ukázka implementace virtualizace

Naproti tomu i v jednodušší definici implicitních tříd pomocí množin atributů objektů má každá třída s k atributy přesně 2^k-1 předků a z toho k přímých předků. Kromě velkého počtu předků je problém i v tom, jak určit správnou hierarchii dědění tříd. Výsledkem je, že nad rozvolněnými objekty je obtížné definovat virtualizaci při volání metod. Při volání explicitně nedefinované metody *ListItem*::*f*() je proto komplikované určit, zda volat metodu {*Key*}::*f*() nebo {*Next*}::*f*(). V případě nedefinované metody *TreeItem*::*f*() by run-time musel rozhodnout o volání některé z metod {*Left*,*Right*}::*f*(), {*Left*,*Next*}::*f*(), {*Right*,*Next*}::*f*(), případně z metod {*Left*}::*f*(), {*Right*}::*f*(), {*Next*}::*f*(). To by bylo možné udělat pouze explicitním deklarováním, kterému se model snaží vyhnout. Navíc objekty v čase migrují a tedy určit aktuální třídu určitého objektu může být netriviální. Volání funkce by potom vedlo k vysoké zátěži systému.

Na druhou stranu virtualizace představuje v dnešní době velmi silný a efektivní nástroj při vývoji aplikací. Převážné využití je založeno na tom, aby funkce realizující stejné procesy měly shodná jména. Proto jsou u rozvolněných objektů definovány tzv. zástupné metody. Jsou to speciální funkce jako například makra definovaná uživateli. Pomocí maker lze efektivně

simulovat hlavní část virtualizace, kdy provedení nějaké funkce závisí na dostupnosti určitých atributů objektů. Toto může být velice výhodné i ve víceuživatelském prostředí, kdy má každý uživatel různá práva jak na objekty tak i na samotné atributy, a proto se mu objekty jeví jako zařazené do jiných tříd.

Zástupné metody dovolují explicitně určit jaká funkce se má v jakém kontextu a za jakých podmínek zavolat. Výsledkem je, že se pod jedním názvem makra může skrývat více funkcí s různým počtem i typem parametrů, protože konkrétní volání metody si uživatel určuje sám. Nicméně funkce samotné musí mít různá jména v rámci aplikace, která pracuje s daty z úložiště. Každá funkce potom může reprezentovat podobný algoritmus.

Zástupné metody zde přinášejí pro programátora jistá nebezpečí. Je totiž nutné pamatovat na všechny možnosti rozvětvení v závislosti na vstupních parametrech pro každou zástupnou metodu. V opačném případě by se aplikace nad takovým úložištěm mohla dostat do slepé větvě a to by způsobilo její nestabilitu. Rozšíření pomocí zástupných metod je tedy užitečný nástroj pro emulaci virtualizace, ale je nutné jej používat opatrně.

Například může být problematické pozdější přidání nového uživatele s novými právy na dostupnost jednotlivých atributů objektů, s jejichž kombinací autor zástupné metody nepočítal, a tak se může vybrat zbytečně neefektivní způsob realizace procesu.

4.4 Manipulace s daty

Tato sekce je věnována nutným požadavkům, které musí splňovat databázové prostředí, kde má být implementováno úložiště pro rozvolněné objekty.

4.4.1 Transakční zpracování

Pro bezpečné zpravování dat je požadováno, aby databázový systém podporoval klasické transakční zpracování dat. Pokud by tato podpora chyběla, mohlo by v ukládaných datech docházet k nekonzistencím.

4.4.2 Modifikace objektů

Každá instance jakéhokoliv rozvolněného objektu musí být nejprve explicitně vytvořena. Poté lze s objektem pracovat, čímž se myslí přidávat či odebírat atributy nebo je modifikovat. Každý objekt je považován za již nežijící, pokud v daném časovém okamžiku již nemá definovaný žádný atribut s hodnotou. Pokud je objekt zcela smazán, jsou zrušeny i

všechny jeho atributy ve všech časových okamžicích jeho života. Je proto smazán i záznam o samotné instanci objektu z katalogu.

4.4.3 Práva

Dekompozice datových objektů na jednotlivé atributy ukládané samostatně do relace má za následek velmi snadné definování práv na jednotlivé objekty. Granulitu práv lze stanovit až na úroveň jednotlivých atributů objektů a tím přesně specifikovat přístupová omezení. Tato vlastnost je ideální pro víceuživatelský databázový systém, kdy každý uživatel má mít různá práva. Kromě práv lze v DB systému navíc specifikovat i prioritu odezvy pro každou skupinu uživatelů při velkém zatížení systému.

Například skladník by měl mít možnost rychle zjistit kolik a jakých kusů produktu je na skladě v přiměřené době odezvy. Měl by mít právo přistupovat k ceně produktu bez možnosti její modifikace. Měl by však být schopen měnit názvy, označení a rovněž počty kusů pomocí naskladňovacích a vyskladňovacích operací. Ostatní informace o produktu nejsou pro skladníka zajímavé, a proto pro něj mohou být zcela skryty. Naproti tomu je nutné, aby účetní mohla modifikovat ceny a viděla všechny potřebné další údaje. Možnost utajení pomocí skrytí informace vede kromě zvýšení bezpečnosti i ke zvýšení čitelnosti výsledků, neboť uživatel není obtěžován zobrazováním pro něj nedůležitých informací. Zároveň se skrýváním informací (atributů) se automaticky znepřístupňují i metody, které tyto informace pro své vykonání vyžadují.

4.4.4 Množiny

Dotazy zpravidla vrací referenci na objekt nebo množinu referencí. Pokud je návratovou hodnotou jeden objekt či reference na instanci určitého objektu, lze s tímto výsledkem pracovat jako s množinou o jednom prvku. Pro připomenutí zavolání funkce na množině, která očekává jeden vstupní parametr, je ekvivalentní provedení funkce nad každým prvkem množiny.

Databázový systém by z tohoto ohledu měl podporovat základní množinové operace jako jsou průnik, sjednocení a rozdíl.

4.4.5 Dotazy, omezení a výsledky

Pro získávání dat se používají dotazy v datovém úložišti. Dotazem se rozumí libovolná

booleovská formule, jejíž aplikací se podle pravdivosti získá reference na instanci objektu a nebo množina referencí na objekty. V rámci množiny vrácených objektů je vhodné, aby implementace modelu nějakým způsobem umožňovala navigaci mezi objekty.

V instancích schématu modelu pro rozvolněné objekty je možné stanovit omezení pro provedení funkce buď implicitní třídou a nebo obecněji libovolnou podmínkou nad množinou atributů. V podstatě se vždy jedná o omezení proveditelnosti podmínkou, protože definování třídou pouze určuje požadovanou množinu atributů. Každá reference je v daném modelu považována za dotaz, který vrací jedinou instanci konkrétního objektu 9. Všechny dotazy lze skládat dohromady a tím vytvářet složitější dotazy, pokud si odpovídají vstupní a výstupní parametry.

Výsledkem dotazu je množina referencí na objekty, které dotazu vyhovují. Vrácená množina referencí může být i prázdná. Pouze v případě volání triggeru je vrácena žádná množina.

Dotazy lze volat nad libovolnými množinami objektů (referencí na objekty). Tedy jak nad množinou dat z celé databáze, tak i nad množinami vrácenými jinými dotazy. Celkově tak v nejobecnějším případě dostáváme řetěz jednotlivých zastupitelností, který vypadá: množina→(omezení↔dotaz)→reference.

4.5 Vztahy mezi objekty

V reálném světě nestačí pouze uchovávání informací o objektech, ale je nutné i zaznamenat jednotlivé vztahy mezi nimi. Datové úložiště pro rozvolněné objekty podporuje ukládání všech známých vztahů mezi objekty. 0:1, 1:1, 0:n, 1:n až m:n. Realizace je založena na referencích na jednotlivé instance objektů.

Jednoduché vztahy 0:1, 1:1, 0:n, 1:m tak lze snadno uchovávat jako speciální druh atributu. V klasických databázových systémech se pro tento účel používají cizí klíče. Toho lze snadno využít v modelu předkládané práce, neboť se zde předpokládá jednoznačná identifikace objektů pomocí přirozených čísel. Při modifikaci dat odkazovaných z jiných objektů je možné využít i speciální režimy chování cizích klíčů označené v klauzuli *on delete* klíčovými slovy *cascade* či *restrict*. Pokud uživatel potřebuje silnější kontrolu, lze ji nadefinovat triggerem.

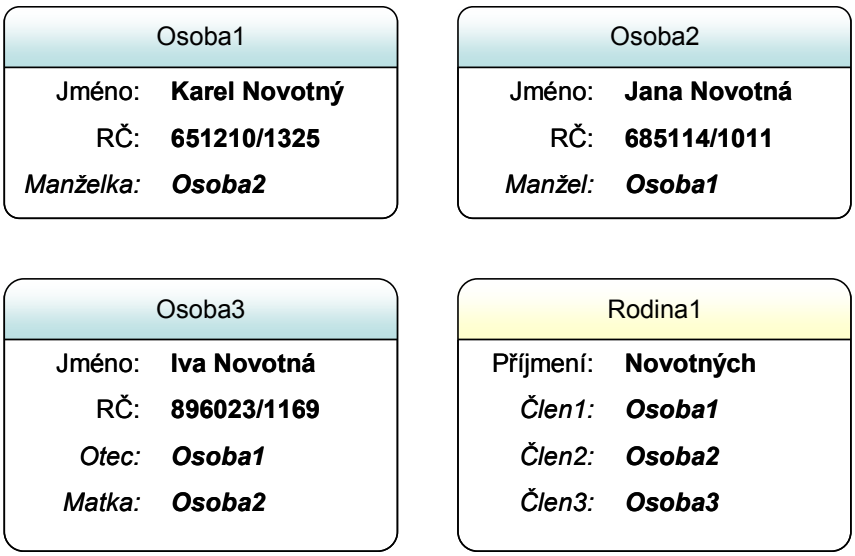
Pro případ vztahů m:n, více-árních vztahů a vztahů s doplňujícími informacemi je nutné vytvořit nový objekt, který bude vztah reprezentovat. Takové objekty budou dále v práci označené jako *objekty vztahové*. V klasických databázových systémech se pro

ukládání složitých vztahů většinou používají zvláštní relace. V modelu rozvolněných objektů se pro uchovávání složitějších vztahů používají opět rozvolněné objekty.

Vhodnost specifikování vztahu pomocí vztahových objektů je vidět na následujícím příkladu. Nechť se mají v úložišti uchovat informace o vztahu tří osob - matky, otce a dítěte. Každá osoba je pro jednoduchost definovaná několika málo údaji: jménem a rodným číslem (z rodného čísla lze odvodit pohlaví osoby, protože ženy mají k měsíci přičteno 50).

Na obrázku (Obrázek 5) je příklad s vazbami mezi objekty. Každý objekt má svůj textový identifikátor, který je jednoznačný v rámci celého úložiště (například *Osoba1*, *Rodina1*, …). Každý objekt má několik atributů, název atributu je znázorněn v levé části objektu, a hodnota pro daný atribut je vpravo a zvýrazněna tučným písmem. Atribut reference je napsán kurzívou. Pro odkazování atributu daného objektu je použita tečková notace. Například *Osoba1.RČ* je označení atribut *RČ* objektu *Osoba1*, jehož hodnota je 651210/1325.

Na modelovém příkladu z obrázku (Obrázek 5) jsou celkem tři osoby. Každá z nich nese určité informace o sobě a také alespoň jeden *referenční atribut* (odkaz na určitý objekt). Vazby atributů *Osoba1.Manželka* a *Osoba2.Manžel* jsou ukázkové vazby 1:1. V příkladu jsou znázorněné obě vazby, i když k zachycení informace manželů by stačil pouze jeden atribut z těchto dvou. Druhý vztah lze totiž z úložiště získat deduktivně. Stávající modelace manželství umožňuje uchovávat i netypické situace. Například *Osoba1* může tvrdit, že má manželku *Osobu2* a přitom *Osoba2* tvrdí, že má za manžela někoho jiného. Pokud by programátor takovou to situaci považoval za nekonzistentní a chtěl by se ji vyhnout, bylo by vhodné vytvořit vztahový objekt manželství, kde by byly dva *referenční atributy* na muže a ženu. Modelování manželství pomocí nového vztahového objektu může však přinášet jiná úskalí, jako například uložení pouze jednoho atributu či v případě uložení obou atributů (muže i ženy) shodnou hodnotu těchto atributů.



Obrázek 5 - typy vazeb mezi objekty

Atributy *Osoba3.Otec* a *Osoba3.Matka* označují, že *Osoba3* je dcera lidí *Osoba1* a *Osoba2*. V tomto případě se jedná o vazby typu 0:n. Rozlišení atributů *Otec* a *Matka* je u osoby *Osoby3* zbytečné. Stačilo by pamatovat si atributy vazby 1:n *Osoba3.Rodič1* a *Osoba3.Rodič2*, protože objekty *Osoba1* a *Osoba2* mají atribut *Pohlaví* díky, kterému lze vydedukovat o jakého rodiče se jedná, zda o matku či otce.

Poslední objekt v úložišti je *Rodina1*. Jedná se o takzvaný *vztahový objekt*, který zachycuje vztah typu 1:n a navíc obsahuje jeden informační atribut *Příjmení*. Při skutečné implementaci by v úložišti byl pravděpodobně ještě uložen atribut *Rodina1.Počet_členů*, který by udával počet *referenčních atributů* pro každou vazbu typu 1:n.

4.6 Katalog

Katalog úložiště pro rozvolněné objekty slouží stejně jako v klasických databázových systémech především k popisu vlastního úložiště. Pro uživatele je katalog souborem informací určených pouze pro čtení. Jsou v něm uložené informace dvojího druhu.

Prvním druhem jsou *interní informace o úložišti*, které slouží k efektivnější práci s úložištěm. Mohou zde být např. názvy atributů, vazby atributů na konkrétní instance objektů, temporalita atd. Při uložení těchto informací potom lze získat efektivnější úložiště.

Při evidenci těchto pomocných údajů je možné snadno zjistit, jaké atributy daný objekt v určitý časový okamžik má či nikoliv. Lze snadno zjistit čas narození a úmrtí objektu atd. Tyto pomocné informace jsou zpravidla uložené redundantně a lze je z úložiště vydedukovat. Kromě poskytování informací bývají tato data využívaná pro zefektivnění manipulujících

operací nad objekty.

Druhým druhem jsou *externí informace*. Sem patří informace o vlastním úložišti. Těmi mohou být např. parametry, se kterými byla daná instance úložiště vytvořena a které se po jeho vytvoření již více nemění.

Katalog je tedy nejen užitečná část úložiště, ale také část nutná. Kromě toho, že se jedná o zdroj informací o vytvořeném schématu úložiště pro uživatele, dovoluje jeho vhodná organizace efektivně realizovat požadovanou funkčnost úložiště.

Přesná data v katalogu a jejich formát závisí na konkrétní implementaci. V modelu rozvolněných objektů lze katalog navenek realizovat jako sadu rozvolněných objektů. Tato reprezentace mimo jiné dovolí využít dotazovací prostředky úložiště pro získávání informací o úložišti stejným způsobem jako o datech aplikace samotné. Programátor tedy nemusí rozlišovat odkud zpracovávaná data pocházejí.

4.7 Okrajové podmínky

Tato sekce je věnována nejen okrajovým podmínkám, ale i doplňujícím informacím, které je nutné znát před vlastním používáním úložiště pro rozvolněné objekty.

4.7.1 Multi atributy

V modelu rozvolněných objektů je nutné, aby všechny různé atributy měly své unikátní jméno, jinak nebude implementace schématu úložiště správně fungovat. Problémy s použitím mohou nastat při použití špatného návrhu, kdy se uživatel pokusí využít temporálních atributů k vyjádření situace, že v jednu dobu měl objekt přiřazeny dvě a více hodnot nějakého atributu. Představme si situaci, že vrátný *Josef* z příkladu v sekci (3.4) v roce 1995 získal řidičské oprávnění skupiny A. O rok později, tedy v roce 1996, získal ještě řidičské oprávnění skupiny B. Za další rok byl vrátnému řidičský průkaz odebrán.

Objekt	ŘP	Platí_od	Platí_do
Josef	*A*	*1995-01-01*	1997-01-01
Josef	B	*1996-01-01*	*1997-01-01*

Objekt	ŘP	Platí_od	Platí_do
Josef	*A*	*1995-01-01*	1996-01-01
Josef	AB	*1996-01-01*	*1997-01-01*

Tabulka 3 - ukázka multi-atributů

V tabulce (Tabulka 3) jsou shodné hodnoty atributů označeny kurzívou a méně výraznou barvou, aby čtenář mohl snáze rozlišit měnící se hodnoty atributů. V levé části

tabulky je zachycen problém s nesprávným použitím temporality, neboť časové intervaly platnosti nejsou disjunktní, což může být při indexování problém. V pravé části tabulky je možné řešení tohoto konkrétního problému bez deklarace nové třídy a to tak, že se hodnoty uloží do jediné buňky. Dalším řešením tohoto problému je změna návrhu modelu. Do schématu lze přidat dva atributy *ŘP_A* a *ŘP_B* místo atributu jediného.

Ne všechny kolize ukládání více hodnot pro jeden atribut lze vyřešit uložením více hodnot do jednoho atributu. Nejobecnějším řešením je přidat novou vztahovou třídu *Oprávnění* s dvojicí atributů *RČ* a *Skupina* bude modelovat vztah mezi osobami a skupinami řidičských oprávnění.

4.7.2 Temporalita a redundance

Přidání temporality do modelu umožňuje lépe uchovávat informace o objektech z reálného světa, avšak toto rozšíření přináší nové problémy. Asi největším problémem je nárůst množství uchovávaných dat, protože velikost úložiště s časem roste a manipulace s daty se stává méně efektivní. Ač se na první pohled zdá, že nadbytečné redundanci a tím i růstu velikosti úložiště se lze vyhnout vhodným návrhem úložiště, není to pravda. Důvodem je nutnost data, včetně historických, uchovávat *„na jednom“* místě. Zvláště je nárůst dat patrný na indexech, které jsou zpravidla řešeny jako B-stromy či R-stromy.

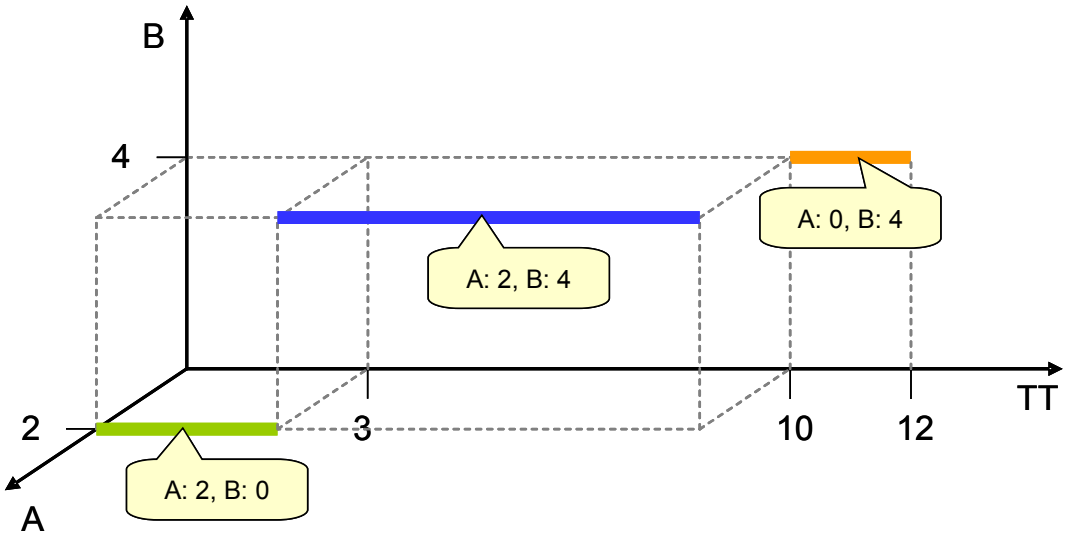
V tabulce (Tabulka 4) jsou zaznamenány hodnoty pro dva atributy A a B. Atribut A má definovánu hodnotu 2 v intervalu od 0 do 10 a atribut B má definovánu hodnotu 4 v intervalu od 3 do 12. Pro označení nedefinované hodnoty atributu se v tomto příkladu používá hodnota 0 místo zavedeného *null*. Důvodem používání 0 místo *null* je přehlednější znázornění problému obrázkem.

Atribut	Hodnota	Platí_od	Platí_do
A	2	0	10
B	4	3	12

Tabulka 4 - hodnoty atributů pro příklad redundance

Prostorová reprezentace dat z tabulky (Tabulka 4) je znázorněna na obrázku (Obrázek 6). Na obrázku jsou osy odpovídající atributům označeny A, B a osa časové složky označená jako TT. Doba existence atributu A je reprezentován zelenou (A: 2, B: 0) a modrou (A: 2, B: 4) úsečkou. Doba platnosti atributu B je zakreslena pomocí modré (A: 2, B: 4) a oranžové (A: 0, B: 4) úsečky. Je vidět, že pro dva atributy A a B s překrývajícími se intervaly platnosti jsou

v prostorové reprezentaci potřebné tři úsečky.



Obrázek 6 - prostorová reprezentace redundance

Při práci s více různými indexovanými atributy tak díky temporální redundanci velikost indexu neúměrně narůstá. Redundance vzniká, když přidaná hodnota nějakého atributu dělí určitou nadrovinu. Obecně pro k různých hodnot atributů s různými intervaly času platnosti může vzniknout až $2k$-1 nadrovin. Tento fakt je nutné brát v potaz při tvorbě indexů, protože zvětšení množiny indexovaných atributů může vést až ke dvojnásobné paměťové režii pro každý index.

4.8 Mapování úložiště

V této sekci se primárně používají definice ze sekce (4.2). Tedy rozvolněný objekt je podobný objektu v jakémkoliv jiném objektově orientovaném programovacím jazyce. Největší rozdíl se týká toho, že informace o objektu (atributy) se ukládají samostatně do relace. Jméno každého atributu v rámci úložiště musí být přitom unikátní.

Mapování úložiště z navržené abstrakce do reálného databázového systému lze formálně popsat rozdělením úložiště do několika relací. Přesný počet nutných relací bude diskutován v této kapitole. Pro větší efektivitu se však při implementaci doporučuje zahrnout i katalog (4.6). Katalog je vhodné implementovat jako část úložiště pro rozvolněné objekty. Přesný počet nutných relací v úložišti vzhledem k zahrnutí katalogu závisí jak na návrhu programátora, tak i na cílovém databázovém systému. Proto se bude následný formální popis zabývat jenom nutným počtem a strukturou relací pro mapování.

Mapování úložiště do reálného prostředí musí zachovat základní výhodu modelu

rozvolněných objektů, což je snadné přidání či smazání jakékoliv informace o objektech. Horizontální způsob ukládání je nevhodný, protože po přidání atributu je nutné pro každou potenciální třídu vytvořit relaci. Při mazání je nutné zase smazat všechny relace tříd, které mohly daný atribut obsahovat. Navíc je možné, že hodně relací bude prázdných a manipulace s nimi by byla zbytečná. Vertikální mapování nelze použít, protože u rozvolněných objektů nemusí být třídy předem známé. Navíc po přidání jednoho atributu by se mohl počet relací v úložišti rozrůst až na dvojnásobek. Způsob filtrovaného mapování je obecně pro ukládání rozvolněných a zvláště pak pro temporální varianty rozvolněných objektů nevhodné, protože hlavní tabulka bude typicky obsahovat velmi velké množství *null* hodnot a při změně hodnoty atributu bude nutné zkopírovat celý záznam v relaci.

Nutnost efektivního přidání nebo mazaní atributu jednoznačně určuje, jaké mapování je nutné použít. Z výše popsaných důvodů není vhodné horizontální, vertikální ani filtrované mapování. Nejvhodnější je ukládání rozvolněných objektů samostatně do relací, kdy každá relace reprezentuje právě jeden atribut. Tedy v každé tabulce je uložena množina hodnot atributu všech objektů v jakýkoliv časový okamžik.

Předpokládá se tedy ukládání atributů samostatně vždy do jedné relace. Doména hodnot $D_i \in D$ pro každý atribut $a_i \in A$ jde definovat v cílovém databázovém systému určitým ekvivalentním typem, nechť je to typ $map_D(D_i)$.

Pro identifikaci objektů je ve skutečném světě nutné použít pro každou instanci objektu unikátní identifikátor. V modelu rozvolněných objektů je i identifikátor dalším atributem daného objektu. Identifikátor je v předkládaném modelu reprezentován přirozeným číslem. V praxi může mít každá instance jeden i více dalších přirozených identifikátorů. Například osoby mohou být identifikované rodným číslem nebo otiskem prstu, automobily mohou být identifikovány pomocí RZ (SPZ) nebo VIN karoserie atd.

Mapování map_A jednotlivých atributů a_i se poté realizuje následovně:

$$map_A(I) = R_0(ID : map_D(I)) \qquad \textbf{R 14}$$

$$map_A(a_i) = R_i(ID : map_D(I), A_i : map_D(I)) \qquad \textbf{R 15}$$

$$map_A(a_i) = R_i(ID : map_D(I), A_i : map_D(D_i)) \qquad \textbf{R 16}$$

Výsledkem je k+1 relací $R_0, R_1, R_2, \ldots, R_k$. Vztah (R 14) definuje nutnou nultou nebo-li kořenovou relaci pro jednoznačnou identifikaci objektů. Vztahy (R 15) a (R 16) definují mapování jednotlivých atributů na skutečné relace v použitém databázovém systému. Pokud je atribut typu reference, je mapován na relace pomocí vztahu (R 15). Mapování ostatních atributů se řídí vztahem (R 16).

Mapování map_A rozdělí data v instanci schématu do k+1 relací. Vyhledávání a provádění operací s objekty jako s celky se jeví jako velmi neefektivní, protože pro každou libovolnou operaci definovanou nad k atributy je nutné spojovat k+1 relací v databázi, tj. včetně tabulky s identifikačním atributem.

Praktické ukázky a pozorování ukazují v pozdější kapitole měření výkonu (6), že s objekty jako s celky se pracuje jenom zřídka. Ve většině zkoumaných případů aplikace pracují jenom s velmi malou podmnožinou atributů objektů a tedy není vždy potřeba spojovat tolik tabulek.

Problémem implementace do existujících databázových systémů je potom začlenění indexů do takového úložiště. Současné systémy zpravidla nepodporují indexy přes více relací. Problém bývá i v případě jediné relace v používání složených indexů, které jsou zpravidla založené na struktuře B-stromů s řetězenými klíči, takže nejsou vhodné na intervalové dotazy přes libovolné sloupce. V mé implementaci modelu do reálného prostředí je proto zahrnuta i implementace odpovídajícího externího indexu, založeného na R-stromech a zohledňujícího temporalitu uložených dat.

Protože nedefinované atributy nejsou ukládány, při vyhledávání objektů není nutné ošetřovat problémy s prázdnými hodnotami, viz 9.

Při implementaci je snazší a efektivnější zavádět integritní omezení na úrovni databázového systému než na nějaké vyšší vrstvě abstrakce.

5 Implementace

Při implementaci jsem se zaměřil především na operace manipulující s rozvolněnými objekty a na efektivní vyhledávání. Implementace úložiště umožňuje pracovat s objekty náležícími více třídám najednou a objektům, které v průběhu času mezi objekty migrují. Při návrhu implementace jsem v první fázi vycházel především z problémů stanovených v motivačním příkladě (1.1). V dalších fázích jsem se soustředil především na efektivitu implementovaného úložiště.

Pro efektivní vyhledávání je úložiště obohaceno o indexy nad rozvolněnými objekty. Z důvodu ukládání rozvolněných objektů samostatně po jednotlivých atributech, bylo nutné implementovat vlastní indexy, protože stávající databázové systémy nepodporují vytváření indexů přes více různých tabulek.

5.1 Programovací jazyk

Úložiště rozvolněných objektů jsem se rozhodl implementovat v databázovém systému Oracle, a to z následujících důvodů:

- V praxi je tento systém hodně rozšířený a používaný.
- V systému Oracle lze nutná rozšíření pro rozvolněné objekty doimplementovat pomocí procedurálního rozšíření a díky existující podpoře pro uživatelsky definované typy a indexy.
- Dokumentace Oracle je dostatečně podrobná a obsahuje i výukové příklady.

Ačkoli implementace úložiště v rámci této diplomové práce byla ovlivněna výběrem databázového systému, snažil jsem se tento vliv minimalizovat. Moduly implementovaného úložiště jsou psány v jazyce PL/SQL[4], C[5] a C++[6]. PL/SQL je překládaný jazyk, vykonávaný přímo jádrem databázového serveru Oracle. Navíc lze v PL/SQL snadno manipulovat s databázovými objekty. V tomto jazyce jsou psány balíky s procedurami a funkcemi, které umožňují pracovat s úložištěm rozvolněných objektů ať při návrhu struktury úložiště tak i při samotné manipulaci s obsahem úložiště. V PL/SQL je tedy psáno celé uživatelské rozhraní

[4] Procedural Language/Structured Query Language je procedurální nadstavba jazyka SQL od firmy Oracle založená na programovacím jazyku Ada.
[5] C je nízkoúrovňový, kompilovaný programovací jazyk, který vyvinuli Ken Thompson a Dennis Ritchie.
[6] C++ je objektově orientovaný programovací jazyk, který vyvinul Bjarne Stroustrup a další v Bellových laboratořích AT&T rozšířením jazyka C.

implementovaného úložiště. Naproti tomu v jazyce C++ je naprogramováno vytváření, úprava a vyhledávání v datových strukturách jednoho druhu implementovaného indexu. Propojení aplikace v jazyce C, C++ s databázovým serverem je zprostředkováno pomocí rozhraní OCI[7] a OCCI[8].

První typ indexu je v úložišti rozvolněných objektů implementován výhradně s použitím jazyka PL/SQL. Hlavní výhodou tohoto řešení je přenositelnost. Všechny funkce a procedury je možné spustit na libovolné platformě, pro kterou je Oracle server dostupný. Je pravdou, že kód psaný v C, C++ bez systémových volání by měl být přeložitelný i na jiných systémech, než pro které byl původně určen, ale v praxi se ukazuje, že zde mohou nastat určité komplikace během překladu.

Přestože lze v jazyce PL/SQL napsat téměř jakoukoliv proceduru, potřebnou pro úložiště rozvolněných objektů v rámci databáze Oracle, ukázalo se, že výkonnostně je PL/SQL mnohem pomalejší něž C, C++. Ekvivalentní algoritmus implementovaný v C, C++ byl místy i řádově rychlejší než stejný algoritmus psaný v PL/SQL. Na druhou stranu je spouštění externích procedur psaných v jazyce C a C++ z Oracle časově velmi drahé. Z tohoto důvodu jsem se v rámci implementace snažil počet volání externích procedur z PL/SQL minimalizovat.

5.2 Databázový server

Implementace je vytvořena pro databázový systém Oracle Database 10g. Tento server je nabízen také v odlehčené verzi Express Edition, která je volně dostupná jak pro studijní, tak i pro komerční použití.

Ve verzi Express Edition, kterou jsem použil, je dostupná plná podpora indexování celých tabulek. Indexování partition tabulek je však omezeno.

5.3 Oracle data cartridge

Databázový server Oracle umožňuje implementaci tzv. *data cartridge*, pomocí kterých lze rozšířit schopnosti databáze o novou funkcionalitu 9. Lze tak například vytvořit nový datový typ a definovat způsob, jak mají být data interpretována a ukládána. *Data cartridge*

[7] Oracle Call Interface je rozhraní pomocí něhož lze programovat externí moduly pro Oracle v jazycích C, C++, Java, Rubby, PHP, Python, Perl, .Net, VB a dalších.
[8] Oracle C++ Call Interface je API, které zprostředkovává aplikacím psaným v C++ přístup k datům v databázi Oracle.

obsahuje balík funkcí a procedur, které definují požadované chování datového typu. Tento balík se instaluje do databázového serveru, kde bude uložen. Všechny jeho funkce a procedury budou spouštěny na serveru.

V databázovém serveru Oracle je rovněž definováno rozhraní, pomocí kterého lze implementovat funkce potřebné pro indexaci nového datového typu. Implementací tohoto rozhraní se vytvoří objekt, který se nazývá *indextype*. Rozhraní indexu může být implementováno pomocí jazyka PL/SQL, Java, C nebo C++. Oracle nazývá indexy, vytvořené s pomocí tohoto rozhraní, *indexy doménovými*. Tyto indexy se používají a vytvářejí prostřednictvím SQL příkazů, podobně jako běžné indexy.

Další částí *data cartridge* může být tzv. *statictics type*, což je rozšíření databázového optimalizátoru. Pomocí této součásti lze definovat funkce pro výpočet statistiky, selektivity a odhady časové náročnosti vyhledávání v indexu. Optimalizátor pak používá tyto funkce při rozhodování, zda má být použit *doménový index* nebo zda raději použít index jiný. Podrobně diskutováno v sekci (6.2).

5.4 Použité vývojové nástroje

V této sekci je kompletní přehled použitého softwaru během implementace diplomové práce.

Operační systém:	Windows XP SP3 CZ
Databázový server:	Oracle Express Edition 10.2.0.1.0 pro Win32
SQL editor:	Oracle SQL Developer 1.5.1
Java:	Java JDK 1.6.0-10
C, C++ editor a kompilátor:	Microsoft® Visual Studio 2005 Team Suite
Textový editor:	Microsoft® Office Word 2003 (Office Profesional)
Grafický editor:	Microsoft® Office PowerPoint 2003 (Office Profesional)
Textový editor:	PSPad 4.5.3

5.5 Struktura relací úložiště

Tato sekce je věnována především popisu struktury celého úložiště, především potom vztahy mezi použitými relacemi tabulek.

5.5.1 Definice typů a časové složky

Z teoretické kapitoly (4), kde je popsáno více různých pojetí rozvolněných objektů, je

implementovaná právě ta část, ve které jsou třídy definované množinou atributů. Tedy aktuální datový obsah každé instance jednoznačně určuje třídu objektu. Pro jednoznačnou identifikaci objektů jsou použita čísla z oboru přirozených čísel.

Úložiště bylo implementováno s lineární temporální složkou s diskrétní hustotou. Pro časovou složku byl použit vestavěný typ Oraclu *TimeStamp*. Hlavní důvod pro vybrání tohoto typu je jeho velká přesnost až na tisíciny milisekund. Navíc je *TimeStamp* synchronní přes jednotlivé časové zóny.

Intervalem času platnosti se rozumí dvojice časových razítek [*VALID_FROM*, *VALID_TO*).

5.5.2 Katalog implementace

Katalog v implementaci předkládaného úložiště obsahuje několik základních tabulek, zejména to jsou tabulky *SYS_ATTRIBUTES, SYS_CLASS_ATTRIBUTE, SYS_CLASSES, SYS_INDEX_ATTRIBUTE, SYS_INDEXES, SYS_OBJECT_ATTRIBUTE* a *SYS_OBJECTS*. Všechny tabulky z katalogu mají prefix „*SYS_*“. Prefix byl zaveden pro uživatele pro přehlednější rozeznání tabulek katalogu a tabulek atributů. Jména tabulek katalogu jsou uložená v balíku *RS_INTER* a uživateli jsou přístupná ze stejnojmenných funkcí tohoto balíku. Hodnoty v těchto tabulkách jsou modifikovány pouze úložištěm a pro uživatele jsou pouze pro čtení. Všechny tabulky katalogu mají podobnou strukturu jako tabulky pro atributy rozvolněných objektů. Tabulky katalogu i tabulky atributů jsou uloženy všechny v jednom schématu.

SYS_OBJECTS je tabulka, ve které jsou uloženy všechny jednoznačné identifikátory instancí objektů. Hodnota ve sloupci *IDENTIFICATOR* je nepovinný unikátní textový popisek objektu. Pokud při vytváření není *IDENTIFICATOR* vyplněný je jeho hodnota automaticky nastavena na *OBJECT_ID*. Hodnota ve sloupci *VALID_FROM* je čas prvního narození objektu. Naopak hodnota ve sloupci *VALID_TO* je čas finálního úmrtí. Pokud instance objektu nemá dosud definovaný žádný atribut jsou hodnoty sloupců *VALID_FROM* a *VALID_TO* prázdné, tedy *null*. Interval času platnosti v této tabulce je obdoba signatury času života objektu, tedy objekt v některých časových okamžicích z intervalu nemusí být živý. Schéma tabulky *SYS_OBJECTS* je popsáno v tabulce (Tabulka 5).

SYS_OBJECTS			
Sloupec	Typ	Vlastnosti	Význam
OBJECT_ID	NUMBER	Nenulový, primární klíč	Jednoznačný identifikátor pro každý objekt v úložišti

IDENTIFICATOR	VARCHAR2(64)	Nenulový, unikátní	Textový popisek objektu
VALID_FROM	TIMESTAMP(6)		Čas platnosti od
VALID_TO	TIMESTAMP(6)		Čas platnosti do

Tabulka 5 – schéma tabulky katalogu *SYS_OBJECTS*

SYS_ATTRIBUTES je tabulka, ve které jsou uložena všechna jména definovaných atributů v úložišti. Definované atributy mohou být buď *obecné atributy* nebo *atributy reference*. V případě definování *atributu reference* má buňka pro daný atribut ve sloupci *TYPE* hodnotu rovnou *ATTR_REF* (9), jinak je v buňce uložen typ atributu. Hodnota ve sloupci *VALID_FROM* je minimální hodnota ze všech časů platnosti v rámci celého úložiště pro daný atribut. Hodnota ve sloupci *VALID_TO* je naopak maximální hodnota ze všech časů platnosti v rámci celého úložiště pro daný atribut. Pokud v úložišti neexistuje ani jedna instance objektu s daným atributem, jsou poté hodnoty sloupců *VALID_FROM* a *VALID_TO* prázdné, tedy *null*. Schéma *SYS_ATTRIBUTES* je popsáno v tabulce (Tabulka 6).

SYS_ATTRIBUTES			
Sloupec	Typ	Vlastnosti	Význam
ATTR_ID	NUMBER	Nenulový, primární klíč	Jednoznačný identifikátor pro každý atribut v úložišti
NAME	VARCHAR2(30)	Nenulový, unikátní	Jméno atributu
TYPE	VARCHAR2(32)	Nenulový	Typ daného atributu nebo REFERENCE
VALID_FROM	TIMESTAMP(6)		Čas platnosti od
VALID_TO	TIMESTAMP(6)		Čas platnosti do

Tabulka 6 – schéma tabulky katalogu *SYS_ATTRIBUTES*

SYS_CLASSES je tabulka, ve které jsou uložena všechna jména pojmenovaných tříd. Tabulka po nainstalování úložiště obsahuje pouze jednu předdefinovanou pojmenovanou třídu *OBJECT*. To je třída pro objekty, které nemají definované žádné atributy. Ve sloupci *ATTR_NAMES* jsou uložena jména atributů, které definují danou pojmenovanou třídu v textové podobně „[jméno atributu][jméno atributu][jméno atributu]…". Ve sloupci *ATTR_MASK* je maska pojmenované třídy. Hodnota masky je v textové podobě „%[jméno atributu]%[jméno atributu]%[jméno atributu]%…" a primárně vychází z *ATTR_NAMES*. Jména atributů jsou jak ve sloupci *ATTR_NAMES* tak i ve sloupci *ATTR_MASK* seřazeny lexikograficky. Maska je řetězec pomocí něhož a operátoru *like* lze určit hierarchii pojmenovaných tříd. Ze sloupce *ATTR_MASK* lze pomocí funkce *replace* získat hodnotu *ATTR_NAMES,* ale i naopak. V tabulce *SYS_CLASSES* je sloupec *ATTR_NAMES* uložen redundantně, neboť je pro uživatele čitelnější než maska. Schéma *SYS_CLASSES* je popsáno v tabulce (Tabulka 7).

SYS_CLASSES			
Sloupec	Typ	Vlastnosti	Význam
CLASS_ID	NUMBER	Nenulový, primární klíč	Jednoznačný identifikátor pro každou pojmenovanou třídu v úložišti
NAME	VARCHAR2(30)	Nenulový, unikátní	Jméno třídy
ATTR_NAMES	VARCHAR2(2000)	Nenulový	Jména atributů, kterými je třída definována
ATTR_MASK	VARCHAR2(2000)	Nenulový	Maska z atributů třídy

Tabulka 7 – schéma tabulky katalogu *SYS_CLASSES*

SYS_INDEXES je tabulka, ve které jsou uložena všechna jména indexů nad tabulkami atributů. Hodnota 1 ve sloupci *LEGAL* indikuje, zda se jedná o kompletně vytvořený index tedy o *plný index*. Jinak hovoříme o takzvaném *abstraktním indexu*, který nebude během vyhledávání použit. V předkládané diplomové práci byly implementovány dva druhy indexů. Ve sloupci *RS_INDEX_TYPE* je uloženo o jaký druh indexu se jedná. Hodnota *INX_TYPE_RO* (9) sloupce *RS_INDX_TYPE* indikuje *externí index*, jinak je použit interní index Oracle (9). Schéma *SYS_INDEXES* je popsáno v tabulce (Tabulka 8).

SYS_INDEXES			
Sloupec	Typ	Vlastnosti	Význam
INDEX_ID	NUMBER	Nenulový, primární klíč	Jednoznačný identifikátor pro každý index v úložišti
NAME	VARCHAR2(30)	Nenulový, unikátní	Jméno indexu
LEGAL	NUMBER(1,0)	Nenulový, hodnoty jen 0, 1	Flag, který určuje, zda je index již plně vytvořený a lze jej používat
RS_INDEX_TYPE	NUMBER(1,0)	Hodnoty jen 0, 1	Flag druhu indexu

Tabulka 8 – schéma tabulky katalogu *SYS_INDEXES*

SYS_OBJECT_ATTRIBUTE je vazební tabulka mezi objekty a atributy. Pro dvojici určité instance objektu a určitého atributu je v tabulce nanejvýš jeden záznam, pokud má daná instance definovaný daný atribut. Hodnota ve sloupci *VALID_FROM* je minimální hodnota ze všech časů platnosti pro dvojici objektu a atributu. Hodnota ve sloupci *VALID_TO* je naopak maximální hodnota ze všech časů pro dvojici objektu a atributu. Z této tabulky lze snadno vyčíst, jaké atributy by mohl mít určitý objekt definované. Interval času platnosti v této tabulce daného atributu pro konkrétní instanci objektu je obdoba signatury. Tedy určitá instance objektu nemusí mít definovaný daný atribut v některý časový okamžik z intervalu signatury. Schéma *SYS_OBJECT_ATTRIBUTE* je popsáno v tabulce (Tabulka 9).

SYS_OBJECT_ATTRIBUTE			
Sloupec	Typ	Vlastnosti	Význam
OBJECT_ID	NUMBER	Nenulový, cizí klíč do tabulky SYS_OBJECTS	Jednoznačný identifikátor objektu

ATTR_ID	NUMBER	Nenulový, cizí klíč do tabulky SYS_ATTRIBUTES	Jednoznační identifikátor atributu
VALID_FROM	TIMESTAMP(6)	Nenulový	Čas platnosti od
VALID_TO	TIMESTAMP(6)	Nenulový	Čas platnosti do

Tabulka 9 – schéma vazební tabulky katalogu *SYS_OBJECT_ATTRIBUTE*

SYS_CLASS_ATTRIBUTE je vazební tabulka mezi atributy a pojmenovanými třídami. Schéma *SYS_CLASS_ATTRIBUTE* je popsáno v tabulce (Tabulka 10).

SYS_CLASS_ATTRIBUTE			
Sloupec	Typ	Vlastnosti	Význam
CLASS_ID	NUMBER	Nenulový, cizí klíč do tabulky SYS_CLASSES	Jednoznačný identifikátor pojmenované třídy
ATTR_ID	NUMBER	Nenulový, cizí klíč do tabulky SYS_ATTRIBUTES	Jednoznační identifikátor atributu

Tabulka 10 – schéma vazební tabulky katalogu *SYS_CLASS_ATTRIBUTE*

SYS_INDEX_ATTRIBUTE je vazební tabulka mezi indexy a atributy. Schéma *SYS_INDEX_ATTRIBUTE* je popsáno v tabulce (Tabulka 11).

SYS_INDEX_ATTRIBUTE			
Sloupec	Typ	Vlastnosti	Význam
INDEX_ID	NUMBER	Nenulový, cizí klíč do tabulky SYS_INDEXES	Jednoznačný identifikátor indexu
ATTR_ID	NUMBER	Nenulový, cizí klíč do tabulky SYS_ATTRIBUTES	Jednoznačný identifikátor atributu

Tabulka 11 – schéma vazební tabulky katalogu *SYS_INDEX_ATTRIBUTE*

5.5.3 Atributy

Jak již bylo zmiňováno, hodnoty každého atributu jsou uloženy pro všechny objekty v odpovídající tabulce. Jméno tabulky atributu je ekvivalentní jménu atributu. Jméno atributu proto musí být v rámci celého úložiště unikátní a musí být jiné než rezervovaný název, což jsou například jména tabulek z katalogu, ale i další. Atributy mohou být dvojího druhu buď *obecný atribut* a nebo *atribut reference*. Všechny tabulky atributů jsou pro uživatele pouze pro čtení. Hodnoty v tabulkách atributů jsou modifikovány pomocí procedur a funkcí z balíků úložiště.

Nechť *AA* je jméno *obecného atributu.* Hodnota každého atributu pro daný objekt je uložena ve sloupci *VALUE*. Hodnota sloupce *VALUE* může být téměř libovolná. Typ textového řetězce byl pro typ atributu zvolen úmyslně, protože usnadňuje značnou mírou implementaci. Zvolení textového řetězce jako jediného základního typu není omezující, neboť se jedná o nejobecnější typ a jakýkoliv vestavěný typ v Oracle lze implicitně zkonvertovat na text. Hodnota sloupce *VALUE,* v implementaci předkládané diplomové práce, nesmí být *null.* V případě, že se uživatel pokouší uložit do sloupce *VALUE* delší řetězec než 256 znaků bude ukládaná hodnota zkrácena na maximální délku 256 znaků. (diskutováno v 5.8.9) Sloupce *VALID_FROM* a *VALID_TO* určují disjunktní intervaly platnosti hodnoty definovaného atributu na instanci objektu. Pro dvojici objektu a hodnoty atributu je v tabulce atributu právě tolik záznamů, kolik existuje disjunktních intervalů času platnosti pro každou dvojici. Schéma tabulky atributu *AA* je popsáno v tabulce (Tabulka 12).

AA			
Sloupec	Typ	Vlastnosti	Význam
OBJECT_ID	NUMBER	Nenulový, cizí klíč do tabulky SYS_OBJECTS	Jednoznačný identifikátor objektu
VALUE	VARCHAR2(256)		Hodnota obecného atributu
VALID_FROM	TIMESTAMP(6)	Nenulový	Čas platnosti od
VALID_TO	TIMESTAMP(6)	Nenulový	Čas platnosti do

Tabulka 12 – ukázkové schéma tabulky *obecného atributu AA*

Nechť *BB* je jméno *atributu reference.* Hodnota každého *referenčního atributu* je uložena ve sloupci *VALUE*. Hodnota *VALUE* musí být číslo instance objektu. *VALUE* je cizí klíč do tabulky *SYS_OBJECTS,* a tedy odkazovat se lze pouze na existující instance objektů. Sloupce *VALID_FROM* a *VALID_TO* určují disjunktní intervaly platnosti hodnoty odkazu definovaného atributu na instanci objektu. Pro dvojici objektu a hodnoty atributu je v tabulce právě tolik záznamů, kolik existuje disjunktních intervalů času platnosti pro každou dvojici. Schéma tabulky atributu reference *BB* je popsáno v tabulce (Tabulka 13).

BB			
Sloupec	Typ	Vlastnosti	Význam
OBJECT_ID	NUMBER	Nenulový, cizí klíč do tabulky SYS_OBJECTS	Jednoznačný identifikátor objektu
VALUE	NUMBER	Cizí klíč do tabulky SYS_OBJECTS	Hodnota reference atributu
VALID_FROM	TIMESTAMP(6)	Nenulový	Čas platnosti od
VALID_TO	TIMESTAMP(6)	Nenulový	Čas platnosti do

Tabulka 13 – ukázkové schéma tabulky *referenčního atributu BB*

5.5.4 Signatury a intervaly časů platnosti

V implementovaném modelu se *časovým intervalem* rozumí dvojice časových razítek [VALID_FROM, VALID_TO) s podmínkou *VALID_FROM < VALID_TO*. Spodní hranice intervalu *VALID_FROM* je uzavřená naopak horní hranice *VALID_TO* je otevřená. Uzavřená spodní hranice a otevřená horní hranice *časových intervalů* umožňuje *časové intervaly* spojitě skládat za sebe.

Intervalem času platnosti se u relací *obecných atributů* či *referenčních atributů* rozumí *časový interval* uvedený v daném záznamu tabulky. Procedurální rozhraní nad úložištěm pro uživatele samo zajišťuje, že pro každou dvojici objektu a atributu budou *intervaly časů platnosti* disjunktní. Tento požadavek se využívá při případné indexaci.

Signaturou času platnosti se rozumí nejmenší *časový interval*, pokrývající danou množinu časových intervalů. Signatury se nacházejí například v tabulkách katalogu *SYS_OBJECTS, SYS_ATTRIBUTES* a *SYS_OBJECT_ATTRIBUTE*. Signatura je dvojicí minimálních a maximálních hodnot časů platnosti z dané množiny intervalů (závisí na definici tabulky katalogu). Signatura určuje nutnou, nikoli postačující podmínku času platnosti. Signatura tedy omezí prohledávanou množinu záznamů.

Ukázka času platnosti je uvedená v tabulce (Tabulka 14), kde je znázorněn atribut *AA* s hodnotou *A1* platnou v intervalu [1.1.1994, 1.1.1995), A2 platnou v intervalu [1.1.1995, 1.1.1996) a hodnotou *A3* platnou v intervalu [1.1.1999, 1.1.2001).

AA			
OBJECT_ID	VALUE	VALID_FROM	VALID_TO
1	A1	1994-01-01	1995-01-01
1	A2	1995-01-01	1996-01-01
1	A3	1999-01-01	2001-01-01

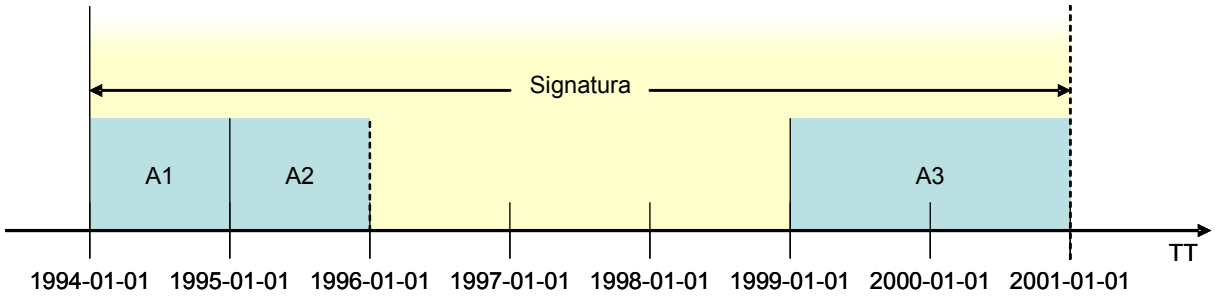
Tabulka 14 - příklad intervalů času platnosti atributu *AA*

Signatura intervalu platnosti atributu *AA* pro objekt 1 z příkladu (Tabulka 14) bude například v tabulce katalogu *SYS_OBJECT_ATTRIBUTE* vypadat podle tabulky (Tabulka 15). V modelovém příkladě je atribut *AA* identifikován číslem 1. Hodnota signatury je [1.1.1994, 1.1.2001).

SYS_OBJECT_ATTRIBUTE			
OBJECT_ID	ATTR_ID	VALID_FROM	VALID_TO
1	1	1994-01-01	2001-01-01

Tabulka 15 - příklad signatury času platnosti atributu *AA* instance objektu 1

Grafický rozdíl mezi *intervalem času platnosti* z tabulky (Tabulka 14) a *signaturou času platnosti* z tabulky (Tabulka 15) je znázorněn na obrázku (Obrázek 7). Uzavřené hranice intervalů jsou vykreslené plnou čárou, otevřené hranice jsou čárou přerušovanou.



Obrázek 7 - rozdíl mezi intervalem času platnosti a signaturou času platnosti

5.5.5 Koncept ovládání úložiště

V úložišti je pět základních balíků, pomocí nichž se celé úložiště ovládá. Před pokračováním ve čtení této sekce je proto doporučeno prostudovat si přílohu (9) především oddíly (9) a (9). Ukázka použití základních funkcí je uvedena v ukázce zdrojového kódu (Kód 1), kde jsou definovány dvě instance objektů. První je instance „pan *Josef*“ rozvolněného objektu typu *Osoba*, druhý objekt je typu *Váza*.

V úložišti je nutné nejdříve vytvořit definovatelné atributy, které potom mohou získávat jednotlivé instance objektů. To se provádí pomocí procedur v balíku *RS_ATTR* (9). Vytvořené atributy poté mohou být definovány na instancích objektů. Zatím neexistující atributy definovat na instance objektů nelze, neboť by bylo nutné při vytváření definovatelných atributů na instanci objektu i explicitně určovat, zda se má jednat o *obecný atribut* či *referenční atribut*. Z tohoto důvodu je nutné nejprve vytvořit definovatelné atributy a až poté definovat na instancích objektů. Základní procedury pro práci s atributy jsou procedury *CREATE_ATTR* (9) a *DROP_ATTR* (9). Na řádcích (04-09) je vytvoření obecných atributů *RC, JMENO, ZAMESTNANI*, *BARVA* a *VYSKA* a jednoho *referenčního atributu MAJITEL*.

Balík *RS_OBJ* slouží k práci s objekty. Pro vytvoření či zrušení instance objekt se používají procedury *CREATE_OBJ* (9) a *DROP_OBJ* (9). Přiřadit definovaný atribut s konkrétní hodnotou určité instanci objektu se provádí pomocí procedury *SET_VALUE* (9). Smazat atribut u instance určitého objektu se provede pomocí procedury *DEL_VALUE* (9). Samotné volání procedur *SET_VALUE* a *DEL_VALUE* je neefektivní, neboť při každé

provedené změně se přepočítají všechny hodnoty z katalogu pro danou instanci objektu a atribut. Pro efektivnější modifikaci objektu lze použít procedury *SET_VALUE_L* (9) a *DEL_VALUE_L* (9), které hodnoty v katalogu nepřepočítávají. Před použitím těchto procedur je však nejdříve nutné nejdříve uzamknout modifikovanou instanci objektu procedurou *LOCK_OBJ* (9) a po ukončení změn u instance objektu danou instanci opět odemknout zavoláním procedury *UNLOCK_OBJ* (9). Všechny hodnoty v katalogu i v indexech jsou poté automaticky přepočítány pouze jednou až v okamžiku odemknutí objektu. Nevýhodou efektivnějšího řešení modifikace určité instance objektu je možnost uzamknutí nanejvýš jednoho objektu v každém databázovém spojení. Navíc při uzamknutí objektu nebudou změny na něm provedené viditelné v ostatních spojeních do té doby než bude objekt odemknutý. Pokud je session ukončena bez odemknutí zamčené instance objektu, může daná instance zůstat v nekonzistentním stavu, protože nebudou aktualizovány hodnoty v katalogu. Je proto nutné po provedení změn uzamčené objekty důsledně odemykat. Ukázka kódu s manipulací s konkrétními instancemi objektů je na řádcích (11-32).

Vyhledávat id objektu podle jeho jednoznačného identifikátoru *IDENTIFICATOR* lze pomocí funkce *IDENT_TO_OBJ* (9), naopak identifikátor objektu lze podle id zjistit funkcí *OBJ_TO_IDENT* (9). Během manipulace s objekty je vhodné, aby uživatel přistupoval k objektu pomocí id, neboť přístup pomocí textového popisku v podobě identifikátoru vyžaduje pro každou zavolanou proceduru či funkci o jeden dotaz typu *select* více.

U procedur či funkcí, které mají jako vstupní parametry interval platnosti, lze i některý z krajních bodů časových intervalů ponechat jako *null*. Hodnota *null* parametru počátku intervalu času platnosti je nahrazena aktuálním časovým razítkem. Hodnota *null* parametru konce intervalu času platnosti je nahrazena časovým razítkem roku 1.1.2100 (plus nekonečno). Doplnění prázdných mezních hodnot intervalu času platnosti lze změnit v proceduře *UPDATE_VALID_INTERVAL* balíku *RS_OBJ* (9).

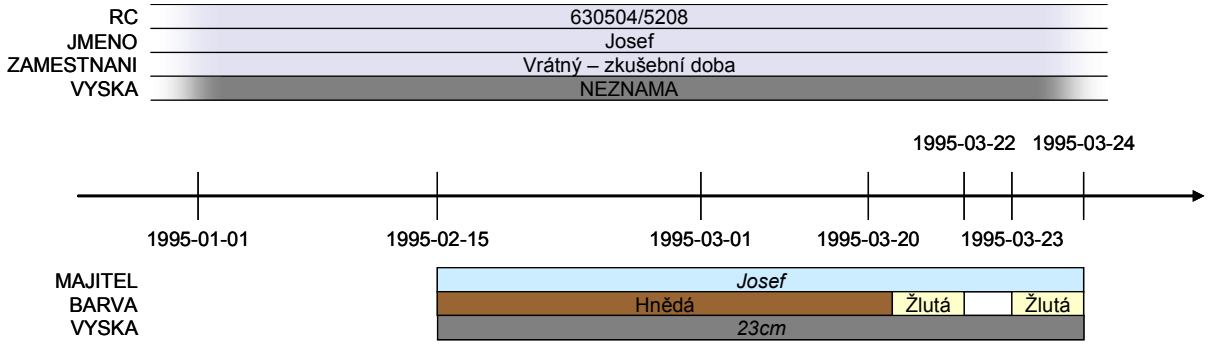
Ukázka zdrojového kódu (Kód 1) popisuje situaci zobrazenou na obrázku (Obrázek 8). Jedná se o krátký nešťastný příběh pana *Josefa* a jeho *vázy*. Definice rozvolněného objektu pana *Josefa* je uvedena na řádcích (11-16). Jak již čtenář ví z předchozích příkladů, tak pan *Josef* nastoupil 1.1.1995 do zaměstnání na pozici vrátného (18). První 3 měsíce má být ve zkušební době. Po první výplatě si pan *Josef* koupil hnědou *vázu* (20-25). Bohužel se 1.3.1995 nešikovnému panu *Josefovi* povede vázu polít žlutou barvou. Navíc kromě smolného začátku měsíce března se pan Josef dozví, že má chybnou nástupní smlouvu, a proto je nutné podepsat novou smlouvu se zaměstnavatelem, proto se mu i o 2 měsíce posune zkušební doba na pozici vrátného. Pan *Josef* se 22.3.1995 rozhodne udělat si radost tím, že se pokusí vyčistit vázu od

žluté barvy. Naneštěstí při použití ředidla se rozpustí i původní hnědý podklad a tak váza vyrobená ze skla nemá žádnou barvu (31). Čistě skleněná váza se panu *Josefu* nelíbí a tak ji hned druhý den nabarví opět na žlutou. Nakonec se ke vší té smůle váza 24.3.1995 rozbije (34).

```
01   DECLARE
02     OBJ_ID NUMBER;
03   BEGIN
04     RS_ATTR.CREATE_ATTR('RC', RS_INTER.ATTR_TEXT());
05     RS_ATTR.CREATE_ATTR('JMENO', RS_INTER.ATTR_TEXT());
06     RS_ATTR.CREATE_ATTR('ZAMESTNANI', RS_INTER.ATTR_TEXT());
07     RS_ATTR.CREATE_ATTR('MAJITEL', RS_INTER.ATTR_REF());
08     RS_ATTR.CREATE_ATTR('BARVA', RS_INTER.ATTR_TEXT());
09     RS_ATTR.CREATE_ATTR('VYSKA', RS_INTER.ATTR_TEXT());
10
11     OBJ_ID := RS_OBJ.CREATE_OBJ_RET_ID('NESTASTNY JOSEF');
12     RS_OBJ.LOCK_OBJ(OBJ_ID);
13     RS_OBJ.SET_VALUE_L('RC', '630504/5208', TO_TIMESTAMP('1963-05-04',
          'YYYY-MM-DD'), NULL);
14     RS_OBJ.SET_VALUE_L('JMENO', 'JOSEF', TO_TIMESTAMP('1963-05-04',
          'YYYY-MM-DD'), NULL);
15     RS_OBJ.SET_VALUE_L('VYSKA', 'NEZNAMA', TO_TIMESTAMP('1963-05-04',
          'YYYY-MM-DD'), NULL);
16     RS_OBJ.UNLOCK_OBJ();
17
18     RS_OBJ.SET_VALUE(OBJ_ID, 'ZAMESTNANI', 'VRATNY (ZKUSEBNI DOBA)',
          TO_TIMESTAMP('1995-01-01', 'YYYY-MM-DD'), TO_TIMESTAMP('1995-04-
          01', 'YYYY-MM-DD'));
19
20     OBJ_ID := RS_OBJ.CREATE_OBJ_RET_ID('VAZA');
21     RS_OBJ.LOCK_OBJ(OBJ_ID);
22     RS_OBJ.SET_VALUE_L('BARVA', 'HNEDA', TO_TIMESTAMP('1995-02-15',
          'YYYY-MM-DD'), NULL);
23     RS_OBJ.SET_VALUE_L('MAJITEL', RS_OBJ.IDENT_TO_OBJ('NESTASTNY
          JOSEF'), TO_TIMESTAMP('1995-02-15', 'YYYY-MM-DD'), NULL);
24     RS_OBJ.SET_VALUE_L('VYSKA', '23cm', TO_TIMESTAMP('1995-02-15',
         'YYYY-MM-DD'), NULL);
25     RS_OBJ.UNLOCK_OBJ();
26
27     RS_OBJ.SET_VALUE(OBJ_ID, 'BARVA', 'ZLUTA', TO_TIMESTAMP('1995-03-
          01', 'YYYY-MM-DD'), NULL);
28
29     RS_OBJ.SET_VALUE(RS_OBJ.IDENT_TO_OBJ('NESTASTNY JOSEF'),
          'ZAMESTNANI', 'VRATNY (ZKUSEBNI DOBA)', TO_TIMESTAMP('1995-01-
           01', 'YYYY-MM-DD'), TO_TIMESTAMP('1995-06-01', 'YYYY-MM-DD'));
30
31     RS_OBJ.SET_VALUE(OBJ_ID, 'BARVA', 'ZLUTA', TO_TIMESTAMP('1995-03-
          20', 'YYYY-MM-DD'), NULL);
32     RS_OBJ.DEL_VALUE(OBJ_ID, 'BARVA', TO_TIMESTAMP('1995-03-22',
'YYYY-
          MM-DD'), TO_TIMESTAMP('1995-03-23', 'YYYY-MM-DD'));
33
34     RS_OBJ.TRIM_OBJ(OBJ_ID, NULL, TO_TIMESTAMP('1995-03-24', 'YYYY-MM-
          DD'));
35   END;
```

Kód 1 - ovládání úložiště pomocí balíků

Na obrázku (Obrázek 8) jsou schematicky znázorněny hodnoty atributů. Nad časovou osou *TT* jsou atributy pana *Josefa* a pod ní *Vázy*. Časová osa je čistě orientační, protože její měřítko je zdeformováno. Na obrázku je i vidět, že společné atributy mohou mít i zcela různé druhy objektů. V tomto příkladě je společný atribut *VYSKA*, který je zakreslen u objektů nejníže (a je podbarven šedou barvou).



Obrázek 8 - znázornění nešťastného příběhu

Vybrané tabulky katalogu po vykonání kódu (Kód 1) vypadají tak, jak je uvedeno v tabulkách (Tabulka 16).

SYS_ATTRIBUTES				
ATTR_ID	NAME	TYPE	VALID_FROM	VALID_TO
1	RC	VARCHAR2	1963-05-04	2100-01-01
2	JMENO	VARCHAR2	1963-05-04	2100-01-01
3	ZAMESTNANI	VARCHAR2	1995-01-01	1995-06-01
4	MAJITEL	REFERENCE	1995-02-15	1995-03-24
5	BARVA	VARCHAR2	1995-02-15	1995-03-24
5	VYSKA	VARCHAR2	1963-05-04	2100-01-01

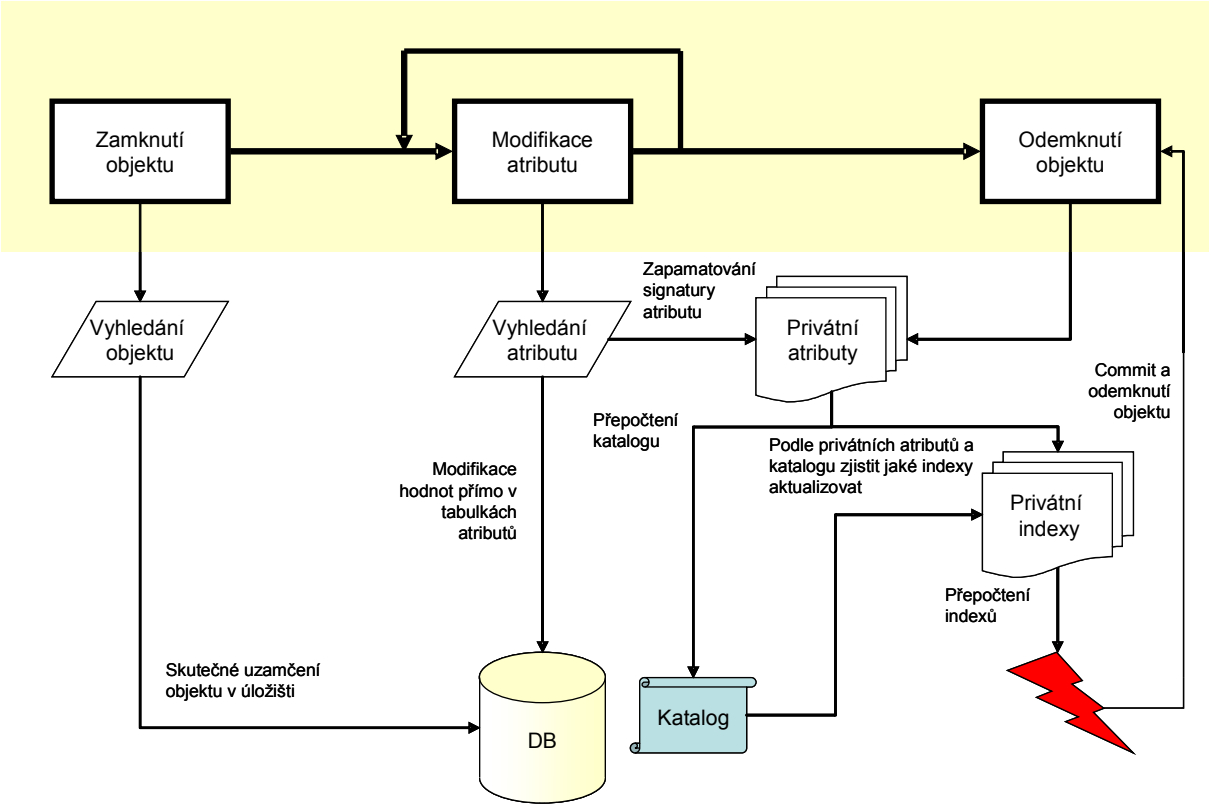
SYS_OBJECTS			
ATTR_ID	NAME	VALID_FROM	VALID_TO
1	1	1963-05-04	2100-01-01
2	2	1994-01-01	1995-03-24

SYS_OBJECT_ATTRIBUTE			
OBJECT_ID	ATTR_ID	VALID_FROM	VALID_TO
1	1	1963-05-04	2100-01-01
1	2	1963-05-04	2100-01-01
1	3	1995-01-01	1995-06-01
1	6	1963-05-04	2100-01-01
2	4	1995-02-15	1995-03-24
2	5	1995-02-15	1995-03-24
2	6	1995-02-15	1995-03-24

Tabulka 16 – vybrané tabulky katalogu u nešťastného příběhu

Na obrázku (Obrázek 9) je v horní části zakreslena logika ovládání. Logický postup uživatele je zvýrazněn tučnou čarou. Co se pod jednotlivými uživatelskými akcemi ve

skutečnosti v úložišti skrývá za procesy, je znázorněno ve spodní části obrázku. Logika odpovídá ovládání úložiště podle kódu (Kód 1).



Obrázek 9 - procesy v úložišti

5.6 Pojmenované třídy

V úložišti lze pro pohodlí programátora jednotlivé třídy pojmenovat, a tedy explicitně vytvářet pojmenované třídy. Všechny procedury balíku pro editaci pojmenovaných tříd jsou popsány v uživatelské příručce v sekci (9). Před pokračováním v této kapitole je doporučeno zmíněnou přílohu prostudovat.

Pojmenované třídy lze vytvářet pouze nad již vytvořenými atributy. Ukázka vytvoření pojmenované třídy je ve zdrojovém kódu (Kód 2). Ukázka vychází z nešťastného příběhu pana *Josefa* z obrázku (Obrázek 8). Novou pojmenovanou třídu lze vytvořit ze třídy *OBJECT* pomocí procedury *CREATE_CLSS* (9) či z již existující pojmenované třídy pomocí procedury *COPY* (9). K editaci existujících tříd slouží procedury *ADD_ATTR* (9) a *DEL_ATTR* (9). Každá pojmenovaná třída je definována výčtem již vytvořených definovatelných atributů. Pojmenované třídy tedy nelze definovat zápornou podmínkou. Tedy nelze stanovit, že určitý atribut nesmí mít instance objektů definované, aby náležely nějaké pojmenované třídě.

```
01   BEGIN
02     RS_CLSS.CREATE_CLSS('OSOBA');
03     RS_CLSS.ADD_ATTR('OSOBA', 'RC');
04     RS_CLSS.ADD_ATTR('OSOBA', 'JMENO');
05
06     RS_CLSS.COPY('OSOBA', 'ZAMESTNANEC');
07     RS_CLSS.ADD_ATTR('ZAMESTNANEC', 'ZAMESTNANI');
08
09     RS_CLSS.DROP_CLSS('OSOBA');
10   END;
```

Kód 2 - editace pojmenovaných tříd

V tabulkách (Tabulka 17) je několik vybraných tabulek katalogu s hodnotami, které vzniknou po provedení zdrojového kódu (Kód 2).

SYS_CLASSES			
CLASS_ID	NAME	ATTR_NAMES	ATTR_MASK
1	OBJECT		%
2	OSOBA	[JMENO][RC]	%[JMENO]%[RC]%
3	ZAMESTNANEC	[JMENO][RC][ZAMESTNANI]	%[JMENO]%[RC]%[ZAMESTNANI]%

SYS_CLASS_ATTRIBUTE	
CLASS_ID	ATTR_ID
2	1
2	2
3	1
3	2
3	3

Tabulka 17 – vybrané tabulky katalogu s příkladem pojmenovaných tříd

Ukázka použití pojmenovaných tříd k výběru odpovídajících instancí objektů je ve zdrojovém kódu (Kód 3). Vyhledání v tomto modelovém příkladě nalezne všechny instance, které odpovídají právě třídě definované množinou atributů. Funkce pro vyhledání instancí objektů jsou definovány se vstupním parametrem množiny atributů a nikoli se jménem pojmenované třídy, což umožňuje vyhledávat objekty podle tříd, které nejsou v úložišti explicitně pojmenované. Díky temporalitě mohou objekty v závislosti na časovém okamžiku třídu měnit, proto je vždy nutné specifikovat to, v jakém specifikovaném časovém okamžiku se hledání provádí. Funkce *NEXT_OBJECT_BY_ATTRS* (9) a *NEXT_OBJECT_BY_MASK* (9) vrací postupně vyhovující identifikátor odpovídající instance objektu, které má id nejmenší větší než je uvedeno ve vstupním parametru. Pokud není nelezen žádný objekt, je vráceno id s hodnotou 0. Tedy při selekci instancí objektů je podporována pouze dopředná navigace mezi objekty.

Konverze mezi jménem pojmenované třídy, množinou atributů či maskou lze pohodlně provádět pomocí funkcí *CLSS_TO_ATTRS* (9), *CLSS_TO_MASK* (9), *ATTRS_TO_CLSS* (9), *ATTRS_TO_MASK* (9), *MASK_TO_ATTRS* (9) a *MASK_TO_CLSS* (9). Hodnoty vracené těmito funkcemi jsou výsledky vhodných dotazů pouze z tabulky *SYS_CLASSES*.

```
01   DECLARE
02     OBJ_ID NUMBER;
03     ATTRS VARCHAR2(256);
04     VALTIME TIMESTAMP;
05   BEGIN
06     VALTIME := TO_TIMESTAMP('1995-02-02', 'YYYY-MM-DD');
07     ATTRS := RS_CLSS.CLSS_TO_ATTRS('ZAMESTNANEC');
08     OBJ_ID := RS_OBJ.NEXT_OBJECT_BY_ATTRS(ATTRS, VALTIME, 0);
09     WHILE (OBJ_ID > 0)
10     LOOP
11       DBMS_OUTPUT.PUT_LINE(OBJ_ID);
12       OBJ_ID := RS_OBJ.NEXT_OBJECT_BY_ATTRS(ATTRS, VALTIME, OBJ_ID);
13     END LOOP;
14   END;
```

Kód 3 - příklad použití pojmenovaných tříd

5.6.1 Navigace mezi objekty

V úložišti je při selekci instancí objektů v případě předpokladu výběru malé množiny dat vrácen celý výsledek jako tabulka s požadovanými daty. V rámci takové tabulky se může uživatel pohybovat po vrácených instancích libovolně.

V případě většího výsledku jsou instance vyhledávané postupně. Vždy je vrácena instance z vyhovujících instancí s nejmenším větším id. Tedy vyhledávání je dopředné a v rámci jednoho požadavku se jedná o sekvenční dopředné čtení. Konkrétní příklad použití dopředné navigace je v kódu (Kód 3).

5.7 Optimalizace v Oracle

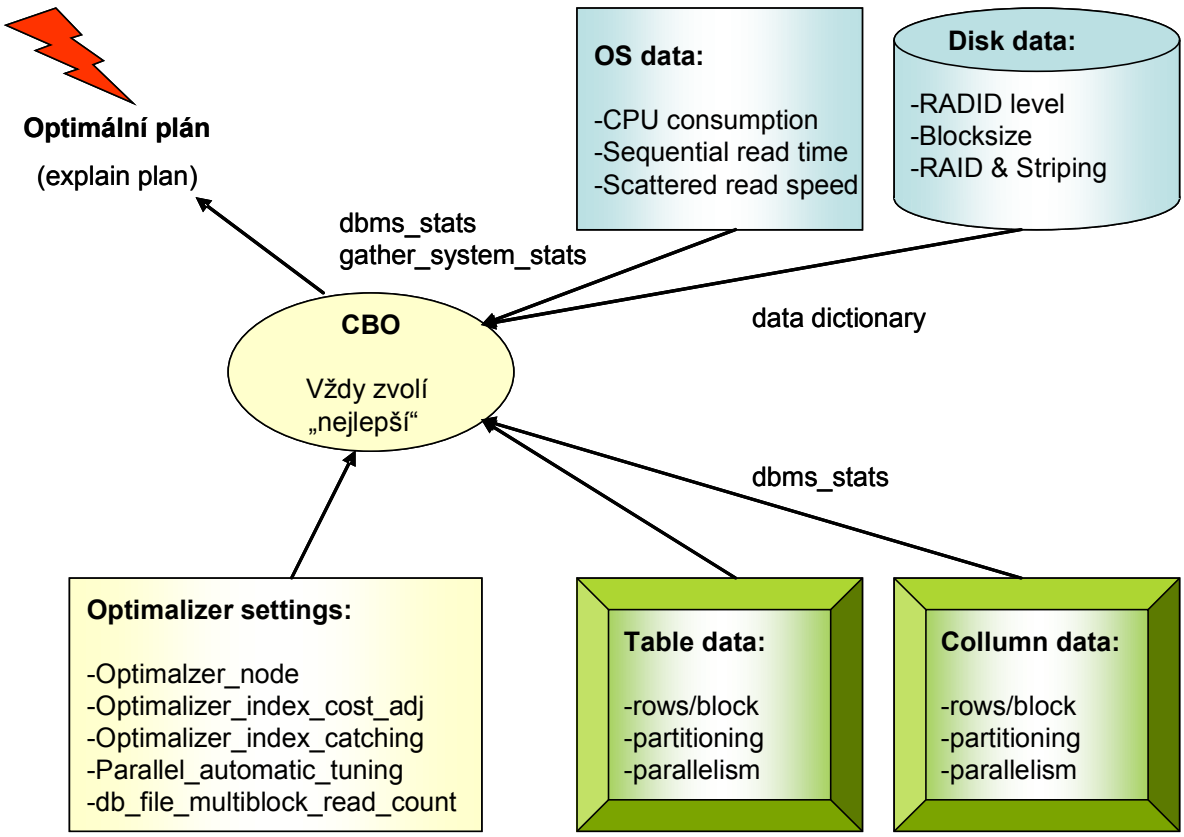
Tato sekce je věnována nastavení a optimalizaci dotazů v prostředí Oracle. Především poté nastavení používání indexů. Jsou diskutovány pouze vybrané postupy použitelné v implementovaném úložišti této práce. Informace, zde popsané podstatnou mírou ovlivnily výslednou implementaci.

5.7.1 Optimalizace dotazů

Cílem databázových systémů je si každý dotaz vnitřně naplánovat a vykonat ve tvaru vyžadujícím minimální možný čas provedení. Bohužel, automatická optimalizace každého dotazu není všemocná a stále ještě existuje několik doporučených rad, pomocí nichž lze docílit časově ještě méně náročné vykonání dotazu.

Databázový systém Oracle ukládá o každé tabulce, sloupci i indexu statistiky, podle kterých se snaží vždy nalézt optimální plán vykonání. Od verze 8 je v systému nový optimalizátor, který využívá statistiky. Pro plné využití všech statistik se doporučuje klást Oraclu dotazy s co nejvíce konkrétními hodnotami. V případě použití parametrů v dotaze je nejdříve dotaz optimalizován s nedefinovanými parametry a až při vykonávání dotazu se hodnoty parametrů doplní. Je tedy možné, že dotaz bez parametrů bude vykonán rychleji než obecný dotaz. Z jakých vstupů se snaží Oracle vytvořit optimální plán pro každý dotaz, je znázorněno na obrázku (Obrázek 10) 9.

Dále je doporučeno, aby všechny typy sloupců a parametrů, s nimiž jsou porovnávány, byly shodné a nemuselo tak docházet k implicitním konverzím. V patologických případech může být totiž implicitní konverze prováděná při zpracování každé řádky dotazu. Implicitní konverze a porovnávání jiných typů může vést k nepoužití indexů během dotazu. Z těchto důvodů je doporučeno nejdříve všechny proměnné, použité v dotazu explicitně přetypovat v nových proměnných a dotaz položit poté s hodnotami z nových proměnných.



Obrázek 10- optimální plán

5.7.2 Nucené používání a zakazování indexů

Po zkonstruování syntakticky optimálního dotazu je v Oracle možné další rychlení vykonání dotazu docílit s použitím indexů. Pokud se uživateli zdá, že plánovač zbytečně používá či nepoužívá nějaký index, je možné ručně určit, jaký index se má při vykonání dotazu použít [19].

Nápověda pro vykonání dotazu se v Oracle píše podle vzoru ve zdrojovém kódu (Kód 4) do bloku s komentářem, začínajícím znakem „+“ bezprostředně za prvním klíčovým slovem. Nápovědu lze použít pro dotazy *SELECT, UPDATE* a *DELETE*.

```
SELECT /*+ comment */ ........ ;
```

Kód 4 - hinty v dotazech

Vynucení použití indexu se provádí pomocí nápovědy ve formátu */*+ *INDEX (table [index [index]...]) */*. Například tedy můžeme vynutit použití index *idx_job_code* ve zdrojovém kódu (Kód 5).

```
select /*+ INDEX(emp_city idx_job_code) */ empname,
job_code from emp where job_code = 'T';
```

Kód 5 - příklad vynucení použití indexu

Kromě klíčového slova *INDEX* v prefixu komentáře lze i určit postup procházení dat klíčovými slovy *INDEX_ASC, INDEX_DESC* a *FULL*.

Naopak, pokud není použití indexu navrženého optimalizátorem vhodné, lze vynutit nepoužití indexu klíčovým slovem *NO_INDEX*. Příklad nuceného vynechání indexu je ve zdrojovém kódu (Kód 6).

```
select /*+ NO_INDEX(emp_status emp_status) */ empname,
status from emp_status where status = 'P';
```

Kód 6 - příklad zakázání použití indexu

Patologická situace použití indexů nastává v případě, že jsou použité tabulky v dotazech malé [20]. V těchto případech zpravidla Oracle nepoužije žádný index, protože nic nebrání načtení tabulek do paměti a jejich přímému průchodu. Tento způsob je v tomto případě nejrychlejší, neboť malé tabulky jsou zpravidla uloženy v několika blocích za sebou.

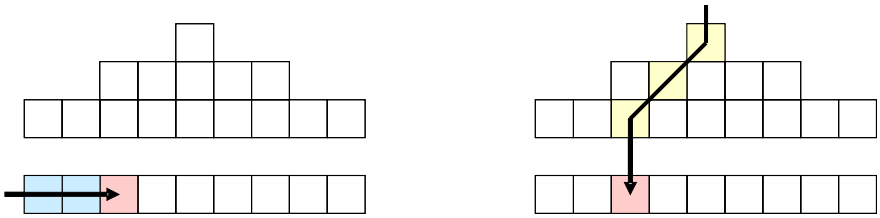
5.7.3 Nastavení parametru doménového indexu

Pro každý index si ve verzi Oracle 10g pamatuje několik parametrů, podle kterých se při vytváření optimálního plánu rozhoduje, jak vhodné je daný index pro daný dotaz použít 9. Nejvlivnější parametry pro optimalizaci dotazu jsou:

- *optimizer_index_caching* parametr nastavuje sama DBA. Určuje pro daný index, jak moc jsou data indexu načtená v keš paměti (RAM). Rozsah hodnot je od 0 do 100 9. Tento parametr má největší vliv na dotazy se spojením dvou a více tabulek.
- *optimizer_index_cost_adj* je nejdůležitější parametr pro každý index, který má největší vliv na rozhodování zda daný index použít. Rozsah hodnot tohoto parametru je 1 až 10000, defaultní hodnota Oraclu je nastavena 100. Větší hodnota parametru říká, že se bude tabulka pravděpodobněji procházet postupně full-scanem, menší hodnota naopak upřednostňuje přístup k záznamům skrze index.

Na obrázku níže (Obrázek 11) níže je graficky znázorněn rozdíl mezi přístupen k datům full-scanem a skrze index. Index je na obrázku tvořen stromem nad spodní řadou dat

tabulky. V levé části obrázku se full-scanem procházejí data tabulky postupně, nakonec se přečtou všechna primární data pro zpracování daného dotazu. Naopak v pravé části obrázku se nejprve podle klíče v indexu naleznou data a do tabulky s daty se poté přistoupí pouze pro požadovaný záznam.



Obrázek 11 – vyhledání full scanem (bez indexu) vs s indexem

5.8 Indexy

Hlavním cílem indexů v databázovém úložišti je co nejvíce zefektivnit vyhledání v datech na úkor času nutného pro modifikaci a větší paměťové režie indexovaných dat. V mé implementaci lze díky katalogu rychle zjišťovat informace o libovolném objektu v případě, že uživatel zná jednoznačný identifikátor objektu *OBJECT_ID*. Tedy zjištění atributů či jejich hodnot je pro libovolnou instanci objektu v úložišti dostatečně efektivní. Naopak v případě hledání určitého objektu *OBJECT_ID* či množiny objektů podle hodnot určitých atributů je situace horší. Toto hledání přímo nad tabulkami by bylo velmi neefektivní, a proto jsem se při implementaci indexů soustředil právě na tento problém.

5.8.1 Struktura indexu

V úložišti se vždy indexují atributy, podle kterých se mají vyhledávat jednotlivé instance objektů. Indexovaným klíčem je tedy (k+2)-tice, kde k je počet indexovaných atributů plus dva sloupce pro temporální složku *VALID_FROM* a *VALID_TO*. V důsledku indexování k-tic a použité temporality mohou podle sekce (4.7.2) vznikat redundantní hodnoty v částech indexovaných k-tic. Celá indexovaná (k+2)-tice tedy klíč je však v rámci jednoho objektu unikátní. Indexovaní množiny atributů, může způsobit, že klíč nepůjde v určitý časový okamžik vytvořit, protože některý z indexovaných atributů nebude definovaný v daný časový okamžik.

Pro dvojici instanci objektu a časový okamžik musí být definováno všech k indexovaných atributů, aby bylo možné vytvořit klíč, který bude následně použit v indexu,

jinak záznam v indexu nebude vůbec. Interval času platnosti *k*-tice je ohraničen *VALID_FROM* a *VALID_TO*. V každý časový okamžik $t \in [VALID\_FROM, VALID\_TO)$ *k*-tice klíče existuje v úložišti alespoň jedna instance objektu $o \in O$ taková, která má definovány všechny atributy. Tedy $val(o, a_i)^t != null$ pro všechny $a_i \in$ (*k*-tice). Pro jasnější rozlišení budou nyní *VALID_FROM* a *VALID_TO* pro *k*-tici označovány jako $VALID\_FROM_{k-tice}$ a $VALID\_TO_{k-tice}$ a pro atribut a_i jako $VALID\_FROM_{a_i}$ a $VALID\_TO_{a_i}$. Navíc $VALID\_FROM_{k-tice} \leq VALID\_FROM_{a_i}$ a $VALID\_TO_{k-tice} \geq VALID\_TO_{a_i}$.

Indexované hodnoty jsou id jednotlivých instancí objektů, pro které šlo vytvořit celý klíč tedy (*k*+2)-tici.

5.8.2 Uzamykání objektů

Jak již bylo zmíněno v sekci (5.5.5) každá instance objektu, jejíž atribut se bude modifikovat, tedy mazat či se bude měnit hodnota daného atributu, je nutné uzamknout teprve poté, co lze hodnotu daného atributu modifikovat a nakonec je nutné instanci odemknout. I všechny ovládací funkce a procedury, které nevyžadují uzamčený objekt na vstupu interně objekt uzamknou, poté zmodifikují a nakonec odemknou. Z důvodu tohoto omezení lze některé operace provádět najednou a optimálněji, konkrétně se jedná o přepočetní hodnot v katalogu pro daný objekt a upravení indexu podle nových hodnot atributu.

Během modifikace uzamčené instance objektu si úložiště poznamenává do privátních proměnných *LOCK_ATTR_IDS*, *LOCK_ATTR_VALID_FROM* a *LOCK_ATTR_VALID_TO* modifikované atributy a případně další dodatečné informace. Proměnné postupně obsahují id a signatury (hodnoty od kdy a do kdy byl daný atribut modifikován) modifikovaných atributů. Tyto proměnné jsou při uzamčení instance objektu vynulovány.

Při odemknutí uzamčené instance objektu dojde v první řadě ke zviditelnění nových změněných hodnot modifikovaných atributů a k přepočtení statistik v katalogu o úložiště, kde jsou upraveny patřičné hodnoty modifikovaných atributů pro uzamčenou instanci. V další fázi odemykání objektu dojde k vyhledání indexů, které bude nutné upravit. Nutně modifikované indexy jsou uloženy v privátních proměnných *MOD_INX_IDS*, *MOD_INX_VALID_FROM* a *MOD_INX_VALID_TO*. Proměnné postupně znamenají id upravovaného indexu a časovou složku od kdy do kdy bude daný index nutné aktualizovat. I při modifikaci jednoho atributu je nutné přepočítat celý klíč indexovaného záznamu, a tedy stačí si pamatovat pouze jaký index

a v jaký časový interval má být přepočten. Samotné přepočtení indexu pro uzamčenou instanci objektu se provádí tak, že se nejprve smažou všechny klíče v indexu pro uzamčenou instanci objektu ve vypočtený časový okamžik. Až po vyčištění všech záznamů v daném časovém intervalu se znovu projdou hodnoty v indexovaných atributech podle kterých se vytvoří nové klíče pro uzamčený objekt v daný časový interval a tyto nové klíče se nakonec vkládají do indexu. Při vkládání nových klíčů se kontrolují krajní klíče s těmi, co jsou již v indexu a případně se shodné klíče slévají. Tedy pouze upraví případný čas platnosti klíče tak, aby nevznikaly redundantní klíče za sebou podle složky časové.

Je vidět, že indexy tedy používají pouze operace *delete* a *insert*, které je nutné řešit a operace *update* je tedy vynechána.

Důvodem vynucení přepočítávání statistik a indexů na pouze jednom místě je možnost tyto operace provádět souběžně, a tak i mnohem efektivněji. Neboť jak pro přepočtení statistik, tak i pro přepočtení indexu jsou nutné přístupy do primárních tabulek modifikovaných atributů.

Úložiště je optimalizované na práci s právě jedním objektem. Je proto nejvýhodnější vždy každý zmodifikovat najednou a teprve poté začít pracovat s dalším objektem. Naopak není optimální postupovat při modifikaci podle temporální složky a v nějaký časový okamžik změnit všechny hodnoty atributů více instancí a poté se přesunout v čase o kousek dále a operaci opakovat. Přepočtení indexů a statistiky má také za důsledek to, že až do odemknutí instance se bude v dotazech typu *select* vracet/nevracet třeba i daná uzamčená instance, neboť hodnoty v indexech nebudou aktuální.

Pokud při modifikaci uzamčeného objektu přetečou privátní proměnné, jsou statistiky interně přepočteny a vyprázdněny. To má za následek to, že v dané session již mohou být částečně upravené hodnoty například v katalogu či indexu. Úplné odstínění této vlastnosti není možné, lze pouze zvětšením rozsahů privátních polí v balíku *RS_OBJ* tento efekt oddálit. Nicméně aktuální nastavené rozsahů polí se při plnění úložiště náhodnými daty nepodařilo překročit.

5.8.3 Vytvoření indexu

V úložišti jsou implementovány dva druhy indexů. Každý z indexů byl programován v jiném jazyce pro porovnání výkonnosti PL/SQL a externě volaného kódu psaného v C++. První z nich využívá standardní indexy poskytované databází Oracle (5.8.4), druhý je založen na vlastní implementaci R-stromu (5.8.6). Největším problémem pro implementované indexy je nutnost indexovat hodnoty z různých tabulek. Řešení toho problému vedlo k různé syntaxi

při selekci s indexy. Manipulace s indexy se provádí opět skrze procedury v balíku popsaném v příloze (9) a pracuje se s nimi tedy totožně.

Ukázka manipulace s indexem je ve zdrojovém kódu (Kód 7). V implementovaném úložišti se indexy pro oba dva typy indexů vytvářejí dvojfázově. V první fázi se vytvoří *abstraktní index* (02) pomocí procedury *CREATE_INX_ABSTRACT* (9). *Abstraktní index* zatím není nikde reálně vytvořený. O *abstraktním indexu* je pouze záznam v katalogu. Z tohoto důvodu *abstraktní index* nelze použít k vyhledávání. Při vytváření *abstraktního indexu* nejsou kontrolována žádná omezení a lze tedy nadefinovat i nekorektní *abstraktní index*. Po vytvoření *abstraktního indexu* lze k danému indexu přidávat či odebírat jednotlivé již vytvořené definovatelné atributy (03-04). To se provede procedurami *ADD_ATTR* (9) a *DEL_ATTR* (9). K *abstraktnímu indexu* lze také přidat či odebrat množinu atributů určité pojmenované třídy pomocí procedur *ADD_CLSS* (9), *ADD_CLSS_OVER* (9) a *DEL_CLSS* (9). Pro vytvoření *skutečného indexu* je nutné mít předpřipravený vhodně definovaný *abstraktní index*, ze kterého se vytvoří *skutečný index* použitelný pro optimalizaci vyhledávání. To se provede pomocí procedury *CREATE_INX_LEGAL* (9) na řádku (05). Až při vytváření *skutečného indexu* se kontroluje, zda je *abstraktní index* korektní. Omezení korektního indexu jsou diskutována v sekci (5.9.5). Všechna omezení pro korektní index jsou ohlášená výjimkami při pokusu o vytvoření *skutečného indexu* (9). Pokud nelze o indexu říci, že je nekorektní je považován za korektní. Tedy hlavní rozdíly mezi *abstraktním* a *skutečným indexem* jsou ty, že *abstraktní index* lze modifikovat, může být i nekorektní a nelze jej využít při vyhledávání. Oproti tomu *skutečný index* musí být korektní, jinak není vytvořen, má definované indexové struktury a tudíž jej lze použít při vyhledávání. Jakýkoliv index ať abstraktní či skutečný lze smazat procedurou *DROP_INX* (9), která smaže i všechny struktury, pokud je daný *index skutečný*. Konstanty pro druh indexu jsou *INX_TYPE_ORA* (9) pro Oracle index a *INX_TYPE_RO* (9) pro externí index. Konstanty pro indikaci, zda se mají doindexovat data před vytvořením indexu či nikoliv jsou *INX_CREATE_REINDEX_ALL*(9) a *INX_CREATE_NO_REINDEX_ALL*(9).

```
01   BEGIN
02     RS_INX.CREATE_INX_ABSTRACT('SEZNAM_VAZ_IX');
03     RS_INX.ADD_ATTR('MUJ_INDEX', 'MAJITEL');
04     RS_INX.ADD_ATTR('MUJ_INDEX', 'BARVA');
05     RS_INX.CREATE_INX_LEGAL('SEZNAM_VAZ_IX', RS_INTER.INX_TYPE_ORA(),
            RS_INTER. INX_CREATE_REINDEX_ALL());
06
07     RS_INX.DROP_INX('SEZNAM_VAZ_IX');
08   END;
```

Kód 7 - vytvoření indexu

Ve vybraných tabulkách katalogu (Tabulka 18) jsou hodnoty po vykonání zdrojového kódu (Kód 7), na řádcích (01-05).

SYS_INDEXES			
INDEX_ID	NAME	LEGAL	RS_INDEX_TYPE
1	SEZNAM_VAZ_IX	1	0

SYS_INDEX_ATTRIBUTE	
INDEX_ID	ATTR_ID
1	4
1	5

Tabulka 18 - katalog s příkladem vytvoření indexu

Použití indexů při vyhledání se však liší a bude proto ukázáno v kontextu zvoleného druhu indexu. Tedy podrobná diskuze i s názornou ukázkou bude v podsekcích (5.8.4) a (5.8.6).

V indexech nejsou povolené *null* hodnoty, proto jsou v indexech vždy evidované pouze časové intervaly, kdy jsou definovány všechny indexované atributy. Navíc, skutečné indexy lze vytvářet pouze nad zatím prázdnými atributy, protože při vytváření indexu se vytvoří prázdná struktura a nedoplňují se všechny informace do indexu. Pokud je tedy skutečný index vytvořen nad množinou již před vyplněných atributů, bude index prázdný. V indexu budou zaneseny pouze změny v hodnotách indexovaných atributů, které budou provedeny až po vytvoření skutečného indexu.

5.8.4 Interní index Oracle

Tento druh indexu je založen na vestavěném indexu Oracle. Hlavní nevýhodou, kterou bylo nutné řešit při použití tohoto indexu je možnost indexovat pouze sloupce z jedné tabulky. Hlavní výhodou tohoto indexu je poté právě to, že se jedná o vestavěný index Oracle, který během své činnosti dokáže používat statistiky vedené Oraclem o datech v každé tabulce. Využití tohoto indexu je během vyhledávání velmi efektivní, neboť se nevolá žádný externí kód a není nutné ani nic překládat. Navíc použití statistik při hledání mezních případů zcela index odstíní a výsledek tak lze získat až v konstantním čase. Podrobnější diskuze o efektivitě a porovnání indexů je v kapitole (6).

Bohužel, možnost indexování sloupců pouze z jedné tabulky vedla k nutnosti vytvoření nové pomocné indexové tabulky podle jména indexu, která obsahuje hledaný sloupce (*OBJECT_ID*), sloupce hodnot indexovaných atributů a dva sloupce platnosti

(*VALID_FROM*, *VALID_TO*). Sloučení indexovaných hodnot do jedné tabulky a omezení časů platnosti na disjunktní intervaly může vést až k téměř dvojnásobnému počtu záznamů v indexové tabulce (více v sekci 4.7.2). Navíc reálný index je vytvořen až nad pomocnou indexovou tabulkou, kdy se indexují redundantní data. Je vidět, že při modifikaci dat se musí navíc změnit data v indexové tabulce, aby bylo možné přepočítat index. Paměťová režie je proto minimálně 2krát, a kvůli temporalitě až 5krát, větší. Nevýhodou je i nemožnost realizovat intervalové dotazy podle libovolného atributu. Na druhou stranu by se mělo jednat při vyhledávání o nejefektivnější index, což se uvidí v testech.

5.8.5 Vyhledání podle interního indexu Oracle

Vyhledání podle indexu Oracle se použije v případě zadání dotazu *select* na pomocnou tabulku tohoto indexu. Jméno tabulky je totožné se jménem indexu. V tabulce indexu lze vybírat id vyhovujících instancí objektů, ale navíc také hodnoty indexovaných atributů. Tím, že je index vytvořen nad tabulkou, mohou být některé hodnoty atributů instancí objektů v tabulce daného atributu uloženy redundantně, protože se mohou vyskytovat jevy popsané v sekci (4.7.2). Pro vytvoření indexu Oracle je nutné volat proceduru *CREATE_INX_LEGAL* (9) s parametrem *RS_INX_TYPE* rovným konstantě *INX_TYPE_ORA* (9).

Nechť je v úložišti vytvořen vlastní index *SEZNAM_VAZ_IX* podle zdrojového kódu (Kód 7) a v úložišti jsou data po vykonání zdrojového (Kód 1), potom lze klást dotazy s použitím vlastního indexu podle ukázky v kódu (Kód 8). Díky nemožnosti vynechání časových intervalů, kdy nebyly všechny indexované atributy definovány na některé instanci, bude výsledek dotazu vypadat stejně jako na tabulce (Tabulka 19). Případné vynucení použití interního Oracle indexu lze provést způsobem popsaným v sekci (5.7.2), neboť systém Oracle podle statistik rozhodne o použití definované indexu nad tabulkou jinak sám (5.7.1).

```
SELECT /*+SEZNAM_VAZ_INDEX */ OBJECT_ID
FROM SEZNAM_VAZ_IX
WHERE
     MAJITEL=1 AND BARVA='ZLUTA ' AND
     VALID_FROM <= TO_TIMESTAMP('1995-01-01', 'YYYY-MM-DD') AND
     VALID_TO >= TO_TIMESTAMP('1996-01-01', 'YYYY-MM-DD');
```

Kód 8 - select nad indexem Oracle

MUJ_INDEX				
OBJECT_ID	MAJITEL	BARVA	VALID_FROM	VALID_TO
2	1	ZLUTA	15.3.1995	22.3.1995

Tabulka 19 - výsledek použití indexu Oracle

5.8.6 Struktura externího indexu

Cílem každého úložiště je především efektivní vyhledávání mezi uloženými daty. K tomu se obecně používají indexy. Indexování jednoho atributu není v úložišti, založeném na relační databázi problém, neboť se jedná o vytvoření indexu nad jednou tabulkou. V případě současného indexování více atributů je však situace opačná. Žádný ze současných databázových systémů totiž neumožňuje vytvořit složený či dokonce prostorový index nad více tabulkami. V této sekci je proto představen návrh pro vhodný index i s formálním popisem, který je nezávislý na potenciální implementaci v určitém databázovém systému.

Pro současné indexovaní *k* atributů, je díky temporální složce (diskutováno v sekci 5.8.1) nutné vytvoření indexu nad (*k*+2)-cemi. Přesněji se jedná o vytvoření více dimenzionálního indexu nad různými tabulkami, ve kterém budou indexovány úsečky rovnoběžné s temporální osou v (*k*+1) dimenzionálním prostoru.

Jako vhodné struktury pro indexování více rozměrných objektů 9 v této práci byly uvažovány R, R* a Buddy-stromy.

Buddy-stromy 9 jsou velice efektivní při indexování bodů v prostoru, navíc je jejich případná implementace poměrně snadná. Výhodou Buddy-stromů je rovněž podrobný popis operací *insert* a *delete*, které jsou při indexu použity. Bohužel, při pokusu indexovat jiné objekty než body se nechovají optimálně, a proto nemohly být použity..

R-stromy 9 jsou o něco složitější na implementaci než Buddy-stromy, ale umožňují uspokojivě indexovat větší prostorové objekty než body. Největším problémem této struktury je však v literatuře málo popsaná operace *delete*. Přes nutnost doimplementovat korektní operaci *delete* byla nakonec použita právě tato struktura. Algoritmy pro operace s použitými R-stromy jsou diskutovány v sekci (5.8.7).

R*-stromy 9 jsou variantou R-stromů, které umožňují velmi efektivně indexovat vícedimenzionální objekty. Při vyhledávání jsou díky optimálnějšímu uložení efektivnější než R-stromy, ovšem za cenu dražšího insertu, než je tomu u R-stromů. To je způsobeno tím, že při vkládání prvku se provádějí složitější heuristiky podle více kritérií či v případě, že se celý strom *„posunuje nějakým směrem pryč“*, tak prvky na okraji stromu jsou smazány a opět vloženy. V důsledku toho je výsledný R*-strom optimálněji vyvážen. U R*-stromů se však předpokládá menší počet operací *delete* než *insert*, což je předpoklad, který nelze zaručit a proto R*-stromy nakonec nebyly při implementaci použity.

Ačkoli jsou v prezentovaném návrhu použity R-stromy, lze použít i jakýkoliv jiný výše uvedený strom. Buddy-stromy lze použít s tím, že indexovat se budou pouze (*k*+1)-tice a případné porovnání hodnoty *VALID_TO* bude provedeno až při případném dohledání určitého

prvku.

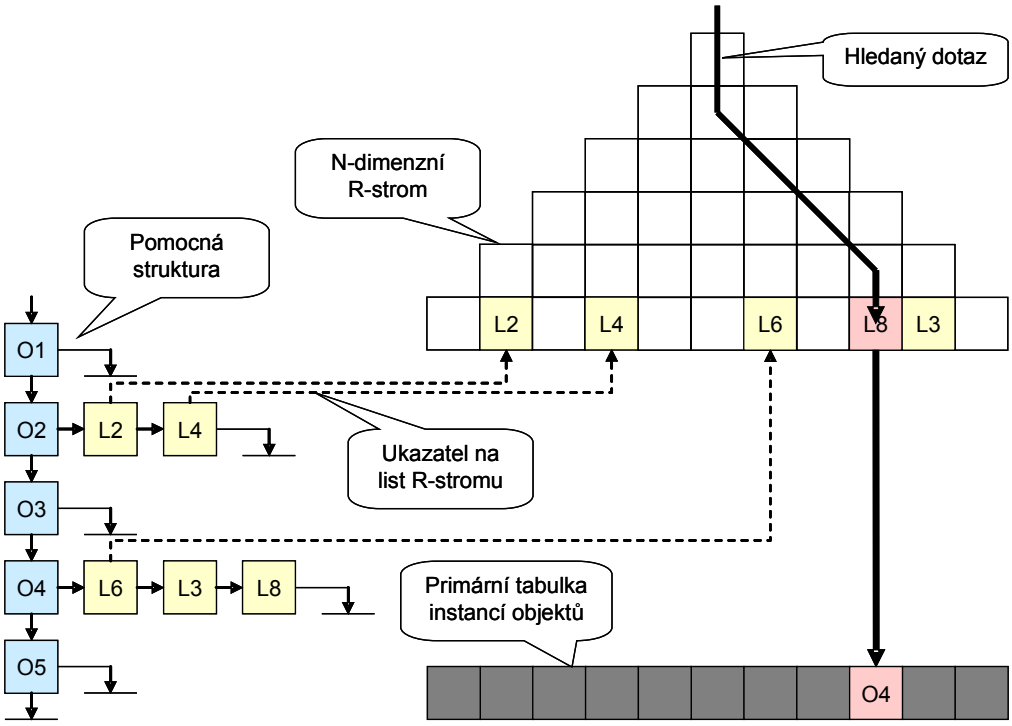
Schéma navrhované struktury založené na R-stromech je znázorněno na obrázku (Obrázek 12). Při návrhu struktury byly brány v potaz problémy, diskutované v sekci (5.5.5).

Na obrázku (Obrázek 12) je v pravé horní části zachycen vícerozměrný strom indexu, podle kterého se bude vyhledávat v případě dotazu typu *select*. V R-stromu indexu jsou indexované hodnoty atributů, tedy vlastnosti instancí objektů, podle kterých se vyhledává. Například to může být, že pan *Josef* má nastavenou vlastnost *VYSKU* na hodnotu *NEZNAMA*. Listy R-stromu ukazují do tabulky instancí objektů, tedy do tabulky *SYS_OBJECTS*, kde jsou uvedeny všechny instance objektů i se svým unikátním identifikátorem. Při dotazech typu *select* tedy navržená indexová struktura slouží obdobně jako klasický vestavěný index Oracle. Průchod ukázkového dotazu typu *select* je znázorněn tučnými šipkami.

Při operaci typu *insert* se ze vstupní (k+2)-tice vytvoří více rozměrný klíč, který se pomocí standardního *insertu* R-stromu (Algoritmus 2) vloží do stromové struktury. Hodnotou pod klíčem bude odkaz do tabulky všech instancí objektů v úložišti. Během provádění operace *insert* se neprovádí žádná činnost navíc.

Při operaci typu *delete* je však situace odlišná. Obecně R-stromy nejsou příliš vhodné na *delete* operace a navíc v navrženém modelu se zamykáním by se nejdříve musely vyhledat hodnoty, které se budou mazat, z těch následně vytvořit klíče a ty smazat. Bohužel v případě, kdy se v době uzamčení objektu mohou hodnoty určitých atributů změnit více než jedenkrát by bylo dohledávání mazaných hodnot atributů problematické a neefektivní. V modelové struktuře pro externí index je proto uložena pomocná struktura v podobě hashmapy s klíči jednoznačných identifikátorů instancí objektů. Každý prvek v této hashmapě poté obsahuje spojový seznam pro daný objekt, setříděný podle *VALID_FROM* s ukazateli na listy R-stromu, kde jsou požadovaná data. To umožňuje si během modifikace uzamčené instance objektu poznamenávat pouze id uzamčené instance a pro každý atribut pouze intervaly platnosti modifikovaných atributů. Pomocí hashmapy se v amortizované složitosti nalezne vhodný seznam pro daný objekt v podle id. Položky v seznamu jsou velmi malé, neboť obsahují pouze časové razítko s hodnotou *VALID_FROM*, ukazatel na list R-stromu a ukazatel na následující prvek daného seznamu. Díky tomu, že jsou prvky malé, lze v případě mazání celou strukturu uložit do paměti a tím využít rychlosti procesorů namísto zdlouhavého načítání z disku. Dále je ušetřeno velké množství času při potenciální rekonstrukci celého klíče pro mazaný záznam, neboť není nutné přistupovat do všech tabulek indexovaných atributů. Navíc při mazání lze v amortizovaně konstantním čase přistoupit k mazanému listu v R-stromu místo prohledávání R-stromu od kořene.

V reálné implementaci může být i hashmapa reprezentována pomocí spojového seznamu, neboť každý prvek obsahuje pouze jednoznačný identifikátor instance a ukazatel na spojový seznam.



Obrázek 12 - navržená struktura pro externí index

5.8.7 Algoritmy R-stromu

Značení u popisu algoritmů pro jednotlivé operace s R-stromy je převzato z textu Pokorného a Žemličky 9:

Indexový (řídící) záznam	E
Část s kvádrem	E.I
Část s ukazatelem nebo identifikátorem	E.p
Objem kvádru	o(I)

Algoritmus pro vyhledání v oblasti *S* ve stromu zakořeněným v *T* je popsán v pseudokódu (Algoritmus 1).

Algoritmus: Search_R(T, S)
Vstup: R-strom s kořenem v *T*, kvádr *S*
Výstup: identifikátory objektů překrývajících *S*

```
if (T ≠ list) then
     foreach E ∈ T do
          if (E.I překrývá S) then
```

```
                SEARCH_R(E.p, S);
else
     foreach E ∈ T do
          if (E.I překrývá S) then
                PRINT(E.p);
```

Algoritmus 1 - vyhledání v R-stromu

Algoritmus pro vkládání indexového záznamu E do stromu zakořeněném v T je popsán v pseudokódu (Algoritmus 2). Při rozdělování přeplněných uzlů je použito štěpení podle Guttmana 9.

Algoritmus: Insert_R(T, E)
Vstup: R-strom s kořenem v *T*, indexový záznam *E*
Výstup: identifikátory objektů překrývajících *S*

```
ChooseLeaf(T, L, E);                                    {Vyber list L z
T}
LL := NIL;
if (L není plný) then
     přidej E do L;
else
     SplitNode(L, LL, E);
AdjustTree(L, LL, T);                                {změň pokrytí z L a
LL}
if (došlo ke štěpení T) then
     Vytvoř nový kořen T;
```

Algoritmus 2 - vložení prvku do R-stromu

Algoritmus: ChooseLeaf(T, L, E)
Vstup: R-strom s kořenem v *T*, indexový záznam *E*
Výstup: list *L*, kam patří *E*

```
N := T;
while (N ≠ list) do
begin
     vyber z N záznam F takový, že F.I potřebuje nejmenší rozšíření na I'
                a E.I ⊆ F.I' a o(F.I') je nejmenší;
     N := F.p;
end;
L := N;
```

Algoritmus 3 - vybrání vhodného listu při vkládání nového prvku do R-stromu

Algoritmus: AdjustTree(L, LL, E)
Vstup: listy *L* a *LL*
Výstup: Rekonstruovatelný R-strom (případně bez nového kořene)
Předpoklady: R-strom s kořenem v *T, P* je rodičovský uzel *L*

```
N := L;
NN := LL;
while (N ≠ T) do
begin
     změň N.I tak, že obsahuje všechny kvádry v N;
     PP := NIL;
```

```
     if (NN ≠ NIL) then
     begin
          vytvoř nový E_NN, kde E_NN.p = NN a E_NN.I pokrývá každý kvádr z NN;
          if (P není plný) then
               přidej E_NN do P;
          else
               SplitNode(P, PP, E_NN);
     end;
     N := P;
     NN := PP;
end;
LL := NN;
```

Algoritmus 4 - zarovnání modifikovaného R-stromu

Algoritmus: SplitNode(P, PP, E)
Vstup: Uzel *P*, nový uzel *PP*, indexový záznam *E*
Výstup: *P*, *PP* s přerozdělenými záznamy
Předpoklady: minimální naplnění uzlu *m*

```
PickSeads;                          {vybere první E_i resp. E_j do P resp.
PP}
while (všechny záznamy nejsou vyčerpány) do
begin
     if (nevyčerpané uzly lze doplnit do P či PP tak, že uzel naplní
právě
          na m záznamů) then
          udělej to;
     else
          PickNext;                          {dodá další E_i do P nebo
PP}
end;
```

Algoritmus 5 - rozdělení přeplněného uzlu R-stromu

Algoritmus: PickSeads

```
foreach E_i, E_j (i ≠ j) do
     d_ij := o(J) - o(E_i.I) - o(E_j.I), kde J je kvádr pokrývající E_i.I a
          E_j.I;
vyber E_i a E_j s maximálním d_ij a přidej do P resp. PP
```

Algoritmus 6 - nalezení nejkritičtějších prvků v uzlu při štěpení

Algoritmus: PickNext

```
foreach zbývající E_i do
begin
     d_1 := přírůstek pro pokrytí kvádru zahrnující kvádr z P a (E_i.I);
     d_2 := přírůstek pro pokrytí kvádru zahrnující kvádr z PP a (E_i.I);
end;
vyber E_j s maximálním |d_1 - d_2|;
přidej E_j do P resp. PP tak, aby bylo rozšíření nejmenší;
```

Algoritmus 7 - vybrání dalšího nejkritičtějšího prvku z ještě nepřerozdělených prvků při štěpení

Algoritmus pro vymazání určitého listu z R-stromu je popsán v pseudokódu (Algoritmus 8). V odborných textech není mazaní v R-stromu dostatečně podrobně popsáno a

je použit vágní termín *„nejvhodnější“*, v mé implementaci se pod pojmem *„nejvhodnější“* skrývá heuristika nalezení nejbližšího uzlu s největším překrytím.

Algoritmus: Delete_R(T, L)
Vstup: R-strom s kořenem v *T*, mazaný list *L*
Výstup: Korektní R-strom bez listu *L*
Předpoklady: *P* je rodičovský uzel *L,* minimální naplnění uzlu *m*

```
N := L;
Odstraň L z P;

while (N ≠ T)
begin
     E := NIL;
     E := F ∈ P takové, že F má větší naplnění než m a F je nejblíže E;
     if (E != NIL) then
     begin
          přesuň z E do N takový nejvýhodnější prvek;
          přepočítej pokrytí E;
     end
     else
     begin
          E := F ∈ P takový, že vzdálenost E a F je nejmenší;
          Všechny prvky E přidej do N a odstraň prázdné E;
     end;
     Přepočítej pokrytí N;
     N := P;
end;
if (N má menší naplnění než v [0, m-1]) then
     uzly 1. úrovně nahraď je uzly z 2. úrovně;     {slití poduzlů
kořene}
přepočítej pokrytí N;
```

Algoritmus 8 - vymazání prvku z R-stromu

Algoritmus pro konstrukci celého R-stromu s plným naplněním je uveden ve funkci *Pack* (Algoritmus 9), která postupným rekurzivním voláním staví R-strom od listů až ke kořenu. Jádrem je funkce *NS*, která vrací ze seznamu prvků takový prvek, který je prostorově nejblíže prvku uvedeného ve druhém parametru.

Algoritmus: Pack(Olist)
Vstup: Seznam objektů Olist
Výstup: Kořen R-stromu *T*

```
if (|Olist| = m) then
begin
     vytvoř uzel N obsahující indexové záznamy pro |Olist| objektů;
     Return(N);
end
else
begin
     uspořádej Olist podle určitého prostorového criteria (např. podle x)
     Nlist = Ø;                                {inicializuj prázdný seznam}
     while (Olist ≠ Ø) do
     begin
```

```
            I[1] := první prvek z Olist;
            Olist := tail(Olist);
            I := 1;
            while (i < m - 1) do
            begin
                  I[i] := NS(Olist, I[1]);      {vrať nejbližší prvek I[1]}
                  i := i + 1;
            end;
            vytvoř uzel N obsahující indexové záznamy pro m objektů v I;
            Nlist := Append(Nlist, N);                    {přidej nový
uzel}
      end;
      return(Pack(Nlist));
end;
```

Algoritmus 9 - konstrukce R-stromu ze seznamu prvků

5.8.8 Doménový vs. externí index

Při výběru vhodného databázového systému, kde bude implementovaná diplomová práce byly uvažovány databázové systémy MySQL, MSSQL a Oracle. V systémech MySQL a MSSQL není žádná podpora pro vytvoření vlastního indexu. Jediný systém, kde lze implementovat vlastní index uživatelem, je Oracle.

Při implementaci externího indexu jsem primárně čerpal z Dokumentace Oracle Cartridge 9, 9 a 9. Bylo nutné vyřešit tyto problémy:

- Nad každou tabulkou lze vytvořit nanejvýš jeden doménový index daného typu.
- Pro každý typ vstupního sloupce je nutné vytvořit novou hlavičku pro funkce indexu.
- S počtem povolených typů sloupců a počtem indexovaných sloupců roste počet hlaviček funkcí exponenciálně.
- Při implementaci ve více různých jazycích je nutné při konverzích řešit různé výstupy, neboť každý jazyk konvertuje podle jiných lokálních nastavení programu, databázového systému, OS atd.
- Při indexování dlouhých hodnot indexovaných atributů je nutné hodnoty vhodně *„oříznout“* a vždy vytvořit stejně dlouhý klíč.
- Ukládání všech struktur indexu na blokové zařízení, díky které bylo nutno struktury ručně serializovat.

Doménový index je v systému Oracle možno definovat pomocí rozhraní pro doménový index 9, kde je nutné definovat vlastní *indextype* 9. *Indextype* doménový index v Oracle je navržen pro vytvoření vlastního indexu nad jednou tabulkou, což v případě jeho

použití při indexaci hodnot z více tabulek přináší jistá omezení.

Asi nejzásadnější je právě vytvoření indexu, který se pro systém Oracle musí navenek tvářit jako index nad jednou tabulkou. To se zpravidla provádí tak, že se index vytvoří nad nějakou abstraktní tabulkou, kde se nacházejí vyhledávaná data. V této práci by to byla tabulka *SYS_OBJECT* pro dotazy vyhovujících instancí objektů podle hodnot atributů. Vytvoření jednoho doménového indexu nad jednou tabulkou pro indexovaní určité *k*-tice atributů lze provést snadno. Problém však nastává v případě, že se uživatel pokusí indexovat jinou *l*-tici atributů. Zde již operace selže, neboť v Oracle je dovoleno nad jednou tabulkou a jednou fiktivní množinou sloupců definovat pouze jeden index daného typu. Z tohoto důvodu je nutné vytvořit nový indextype i se všemi funkcemi a operátory s druhý index vytvořit s tímto novým typem indexu. Navíc – protože jsou fakticky všechny indexy definované nad jednou tabulkou – je nutné vždy generovat nová jména operátorů pro daný indextype, aby Oracle podle jména operátoru určil, jaký index v dotazu použít. Při definici například porovnávacích operací je nutné skutečně vyplnit i funkce, které mají porovnávání provádět, i když v indexu musí být totožné operace definované také. Tedy výsledkem je redundantní implementování porovnávacích operací, které v indexu nad více tabulkami není zcela triviální.

Kromě výše popsaných problému při vytvoření indexu je nutné uvědomit si další omezení. Při skutečném vyhledávání u dotazů s více operátory se do indexu předává pouze jméno a parametry z prvního operátoru. Použití druhých a další operátorů provádí systém Oracle sám na základě přidružených funkcí k daným operátorům. Tedy například u složeného intervalového dotazu to znamená použití indexu pouze na interval v prvním operátoru a potom samotné dohledání a porovnání hodnot v parametrech druhého operátoru. V případě použití spojky or mezi operátory je výsledkem vyhodnocení prvního operátoru podle indexu a druhý se prochází již bez použití indexu. S tímto nestandardním chováním jsem se setkal pouze při používání doménového indexu. Toto chování, které je pro uživatele nežádoucí lze odstranit pouze definováním nového operátoru a přidružené funkce pro dvoj interval, případně troj interval a tak dále.

Navíc, jak již bylo v úvodu této sekce zmíněno, pro každý typ sloupce je nutné definovat nové hlavičky funkcí.

Tyto a další problémy spojené s implementací jsem se snažil řešit. Řešení není založené na použití skutečného doménového indexu, neboť jeho svázanost s jednou tabulkou poměrně velká, ale na využití vlastního mechanismu, který z něj vychází a některá omezení eliminuje. Byly implementované 3 základní funkce jimiž se indexová struktura ovládá:

createInxStruct, reInsert a *findStruct*.

Struktura implementovaného externího indexu odpovídá navržené teoretické struktuře (5.8.6). Data indexu jsou uložena v tabulce se dvěma sloupci s klíčem a blobem. Vícerozměrný R-strom je uložen v blobu s klíčem číslo 0, ostatní řádky s kladnými hodnotami klíče jsou použity pro uchování pomocné struktury pro objekt s id stejné hodnoty. Není proto nutné implementovat hashmapu pomocné struktury. Potřebná data lze podle id objektu dohledat v blobu se shodným klíčem pomocí prostředků v Oracle. Rozdělením pomocné struktury pro jednotlivé objekty jsou jednotlivé části velice malé a proto mohou být ukládány jako nesetříděné pole. Nultý řádek s R-stromem je velký a je nutné jej implementovat i s patřičnou správou volných bloků obdobně, jako na blokovém zařízení. První blok slouží k uchování metadat o uloženém R-stromě. Další bloky jsou buď prázdné a nebo obsahují uzly R-stromu. Prázdné místo je reprezentována spojovým seznamem, kdy každý volný blok obsahuje číslo dalšího volného bloku. Číslo prvního volného bloku lze získat z metadat. Poslední prázdný blok obsahuje hodnotu 0. Uzly R-stromu jsou dvojího druhu. Listové uzly, nesoucí skutečná data a nebo uzly vnitřní, která data nenesou. Tedy skutečné listy R-stromu jsou uzly předposlední úrovně. Struktura následníků v uzlu R-stromu je u datových položek využita pro uchování dat. Nad uzly je v R-stromu implementovaná hasmapa ukazatelů na právě načtené a v paměti se nacházející bloky, sloužící jako cache. Na konci každé funkce je vyprázdněna a její část obsah se uloží. Ukládají se pouze modifikované uzly R-stromu, ostatní se pouze z paměti uvolní.

Funkce *createInxStruct* vytvoří a uloží strukturu metadat a jedem prázdný uzel, reprezentující kořen R-stromu. Funkce *reInsert* v první fázi provede vyčištění R stromu od nedefinovaných hodnot podle pomocné struktury. V druhé fázi vytvoří disjunktní intervaly, které uloží do R-stromu a aktualizuje pomocnou strukturu. Nakonec sesype pomocnou strukturu do jednotného pole i se všemi modifikovanými uzly ji uloží do blobu. Funkce *findStruct* reprezentuje funkci intervalového hledání, kdy jsou podle potřeby načítány uzly R-stromu a vyhovující listové záznamy jsou uloženy do tabulky výsledků. V důsledku tohoto návrhu byl počet volání externích funkcí minimalizován, neboť se nemuselo volat všech 7 funkcí, které jsou definované v šabloně pro doménový index.

Malý počet externích volání nese úsporu v času, v inicializaci prostředí a především umožňuje jednou načtenou strukturu části R-stromu vícekrát využít. Navíc v tomto návrhu lze alespoň částečně řešit vícenásobné intervalové dotazy, které u doménového indexu nešly řešit vůbec. V poslední řadě úložiště není zahlceno řadou funkcí a operátorů, jejichž počet roste lineárně s počtem použitých indexových typů. Tím, že nebyl použit doménový index, ale

jakýsi externí index a s pouze 3 funkce, musí uživatel sám explicitně říci jaký index použít. Nicméně explicitnímu stanovení použitého indexu v dotazu by se uživatel nevyhnul ani v případě doménových indexů, neboť by musel volit jména operátorů, s jednotlivými indexy svázanými.

5.8.9 Diskuze implementace externího indexu

Jak již bylo nastíněno v podsekci (5.8.8), během implementace bylo nutné řešit několik problémů spjatých jak s použitím systému Oracle, tak i ukládáním na blokové zařízení.

Největší problém představoval až exponenciální nárůst hlaviček funkcí podle počtu indexovaných typů a počtu atributů. Z tohoto důvodu jsem se při vytváření vlastního indexu omezil na indexování pouze 2 atributů. Bohužel omezení pouze počtu indexovaných atributů nestačilo, neboť například již pro 2 povolené typy by to vedlo k definici 4 funkcí pro *reInsert* a 4 funkcí pro *select*.

Z tohoto důvodu jsem se v pilotní implementaci omezil na typ, který je pevně v úložišti stanoven na *varchar2*. Jedná se o nejobecnější typ a jakýkoliv typ lze na tento typ implicitně konvertovat. Ač se může zdát, že pevné omezení typu by se mohlo týkat pouze vlastního indexu, implementace v různých prostředí PL/SQL a C++ sebou nese například i jiné zpětné konverze. Například se jedná o čárky či tečky v desetinných číslech, jiný formát výpisu data a času. Bylo tedy nutné zaručit, aby se konverze prováděly v právě jednom prostředí, neboť jen tak je zaručeno, aby stejný vstup dával vždy totožný výsledek. Proto je napevno stanoven typ *varchar2* pro ukládání hodnot atributů.

Vlastní index umožňuje indexovat s časovou složkou 3-dimenzionální kvádry jako klíče. Indexované hodnoty jsou id instancí objektů. Jedna dimenze je vždy reprezentována uzavřeným intervalem dvojicí *unsigned long longů*, kdy první složka je vždy menší rovna druhé. Indexace pomocí těchto typů přináší dva zásadní problémy. Prvním je indexace pouze prefixu indexovaných hodnot a tedy při vyhledávání je nezbytné u dlouhých hodnot učinit i následné ověření v samotných tabulkách atributů, zda jsou hodnoty skutečně totožné. Druhým problémem je vytvoření z textového řetězce uspořádaný celočíselný klíč. Právě uspořádanost je nutná k tomu, aby fungovaly intervalové dotazy. Konverze z hodnoty atributu na část klíče a zpět se provádí postupným procházením od předu vstupního *stringu*, násobením předchozí hodnoty velikostí rozsahu pro jeden znak (*char)* a přičtením ordinální hodnoty následujícího znaku. Výsledkem je tedy část klíče, která definuje správně lexikografické uspořádání v anglickém jazyce a i číselné uspořádání, což jsou asi nejpoužívanější typy atributů. Pro

zachování třídění podle daného jazyka byla zvažována funkce *NLSSORT*, která ve finální verzi zaimplementována, neboť třídí pouze textové řetězce. Například pro vstupní *stringy* „2“ a „10“ vypadá výpočet následovně. Pro „2“ je klíč roven 50, protože „*ord*(‘2’) = 50“. Pro „10” je klíč roven 12592, protože „*ord*(‘1’) * 256 + *ord*(‘0’) = 12592“. A tedy vygenerované klíče v indexu zachovají uspořádání. To platí pokud uživatel v data do jednoho atributu vždy ukládá hodnoty jednoto typu. Problém však nastává u čísel desetinných, kde tato konverze nefunguje korektně. Pro intervalové dotazy je nutné ukládat čísla vždy jako celé, násobená určitým koeficientem.

5.8.10 Soubory implementovaného indexu

Index byl implementován v jazyce C++, a tedy význam jednotlivých souborů se řídí běžnou konvencí, v souborech s příponou **.h* jsou hlavičky a samotná implementace ne v souborech **.cpp*.

Rozhraní pro používání nízkoúrovňového OCI bylo obaleno objekty v souborech *rsbuzy*. Blíže je popsané i s ukázkami použití v příloze (9).

V souborech *rsdim* jsou implementovány operace pro manipulaci s celými klíči v podobě 3 dimenzionálních kvádrů.

Sext obsahuje konverzní a ořezávací operace pro tvorbu klíče. Tedy z hodnoty atributů generuje unikátní část klíče. V případě, že by uživatel požadoval jiné uspořádání je nutné předefinovat funkce právě v těchto souborech.

V souborech *rstree* je hlavní část logiky. To znamená operace pro manipulaci s R-stromem, jeho serializaci na blokové zařízení, cache a správu volných bloků.

V souborech *rsinx* jsou 3 klíčové externí funkce volané z Oracle na ovládání indexu. Jsou to funkce *createInxStruct, reInsert* a *findStruct*.

V souborech *rsinter* jsou podpůrné funkce indexu při vývoji, jako je vlastní logovací a nástroj v podobě přesměrování standardního výstupu do souboru a profiler pro zjištění úzkých hrdel funkcí.

V poslední řadě všechny konstanty a nastavení nutná pro chod indexu jsou v souboru *rsconsts*.

5.8.11 Vyhledání podle externího indexu

Indexovat tímto druhem indexu je možné právě 2 atributy z důvodů diskutovaných v podsekci (5.8.9). V ukázce použití tohoto indexu předpokládejme, že byl nejprve vytvořen

abstraktní index podle kódu (Kód 7). Z tohoto *abstraktního indexu* byl vytvořen *skutečný index* s přepínačem *INX_TYPE_RO* (9) místo *INX_TYPE_ORA* (9). V ukázkové dotazu (Kód 9) je vyhledání id objektů podle hodnot atributů. Uvedený příklad hledá ve dvou intervalech. První interval je reprezentuje dotaz na rovnost a druhý skutečné intervalové hledání. Výsledky obou dotazů jsou sjednoceny modifikátorem *INX_MOD_APPEND* (9). Výsledkem ukázkového dotazu je pouze id objektu váza, neboť pan Josef nemá definovaný atribut *MAJITEL*.

```
01   EXECUTE RS_INX.SELECT_EQUAL('SEZNAM_VAZ_IX', 'ZLUTA', '1',
          TO_TIMESTAMP('1995-03-20', 'YYYY-MM-DD'));
02   EXECUTE RS_INX.SELECT_IN('SEZNAM_VAZ_IX', '1', '1', 'HNEDA',
'HNEDA',
          TO_TIMESTAMP('1995-02-15', 'YYYY-MM-DD'),
          TO_TIMESTAMP('1995-02-17', 'YYYY-MM-DD'),
          RS_INTER.INX_MOD_APPEND());
03
04   SELECT OBJECT_ID FROM SEZNAM_VAZ_IX;
```

Kód 9 - vyhledání podle vlastního indexu

5.9 Omezení implementace

V této kapitole jsou shrnuta omezení pilotní implementace přikládané k diplomové práci. Všechna omezení byla stanovena tak, aby šlo práci doimplementovat v ročním časovém období. Žádné omezení nebylo nutné dělat z důvodu špatného návrhu.

5.9.1 Hodnoty null

Hodnoty atributů *null* jsou v přikládané implementaci zcela zakázané. Například funkce *GET_VALUE* (9) vrací hodnotu *null* jako nedefinovaný atribut. Navíc při implementaci indexů by bylo nutné ošetřovat problémy s prázdnými hodnotami, viz 9.

Díky zakázání *null* hodnot, tedy nelze rozlišovat případ *„daný objekt nemá danou vlastnost“* od *„daný objekt má danou vlastnost, ale není známa“*. Tj. informace [id=158;barva_očí=modrá] a [id=158;barva_očí=*null*] je odlišná od situace, kdy dvojice s id=158 není definována. Toto rozlišování může být pro jisté aplikace užitečné.

5.9.2 Typ obecného atributu

Neukládání *null* hodnot a nutnost vytvoření vlastního indexu vedla k omezení typu obecného atributu na *varchar2*. Navíc externí procedury psané v c++ musí mít definovaný typ

parametru, což by v případě definování více typů atributů vedlo až k exponenciálnímu růstu hlaviček externích procedur (diskutováno v 5.8.8). Z těchto hlavních důvodů byl typ pro *obecný atribut* omezen právě na typ *varchar2*. Toto striktní definování by nemělo mít kromě efektivity žádný vliv na implementaci úložiště. Efektivita bude v případě použití snížena o nutnou implicitní konverze z jiného typu na *varchar2*. Tyto konverze dělá v případě nutnosti Oracle automaticky a uživatel tak není nucen tento problém řešit, i když je vhodné používat právě typ *varchar2*, aby nedocházelo ke zbytečným implicitním konverzím.

5.9.3 Jednoduché typy

Rozšíření o možnost obecných uživatelem definovaných typů pro atributy by nemělo být pro stávající návrh implementace bez použití indexů problémem. I když tato možnost nebyla prakticky nikdy vyzkoušena. Ač je u *obecných atributů* možnost určit typ daného atributu, jedná se pouze o informaci pro uživatele v jakém formátu data pravděpodobně budou uložena. Neboť kvůli implementaci indexů se u obecných atributů používá pouze typ *varchar2*.

5.9.4 Časová složka

Předkládané úložiště podporuje pouze lineární časovou složku. Cyklická časová složka nemůže být použita, protože při indexaci a optimalizaci je nutné, aby byla časová složka uspořádaná. Bitemporální složka nebyla ani během návrhu uvažována.

5.9.5 Korektní index

Nekorektní index je takový, který obsahuje víc než povolené množství sloupců (ve stávající implementaci maximálně 4 pro interní index Oracle, pro externí index právě 2 sloupce). Index je také nekorektní, pokud nad nějakým atributem je vytvořeno více než povolený počet *skutečných indexů* (maximálně 4). Tato omezení vznikla při vývoji úložiště i s indexy a jsou stanovena především pro ochranu samotného uživatele, aby zbytečně neindexoval všechny atributy.

Tato omezení lze změnit upravením rozsahů v privátních konstantách *MAX_INX_OVER_ATTR* a *MAX_INDEXED_ATTRS* v balíku *RS_INX* (9).

6 Měření výkonu

V rámci testování se kromě stability implementovaného systému testovalo především chování systému při práci s větším množstvím vstupních dat. Tyto testy byly prováděny v první fázi tak, že se opakovaně zpracovávala malá množina dat. V druhé fázi se naopak jednorázově zpracovávala velká množina dat. Mnohonásobné spouštění na velkých datech testováno nebylo z důvodu časové náročnosti.

Měření výkonu úložiště je pro úložiště rozvolněných objektů velmi důležité, neboť ukládání dat samostatně do tabulek negativně ovlivňuje operace nad daty. Důvodem je pro každou *k*-tici procházení *k* tabulek místo například jedné. Tento fakt hraje roli při vykonávání dotazů v současných relačních databázích. Naštěstí lze podle 9 předpokládat, že se často používá pouze malá podmnožina dat a tedy je v dotazech nutné spojovat relativně malé množství tabulek. Nicméně – ve srovnání s klasickým uchováním v normalizovaném databázovém schématu – se čas nutný k operacím nad rozvolněnými objekty podstatně horší. Na druhou stranu je třeba zmínit, že právě klasické databáze a nad nimi pracující aplikace musí znát předem strukturu uchovávaných dat. Její modifikace během používání je velmi drahá a obtížná.

6.1 Hardware

Všechny testy byly prováděny na databázovém serveru Oracle Databáze 10g Release 2 Express Edition. Server běžel na notebooku DELL s operačním systémem Windows XP. Testovací počítač měl následující konfiguraci:

Procesor:	Intel® Pentium® M, 1,70GHz
RAM:	2 GB, DDR 2
HDD:	IC25N, 30GB 5400rpm
OS:	Windows XP SP3 Home Edition Cz.

6.2 Náročnost přepočítání indexů

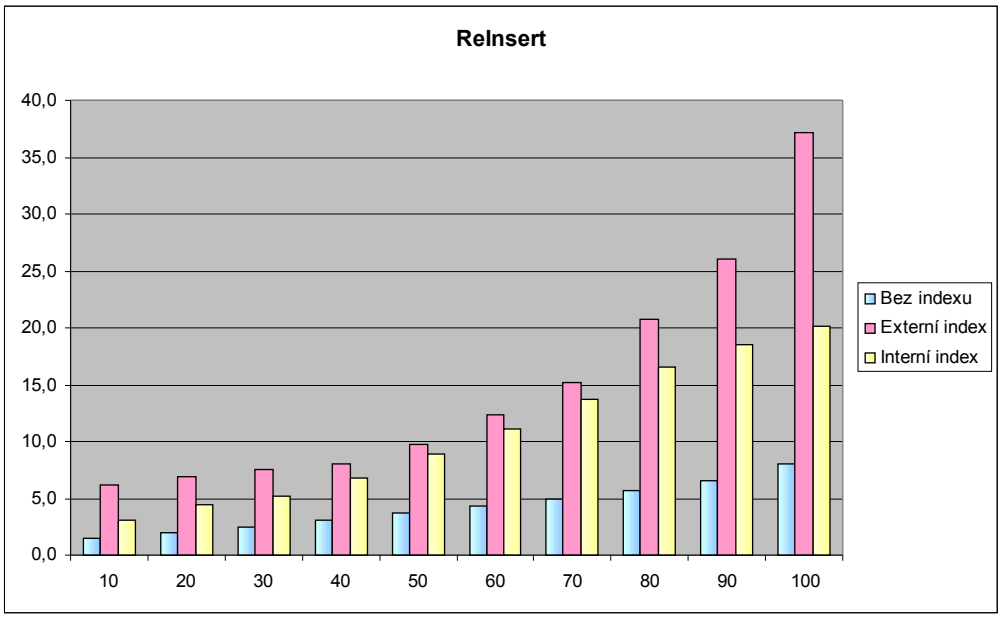
Jedna z nejdražších operací v implementovaném úložišti byla operace přepočítání indexů při odemykání modifikovaných instancí objektů. Proto jsem se věnoval optimalizaci převážně této operace. Pro testovací účely bylo postupně vytvořeno od 10 do 100 instancí objektů, kterým byly definovány tři atributy, jejichž hodnota se v čase měnila. Každé instanci bylo definováno celkem 25 různých hodnot atributů. Časy platnosti jednotlivých hodnot

atributů byly určeny tak, aby vytvořily 23 disjunktních intervalů pro klíče v indexech. Tedy například vytvoření 10 instancí objektů a definování jim 25 různých hodnot atributů znamená v řeči databáze 10 * 25 operací *insert* do různých tabulek bez použití indexu. V případě použití indexu se jedná o 10 * 25 operací *insert* do tabulek atributů, poté o až 10 * 2 * 25 operací *select* na vytvoření disjunktních intervalů a nakonec o 10 * 23 operací vložení disjunktních intervalů do indexu. Při vytváření disjunktních intervalů jsou pro všechny *k*-tice v daný časový interval sledovány čas počátku a konce intervalu, proto je u operace *select* při vytváření disjunktních intervalů koeficient 2. Naměřené výsledky jsou uvedeny v tabulce (Tabulka 20), v levém sloupci jsou uvedené počty vytvořených instancí, kterým byly definovány výše popsané atributy. V ostatních sloupcích je uveden spotřebovaný čas v sekundách.

Počet objektů	Bez indexu	Externí index	Interní index
10	1,5	6,2	3,1
20	2,0	6,9	4,4
30	2,5	7,5	5,2
40	3,1	8,0	6,8
50	3,7	9,8	8,9
60	4,3	12,3	11,1
70	5,0	15,2	13,7
80	5,7	20,7	16,5
90	6,5	26,0	18,5
100	8,0	37,2	20,1

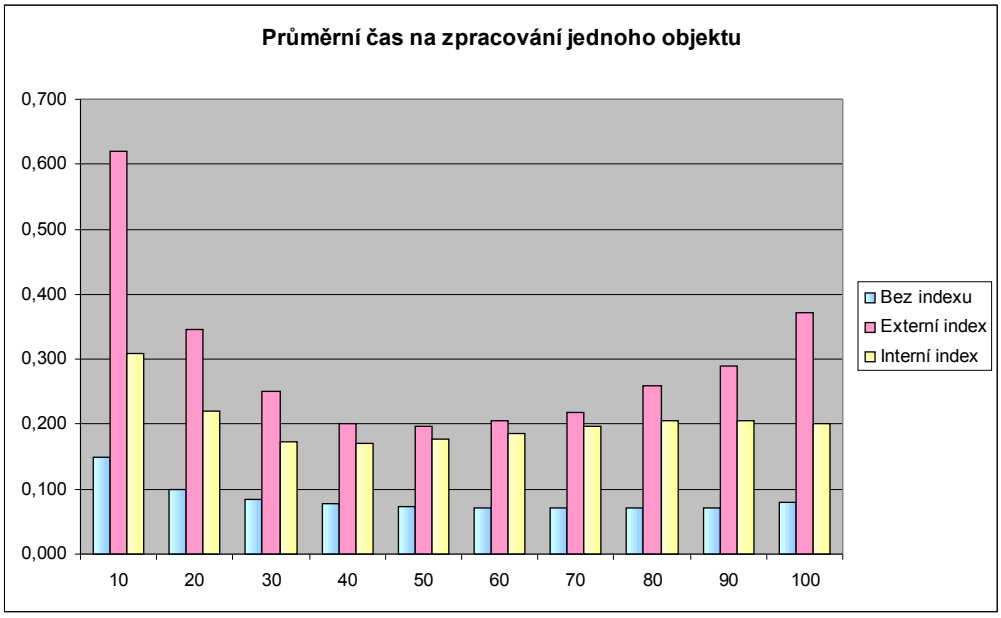
Tabulka 20 - celkový čas i s přepočtením katalogu a indexu

Grafické znázornění výsledků z tabulky (Tabulka 20) je v grafu (Graf 1). Na horizontální ose grafu je počet zpracovávaných objektů a na vertikální ose je potřebný čas v sekundách. Pořadí sloupců grafu odpovídá pořadí sloupců v tabulce. Tedy první sloupec v grafu (druhý v tabulce) je provedení operací bez indexu v úložišti, druhý sloupec (třetí v tabulce) je s *externím indexem* a poslední sloupec je s *interním indexem*. Indexy byly vytvořeny nad dvěma atributy. Prováděné operace byly vytvoření určitého počtu instancí a nadefinování jim 25 různých hodnot atributů. Prováděné operace byly s použitím indexů mnohem pomalejší, což v případě externího indexu bylo primárně způsobeno voláním externího kódu a ukládáním R-stromu do blobu. V případě interního indexu je zpomalení způsobeno tím, že musely být disjunktní intervaly nakopírované do samostatné tabulky, nad kterou byl index vytvořen.



Graf 1 - celkový čas i s přepočtením katalogu a indexu

V grafu (Graf 2) jsou potom naměřené hodnoty normalizovány, kolik času bylo průměrně zapotřebí na zpracování jedné instance objektu. V tomto grafu je na horizontální ose opět počet zpracovávaných instancí a na vertikální ose je průměrný čas nutný ke zpracování jedné instance v sekundách. Z tohoto grafu je patrné, že počáteční připojení externí knihovny inicializace trvá až ⅓ spotřebovaného času. Navíc je v grafu vidět, že volání externího kódu není laciné a tedy cíl minimalizace počtu volání externího kódu byl kladen oprávněně.



Graf 2 - průměrný čas pracování jednoho objektu

Bohužel výsledky externího indexu jsou nejhorší, to může být pravděpodobně použitou strukturou R-stromů a ukládáním stromu do blobu. Právě při procházení kódu a hledání úzkých hrdel bylo zjištěno, že 98% času běhu externí funkce bylo spotřebováno na načítání a ukládání uzlů R-stromu. Z tohoto důvodu byl do cache s uzly R-stromu přidán indikátor, zda byl daný uzel modifikován a zda je nutné jej serializovat do blobu. Během testování a ladění externího indexu bylo zjištěno, že vhodné nastavení parametrů pro externí index v podobě velikosti uzlu R-stromu může ovlivnit výkon indexu v rozsahu pozorovaných 60% až 250%. Velikost uzlu je primárně ovlivněna maximálním počtem následníků. Při měření výkonu bylo zjištěno, že implementovaný R-strom nebyl zcela vhodný a to především, kvůli slabším heuristikám než má R*-strom. Naštěstí se to projevuje pouze v případě ukládání degenerovaných dat do stromu. Na druhou stranu, laciná režie na správu kompletního stromu umožňuje v optimálních případech i přes nutné externí volání dosahovat při *reInsertu* velmi podobných výsledků jako má interní index Oracle.

6.3 Vyhledávání v úložišti

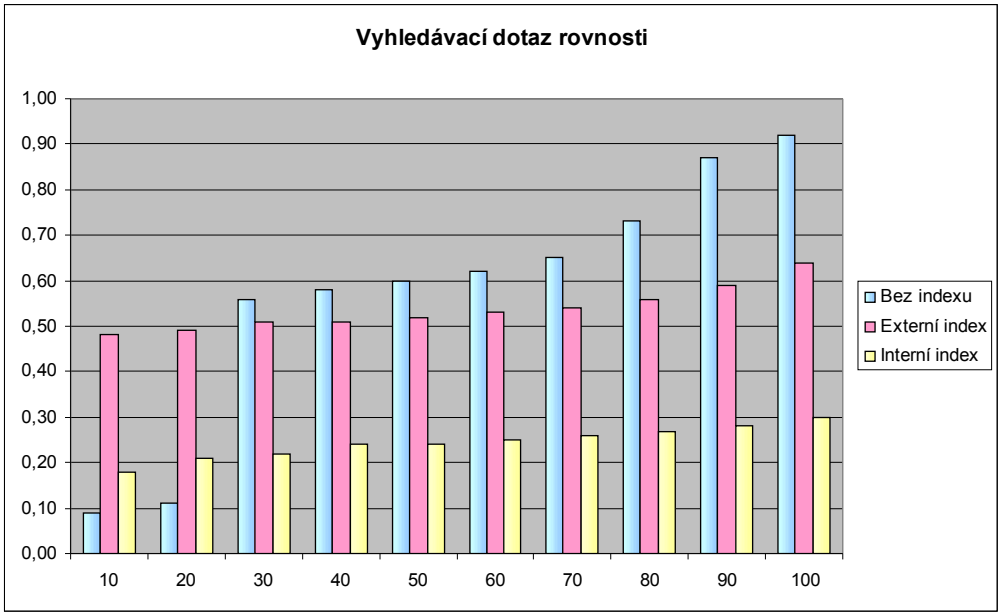
Úlohou indexů v úložišti je zefektivnit vyhledávání. Pozorování času potřebného k vyhledání instancí objektů na základě hodnot atributů v určitý časový interval a naměřené hodnoty jsou v tabulce (Tabulka 21). Úložiště bylo postupně naplněno daty ze sekce (6.2). V úložišti byl kladen nejmenší intervalový dotaz, tedy dotaz rovnosti. Naměřené hodnoty jsou uvedené v sekundách. Sloupce v tabulce postupně znamenají vyhledání rovnosti bez použití indexu, s externím indexem a interním indexem.

Počet objektů	Bez indexu	Externí index	Interní index
10	0,09	0,48	0,18
20	0,11	0,49	0,21
30	0,56	0,51	0,22
40	0,58	0,51	0,24
50	0,60	0,52	0,24
60	0,62	0,53	0,25
70	0,65	0,54	0,26
80	0,73	0,56	0,27
90	0,87	0,59	0,28
100	0,92	0,64	0,30

Tabulka 21- vyhledání rovnosti

Grafická vyobrazení naměřených hodnot je ukázáno v grafu (Graf 3). Horizontální osa je počet objektů v úložišti a vertikální osa je spotřebovaný čas v sekundách. V grafu je vidět,

že při malém naplnění tabulky jsou indexy zbytečné, protože se tabulky procházejí fullscanem, což je nejrychlejší. Tedy indexy má smysl používat až při určitém množství dat v paměti. V případě měření se indexy projevily až při naplnění úložiště 30 instancemi objektů s různými hodnotami atributů. Osobně jsem očekával, že hranice využití bude o něco výše, ale je potřeba si uvědomit, že hraniční naplnění je cca 1500 záznamů ve fyzických tabulkách. V grafu je patrné, že nárůst spotřebovaného času je v případě použití indexů lineární, zatímco při vyhledání bez indexu je nárůst spotřebovaného času vyšší. Nelineární nárůst při vyhledání v úložišti bez indexu je pravděpodobně způsoben spojováním tabulek, ve kterých jsou data uložena. Z měření nejlépe vychází interní index Oraclu. To může být odůvodněno dvěma fakty. Prvním je použití lépe vyvážené struktury R*-stromů v interních indexech Oraclu. Druhým důvodem je to, že volání externích funkcí něco stojí a tak zkreslení výsledků je v malých časech velké.



Graf 3 - vyhledávací dotaz rovnosti

6.4 Diskuze naměřených hodnot

V první řadě je nutné si uvědomit, že naměřené výsledky byly získány z měření provedených na běžném osobním počítači, kde i přes vypnutí všech aplikací běží několik procesů na pozadí na rozdíl od serveru databáze, kde zpravidla běží pouze OS a databázový systém. V důsledku procesů na pozadí a přeplánování tak dochází ke zkreslení výsledků, které je v měřených řádech stovek či desítek milisekund nezanedbatelné. Navíc měřené hodnoty nejsou skutečné naměřené hodnoty, ale průměr z naměřených hodnot, který by sice měl

nejvíce odpovídat očekávané ceně, ale v praxi to tak být nemusí. Důvodem je to, že si Oracle vykonané dotazy cache a jejich plány použije při dalším vykonávání. Tedy získat skutečné ceny prováděných operací může být někdy netriviální. V případě volání externího kódu lze navíc pozorovat velkou režii spojenou s načtením a inicializací služby extproc, která v některých případech činí až 10 sekund.

Nicméně naměřené hodnoty alespoň částečně vypovídají o tom, jak přibližně efektivní jednotlivé operace jsou. A jaká je tendence chování úložiště při použití indexů a bez indexů. V úložišti bez indexů je modifikace atributů velmi levná, problém však nastává při vyhledávání ve velkém úložišti, protože cena dotazu na vyhledání roste nelineárně. To může ve velkých úložištích být značný problém, neboť rychle uložená data jsou velmi pozitivním výsledkem, ale těžko dostupná data jsou uživateli zpravidla k ničemu. Při použití indexů je vyhledání úměrné počtu instancí a atributů v úložišti, i když cena modifikujících operací velmi vzroste.

Z měření vychází nejlépe interní index Oracle, ale je nutné zmínit, že tento index je oproti internímu indexu 2x až 7x tak velký. Navíc interní index není obsluhován další službou, která tvoří middleware mezi knihovnou s externím kódem a systémem Oracle. Právě externí volání kódu stojí část spotřebovaného času.

Při implementaci a zjišťování kritických částí kódu bylo zjištěno, že 98% času tvoří ukládání a načítání uzlů R-stromu do blobu. Z tohoto důvodu bylo uvažováno přepsat externí index tak, aby pracoval s typem bfile. Bohužel práce s bfile je sice o něco rychlejší, ale není již transakční, což by v případě havárie systému mohlo ponechat index v nekonzistentním stavu, z tohoto důvodu bylo ponecháno původní ukládání do blobu 9.

Jako hlavní nevýhodu v implementovaném externím indexu spatřuji vhodnost manuálního nastavení parametrů pro R-strom, které může efektivitu indexu až řádově zrychlit či zpomalit. Vhodné nastavení parametrů je závislé na konkrétním hardwaru, kde je úložiště provozováno. Při použití interního indexu Oracle je nastavení parametrů automatické. Naopak výhoda externího indexu je jeho podstatně menší velikost oproti indexu Oracle.

Měření výkonu získání obsahu určité instance objektu nebylo prováděno, neboť díky katalogu lze obsah instancí získat přímo spojením tabulek atributů s vhodnými tabulkami katalogu. Výsledky měření by tedy nic nevypovídaly o efektivitě úložiště, ale o výkonnosti konkrétního databázového systému, kde bylo úložiště implementováno.

7 Rozšiřitelnost implementace

Jedním z cílu při programování implementace diplomové práce byla možná budoucí rozšiřitelnost. V této kapitole jsou diskutovány možnosti či nástroje, které se v dnešním době v databázových úložištích používají, ale z časových důvodů musely být z implementace vypuštěny.

7.1 Integritní omezení

Jedno z hlavním integritních omezení, které by bylo vhodné, je mít možnost definovat doménu hodnot pro každý atribut. Toto lze v zásadě provést dvěma možnými způsoby. Prvním způsobem je přidání *check* omezení, například pomocí regulárního výrazu, pomocí něhož by byla hodnota daného atributu definována. Druhým způsobem je potom obecnější definování funkce, která podle návratové hodnoty určuje, zda je určitá hodnota atributu validní či nikoliv. Pomocí takové validace hodnoty atributů podle funkce je možné nadefinovat libovolné omezení v širším pojetí. Například lze definovat, že hodnota nějakého atributu musí být větší než hodnota jiného atributu u dané instance objektu.

Žádná z uvedených možností nebyla implementována z důvodu snížení výkonu úložiště. Uživatel tedy nemá pomocí vytvořených balíků možnost definovat integritní omezení v úložišti a validaci dat musí buďto definovat standardními prostředky úložiště, nebo hlídat v aplikaci. V úložišti nemůže hlídat validitu dat pomocí *triggerů* libovolný uživatel, protože celé úložiště je vlastněno specifickým uživatelem a je ovládáno skrze balíky s procedurami. Na jiný přístup k tabulkám uživatel nemá v úložišti práva. Samozřejmě, pokud si uživatel upraví zdrojové kódy a přidá si práva, na modifikaci tabulek, je možné do tabulek přidat *check* omezení, či *triggery*.

7.2 Práva

Rozvolněné objekty a jejich způsob uložení umožňuje stanovit granulitu práv až na úroveň jednotlivých atributů. V mé implementaci se celé úložiště ovládá pomocí procedur v jednotlivých balících, kde se vytváření jednotlivé tabulky pro definované atributy. Proto je vlastníkem tabulek vlastník úložiště a nikoliv uživatel, který volal danou proceduru, která atribut v úložišti vytvořila. Toto způsobuje běžnému uživateli potíže, pokud chce nastavit určitý atribut jako privátní, neboť takový uživatel pravděpodobně nebude mít nad tabulkou atributu právo *grant option*.

Jedním řešením je přihlásit se jako vlastník úložiště a nastavit práva pro tabulku reprezentující daný atribut. Nicméně tímto způsobem lze pouze skrýt hodnoty atributu, veškeré ostatní informace o instancích objektů a atributech budou uložené v katalogu a tedy volně přístupné ostatním uživatelům.

Práva tedy ve stávající implementaci podporována nejsou. Šlo by je pomocí další relace v katalogu jednoduše doimplementovat, ale to není předmětem této diplomové práce.

7.3 Reprezentace obecných faktů

Obecná fakta se v úložišti reprezentují tak, jak bylo popsáno v teoretické části. Pokud má být fakt obecný, znamená to, že má platit po celou dobu existence úložiště. Proto je nutné takovým faktům nastavit korektní interval času platnosti, který začíná na minimální povolené hodnotě pro temporální složku a končí na maximální hodnotě pro temporální složku. V úložišti je minimální hodnota *MIN_TIMESTAMP* (9) rovna 1.1.1900 (mínus nekonečno) a maximální hodnota *MAX_TIMESTAMP* (9) rovna 1.1.2100 (plus nekonečno). Rozsah lze změnit nastavením příslušných privátních konstant v balíku *RS_INTER* (9).

7.4 Havárie systému

Při implementaci byly ve všech systémových tabulkách úložiště i ve všech tabulkách pro definované atributy použity základní vestavěné typy Oracle. To dovoluje po havárii systému použít standardní nástroje Oracle pro zotavení a případnou rekonstrukci dat.

7.4.1 Reference na atributy

V implementaci lze vytvářet *referenční atributy* pouze jako reference na konkrétní instance objektu. V teoretickém případě by šlo povolit i reference na jednotlivé atributy či na nějaké části určitého atributu.To by mohlo mít smysl v případě, že by atribut mohl být nějaký složitější strukturovaný typ. Jiná praktická možnost odkazování se na atributy jednotlivých instancí není známa, a proto není ani v implementaci podporována. Navíc například používání strukturovaných typů by vedlo k jinému způsobu získávání dat takových atributů a možná i k jiné syntaxi. To by mohlo být pro uživatele nepřehledné. Ve stávajícím návrhu lze snadno uložit jakýkoliv strukturovaný typ jako nový rozvolněný objekt, na který se bude poté pouze odkazovat pomocí reference. Z vhodnosti uchování jednotného stylu při ukládání informací je omezení stanoveno na jednoduchý a vestavěný typ Oracle.

8 Závěr

Tato práce navazuje na práci Kopeckého a Žemličky 9, kterou po teoretické i praktické stránce rozšiřuje o temporalitu. Práce formálně popisuje takovýto rozšířený model pro ukládání rozvolněných objektů s časovou složkou. Podle tohoto návrhu bylo implementováno úložiště pro rozvolněné objekty, kde byly zkoumány možnosti efektivnějšího vyhledávání. Vyhledávání byla dvojího charakteru.

První bylo získávání obsahu instancí v určitý časový okamžik pro určitou instanci objektu. Pro tento problém byla navržena struktura katalogu úložiště. Každá dvojice atributu a instance objektu má v klíčové tabulce *SYS_OBJECT_ATTRIBUTE* záznam se signaturou, která umožňuje částečně zabránit zbytečnému procházení vlastních tabulek atributů při dohledávání obsahu konkrétní instance v určitý časový okamžik. Pokud je signatura vyhovující je nutné dohledat hodnotu atributu v tabulce pro daný atribut. To lze realizovat pouze jednou operací spojení.

Druhá – problematičtější – část se zabývá možností vyhledání instancí objektů podle příslušného datového obsahu v určitý časový okamžik. Byly zkoumány možnosti využití *interního indexu* Oracle a definování vlastního *externího indexu*. Za tímto účelem byla navržena obecná struktura indexu pro efektivní vyhledávání podle obsahu. Navržená struktura byla implementována v podobě *externího indexu*. Od původní ideje implementovat vlastní index jako *index doménový* bylo upuštěno z důvodů, diskutovaných v sekci (5.8.8). Výkonnost *externího indexu* a *interního indexu* Oracle byla porovnána v kapitole měření výkonu (6). *Interní index* Oracle dosahoval lepších výsledků než *externí index*, nicméně jeho velikost byla 2x až 7x větší.

Externí index dosahoval více méně uspokojivých výsledků. Jednalo se skutečně o vícerozměrný index nad sloupci více různých tabulek najednou, což současné databázové systémy neumožňují. Rychlost aktualizace dat a vyhledávání s použitím *externího indexu* by však ještě bylo možné zvýšit úpravou implementované struktury R-stromu, neboť bylo zjištěno, že 98% času trvá právě náhodné čtení uzlů R-stromu. Jako největší nevýhoda *externího indexu* byla stanovena vhodnost ručního nastavení parametrů, které ovlivňovalo pozorované výsledky indexu v rozsahu od 60% do 250%.

Bylo implementováno a rozšířeno kompletní úložiště pro rozvolněné objekty. Během testování a měření výkonu se ukázalo, že sice koncept ukládání rozvolněných objektů samostatně do relací není pro práci příliš efektivní, neboť při práci s daty je nutné spojovat velké množství tabulek. Nicméně většina aplikací pracuje jen s malou podmnožinou definovaných atributů. Právě práce s malou podmnožinou atributů vede ke spojování malého

množství tabulek, což rychlost prováděného dotazu ovlivňuje příznivě.

Navíc v implementovaném modelu lze uchovávat data z heterogenních zdrojů i data, která mají jinou vnitřní strukturu. A to bývá u současných systémů problém, který mnohdy vede buď k redundanci dat při uchování či částečné ztrátě dat.

Případně další vylepšení a rozšíření implementovaného úložiště by mohlo být, kromě lehce pozměněné struktury implementovaného *externího indexu*, například rozšíření odkazů a vztahů, které by bylo možné definovat nejen mezi objekty, ale například i mezi jednotlivými atributy. Potenciálních možností rozšíření úložiště je mnoho, proto jim byla věnována samostatná kapitola rozšiřitelnost implementace (7).

9 Literatura

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

[2] Kopecký, M., Žemlička, M. (2004): *Rozvolněné objekty*, Sborník konference Datakon '04, Brno

[3] Kulhánek J, Obdržálek D. (2006): *Generating and handling of differential data in datapile-oriented systems*, In proceedings of the 24th IASTED international conference on Database and applications, Innsbruck, Austria

[4] Tauberer J. (2008): What is RDF and what is it good for?, http://www.rdfabout.com/intro/.

[5] Žemlička, M. (1999): ACASE: *Programování s podporou specifikací rozhraní*, Sborník konference Objekty 1999, České zemědělská univerzita, Praha, 157-163.

[6] Žemlička, M., Brunclík, M., Brůha, V., Caha, P., Cuřín, J., Dědic, S.,Marek, L., Ondruška, R., Průša, D. (1998): *Dokumentace k projektu ACASE*. MFF UK, Praha.

[7] Paton. N. W. (1995): *Extending Database Technology, v: SOFSEM'95: Theory and Practice of Informatics*, Springer-Verlag, Heidelberg, 146-165.

[8] Kopecký, M. (2002): *Object Persistency in C++*. Master Thesis, MFF UK, Praha.

[9] Kopecký, M. (2009): *Administrace Oracle*, http://www.ms.mff.cuni.cz/~kopecky/vyuka/oracle/

[10] Tým (2009): *Hibernate Reference Documentation*, HIBERNATE – Relational Persistence for Idiomatic Java, http://www.hibernate.org/hib_docs/v3/reference/en-US/pdf/hibernate_reference.pdf

[11] Čermák M. (2008): *Index pro textové vyhledávání nad relačními daty*, diplomová práce, MFF UK, Praha

[12] Gorman T., Logix S. (2004): *Understanding Indexes*, www.evdbt.com/2004_paper_549.doc

[13] Kohan T. (2008): *Temporalne databazy*, Dotazovací jazyky I, MFF UK, Praha, http://www.ksi.mff.cuni.cz/~pokorny/dj/prezentace/2_53.ppt

[14] Kovalin S., (2008): *Temporalne databazy – priklady*, Dotazovací jazyky I, MFF UK, Praha, http://www.ksi.mff.cuni.cz/~pokorny/dj/prezentace/2_54.ppt

[15] Ulrych J., Urbánek V., Uher P. (2008): *Temporální databáze a TSQL*, Dotazovací jazyky I, MFF UK, Praha, http://www.ksi.mff.cuni.cz/~pokorny/dj/prezentace/2_72.ppt

[16] Ambler S. W. (1999): *Mapping Objects To Relational Databases*, AmbySoft Inc. White

Paper.

[17] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorenzem W. (1991): *Object-Oreiented Modeling and Design*, Prentice-Hall, Englewod Cliffs, New Persey

[18] Oracle (2005): Oracle Database: *Data Cartridge Developer's Guide*, 10g Release2

[19] Ajay Gursahani (2005): *Using Index Hints in SQL statements* - 'Cause we know more about our application than Oracle does http://www.dbasupport.com/oracle/ora9i/index_hints.shtml

[20] Oracle Tips by Burleson Consulting (2005): *Why doesn't Oracle use my index?*, http://www.dba-oracle.com/t_index_why_not_using.htm

[21] Oracle Tips by Burleson Consulting (2005): *Oracle optimizer_index_cost_adj and SQL Performance*, http://www.dba-oracle.com/oracle_tips_cost_adj.htm

[22] Oracle Tips by Burleson Consulting (2004): *Creating an Oracle index cache*, http://www.dba-oracle.com/art_so_optimizer_index_caching.htm

[23] Pokorný, J., Žemlička, M. (2005): *Základy implementace a souborů a databází*, MFF UK, Praha

[24] Beckmann, N. et all (1990*): The R*-tree: An Efficient and Robust Access Metod for points and Rectangles*, International Conf. on Management of Data, May, Atlanta City USA

[25] Server, B., Kriegel, H.-P. (1990): The Buddy-Tree: *An Efficient and Robust Access Metod for Spatial Data Base Systems*, Proc. of the 16th VLDB Conference, Brisbane Australia

[26] Guttman, A. (1984): R-trees: *a dynamic index structure for spantial searching*, Proc of SIGMOD int. Conf. on Management of Data

[27] Oracle® Database Data Cartridge Developer's Guide, 11g Release 1 (11.1), Part Number B28425-03 (2009): *Extensible Indexing Interface* http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28425/ext_idx_ref.htm#i78714

[28] Oracle® Database SQL Language Reference, 11g Release 1 (11.1), Part Number B28286-05 (2009): *Using Extensible Indexing*, http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/ap_examples001.htm

[29] Hunter, M. J. (2003): Oracle DBA Tips conner: *Introduction to Oracle LOBs*, http://www.idevelopment.info/data/Oracle/DBA_tips/LOBs/LOBS_1.shtml

Příloha A Uživatelská příručka

Úložiště je ovládáno pomocí procedur a funkcí v pěti základních balících. První balík je *RS_INTER* (9), který zpravidla poskytuje informace a nastavení úložiště. Tento balík je vnitřně využíván všemi ostatními balíky. Druhý balík je *RS_ATTR* (9), pomocí nějž lze definovat a modifikovat atributy, které lze následně přiřazovat jednotlivým instancím objektů. Pojmenované třídy a práci s nimi zajišťují funkce a procedury z balíku *RS_CLSS* (9). Pro manipulaci s konkrétními instancemi objektů je určen balík *RS_OBJ* (9). Indexy se ovládají procedurami z balíku *RS_INX* (9).

Ukázka použití balíků *RS_ATTR* (9) a *RS_OBJ* (9) je v příkladu kódu (Kód 1). Bylo zvoleno ukázání použití na příkladech, které jsou i se slovním doprovodem přímo v textu, neboť se domnívám, že pochopení pouze z ukázek kódu bez slovního doprovodu by bylo značně netriviální. Kódech (Kód 2) a (Kód 3) je ukázáno použití balíku *RS_CLSS* (9), které je taktéž doprovázeno slovním komentářem. Práce s balíkem *RS_INX* (9) je v ukázkových kódech (Kód 7), (Kód 8) a (Kód 9).

Instalace úložiště je popsána v souboru **readme.txt** ve složce **bin** na doprovodném cd.

Použité zdrojové kódy jsou uložené na cd ve složce **bin\samples**.

A.1 Balík RS_INTER

Jedná se o základní balík, jehož funkce využívají všechny ostatní balíky v úložišti. Obsahuje soubor funkcí, určených především na zjišťování jmen systémových objektů úložiště, zjištění id podle jména nebo zjištění jména podle id nějakého systémového objektu, či ověření korektního jména pro nový objekt.

A.1.1 TAB_OBJECT_ATTRIBUTE

```
FUNCTION TAB_OBJECT_ATTRIBUTE RETURN VARCHAR2;
```

Vrací jméno vztahové tabulky katalogu mezi instancemi objektů a atributy.

A.1.2 TAB_INDEX_ATTRIBUTE

```
FUNCTION TAB_INDEX_ATTRIBUTE RETURN VARCHAR2;
```

Vrací jméno vztahové tabulky katalogu mezi indexy a atributy.

A.1.3 TAB_CLASS_ATTRIBUTE

```
FUNCTION TAB_CLASS_ATTRIBUTE RETURN VARCHAR2;
```

Vrací jméno vztahové tabulky katalogu mezi pojmenovanými třídami a atributy.

A.1.4 TAB_ATTRIBUTES

```
FUNCTION TAB_ATTRIBUTES RETURN VARCHAR2;
```

Vrací jméno tabulky katalogu pro definované atributy.

A.1.5 TAB_OBJECTS

```
FUNCTION TAB_OBJECTS RETURN VARCHAR2;
```

Vrací jméno tabulky katalogu pro instance objektů.

A.1.6 TAB_CLASSES

```
FUNCTION TAB_CLASSES RETURN VARCHAR2;
```

Vrací jméno tabulky katalogu pro pojmenované třídy.

A.1.7 TAB_INDEXES

```
FUNCTION TAB_INDEXES RETURN VARCHAR2;
```

Vrací jméno tabulky katalogu pro indexy.

A.1.8 SEQ_ATTRIBUTES

```
FUNCTION SEQ_ATTRIBUTES RETURN VARCHAR2;
```

Vrací jméno posloupnosti pro tabulku katalogu s atributy.

A.1.9 SEQ_OBJECTS

```
FUNCTION SEQ_OBJECTS RETURN VARCHAR2;
```

Vrací jméno posloupnosti pro tabulku katalogu s objekty.

A.1.10 SEQ_CLASSES

```
FUNCTION SEQ_CLASSES RETURN VARCHAR2;
```

Vrací jméno posloupnosti pro tabulku katalogu s pojmenovanými třídami.

A.1.11 SEQ_INDEXES

```
FUNCTION SEQ_INDEXES RETURN VARCHAR2;
```

Vrací jméno posloupnosti pro tabulku katalogu s indexy.

A.1.12 TYP_AGE

```
FUNCTION TYP_AGE RETURN VARCHAR2;
```

Vrací jméno typu pro hodnotu s se stářím objektů.

A.1.13 TYP_AGE_TAB

```
FUNCTION TYP_AGE_TAB RETURN VARCHAR2;
```

Vrací jméno typu tabulky se stářím objektů.

A.1.14 SUF_VALID_CHECK

```
FUNCTION SUF_VALID_CHECK RETURN VARCHAR2;
```

Vrací sufix pro omezení validního intervalu.

A.1.15 SUF_INDEX

```
FUNCTION SUF_INDEX RETURN VARCHAR2;
```

Vrací sufix pro jména indexů.

A.1.16 SUF_INDEX_TABLE

```
FUNCTION SUF_INDEX_TABLE RETURN VARCHAR2;
```

Vrací sufix pro jméno tabulky s návratovou množinou id vlastního indexu.

A.1.17 SUF_OBJECT_INDEX

```
FUNCTION SUF_OBJECT_INDEX RETURN VARCHAR2;
```

Vrací sufix pro jména indexů u cizích klíčů na objekty.

A.1.18 SUF_VALUE_INDEX

```
FUNCTION SUF_VALUE_INDEX RETURN VARCHAR2;
```

Vrací sufix pro indexy hodnot.

A.1.19 SUF_UNIQUE

```
FUNCTION SUF_UNIQUE RETURN VARCHAR2;
```

Vrací sufix pro omezení *unique*.

A.1.20 ATTR_TEXT

```
FUNCTION ATTR_TEXT RETURN VARCHAR2;
```

Funkce vrací standardní typ pro *obecné atributy*.

A.1.21 ATTR_REF

```
FUNCTION ATTR_REF RETURN VARCHAR2;
```

Funkce vrací typ pro *referenční atributy*.

A.1.22 INX_TYPE_ORA

```
FUNCTION INX_TYPE_ORA RETURN NUMBER;
```

Vrací konstantu pro typ *interního indexu* Oracle.

A.1.23 INX_TYPE_RO

```
  FUNCTION INX_TYPE_RO RETURN NUMBER;
```

Vrací konstantu pro typ *externího indexu*.

A.1.24 INX_CREATE_REINDEX_ALL

```
FUNCTION INX_CREATE_REINDEX_ALL RETURN NUMBER;
```

Vrací konstantu indikující, že nově vytvořený index má obsahovat i všechna data již vyhovujících objektů.

A.1.25 INX_CREATE_NO_REINDEX_ALL

```
  FUNCTION INX_CREATE_NO_REINDEX_ALL RETURN NUMBER;
```

Vrací konstantu indikující, že v indexu budou indexované pouze informace modifikované až po vytvoření skutečného indexu.

A.1.26 INX_MOD_CLEAR

```
FUNCTION INX_MOD_CLEAR RETURN VARCHAR2;
```

Funkce vrací přepínač pro vymazání tabulky výsledků z předchozího hledání při dotazech hledání s použitím vlastního indexu.

A.1.27 INX_MOD_APPEND

```
FUNCTION INX_MOD_APPEND RETURN VARCHAR2;
```

Funkce vrací přepínač pro sjednocení výsledků s předchozího hledání při dotazech.

A.1.28 MIN_TIMESTAMP

```
FUNCTION MIN_TIMESTAMP RETURN TIMESTAMP;
```

Funkce vrací konstantu času reprezentující mínus nekonečno Aktuálně 1.1.1900.

A.1.29 MAX_TIMESTAMP

```
FUNCTION MAX_TIMESTAMP RETURN TIMESTAMP;
```

Funkce vrací konstantu času reprezentující plus nekonečno. Aktuálně 1.1.2100.

A.1.30 CHECK_VALID_NAME

```
PROCEDURE CHECK_VALID_NAME(STR_NAME IN VARCHAR2);
```

Ověří, zda jméno pro nový objekt nekoliduje s nějakým systémovým jménem úložiště, jako je atribut, index a zda je korektní. Korektní jméno musí odpovídat regulárnímu výrazu *„^[A-Z][A-Z0-9_]{1,30}$“*. Pokud jméno nesplňuje podmínky a nebo je již obsazené, vyhazují se výjimky *INVALID_NAME_CHAR*, *INVALID_NAME_LEN* a *RESERVED_NAME*.

A.1.31 CHECK_VALID_TIME

```
PROCEDURE CHECK_VALID_TIME (VALID_FROM IN TIMESTAMP, VALID_TO IN
TIMESTAMP);
```

Ověří validitu pro krajní hodnoty časového intervalu. V případě nevalidního intervalu vyhazuje výjimku *INVALID_VALID_INTERVAL*.

A.1.32 CHECK_ATTR

```
FUNCTION CHECK_ATTR (ATTR_NAME IN VARCHAR2) RETURN NUMBER;
FUNCTION CHECK_ATTR (ATT_ID IN NUMBER) RETURN VARCHAR2;
```

Zjistí id či jméno atributu. Pokud atribut s daným jménem či id v úložišti neexistuje, vyhazuje výjimku *ATTR_NOT_FOUND*.

A.1.33 CHECK_CLSS

```
FUNCTION CHECK_CLSS (CLSS_NAME IN VARCHAR2) RETURN NUMBER;
FUNCTION CHECK_CLSS (CLSS_ID IN NUMBER) RETURN VARCHAR2;
```

Zjistí id či jméno pojmenované třídy. Pokud třída s daným jménem či id v úložišti neexistuje vyhazuje výjimku *CLSS_NOT_FOUND*.

A.1.34 CHECK_OBJ

```
PROCEDURE CHECK_OBJ (OBJ_ID IN NUMBER);
```

Ověří, zda objekt s daným id v úložišti existuje. Pokud objekt neexistuje vyhazuje výjimku *OBJ_NOT_FOUND*.

A.1.35 CHECK_INX

```
FUNCTION CHECK_INX (INX_NAME IN VARCHAR2) RETURN NUMBER;
FUNCTION CHECK_INX (INX_ID IN NUMBER) RETURN VARCHAR2;
```

Zjistí id či jméno uživatelem vytvořeného indexu. Pokud index s daným jménem či id v úložišti neexistuje vyhazuje výjimku *INX_NOT_FOUND*.

A.1.36 CHECK_LEGAL_INX

```
FUNCTION CHECK_LEGAL_INX (INX_NAME VARCHAR2) RETURN NUMBER;
FUNCTION CHECK_LEGAL_INX (INX_ID IN NUMBER) RETURN VARCHAR2;
```

Vrací číslo 1, pokud je index daným jménem či id je skutečný, tedy vytvořený a plně funkční, jinak vrací 0. Pokud index s daným jménem či id v úložišti neexistuje vyhazuje výjimku *INX_NOT_FOUND*.

A.2 Balík RS_ATTR

Tento balík je určen pro práci s obecnými atributy nebo referenčními atributy. Především se jedná o vytváření či rušení atributů.

A.2.1 CREATE_ATTR

```
PROCEDURE CREATE_ATTR (ATTR_NAME IN VARCHAR2, ATTR_TYPE IN VARCHAR2);
```

Procedura interně volá funkci *CREATE_ATTR_RET_ID*.

A.2.2 CREATE_ATTR_RET_ID

```
FUNCTION CREATE_ATTR_RET_ID (ATTR_NAME IN VARCHAR2, ATTR_TYPE IN VARCHAR2)
RETURN NUMBER;
```

Vytvoří v úložišti atribut s daným jménem. Hodnota *REFERENCE* proměnné *ATTR_TYPE* určuje, že se má jednat o *referenční atribut*, jinak je vytvořen *obecný atribut*. Pokud v úložišti již existuje atribut s daným jménem vyhazuje výjimku *ATTR_EXISTS*. Funkce vrací id posledně vytvořeného atributu.

A.2.3 LAST_CREATED_ATTR

```
FUNCTION LAST_CREATED_ATTR RETURN NUMBER;
```

Vrací id atributu, který byl v daném session vytvořený jako poslední. Pokud takový atribut není, vrací hodnotu 0.

A.2.4 RECREATE_ATTR

```
PROCEDURE RECREATE_ATTR (ATTR_NAME IN VARCHAR2, ATTR_TYPE IN VARCHAR2);
```

Interně zavolá *RECREATE_ATTR_RET_ID*.

A.2.5 RECREATE_ATTR_RET_ID

```
FUNCTION RECREATE_ATTR_RET_ID (ATTR_NAME IN VARCHAR2, ATTR_TYPE IN
VARCHAR2) RETURN NUMBER;
```

Pokud atribut s daným jménem v úložišti neexistuje, je nově vytvořený, jinak se nestane nic. Hodnota *REFERENCE* proměnné *ATTR_TYPE* určuje, že se má jednat o referenční atribut, jinak je vytvořen obecný atribut. Pokud atribut s daným jménem již existuje a hodnota *ATTR_TYPE* nového a starého atributu není shodná, je vyhozena výjimka *ATTR_DIFF_REF*.

A.2.6 DROP_ATTR

```
PROCEDURE DROP_ATTR (ATTR_NAME IN VARCHAR2);
PROCEDURE DROP_ATTR (ATT_ID IN NUMBER);
```

Zruší atribut s daným jménem či id. Pokud v úložišti existuje instance objektu, které má tento atribut definovaný vyhazuje výjimku *ATTR_NOT_EMTPY*. Pokud je nad atributem definovaný uživatelský index, procedura selže.

A.2.7 DROP_ATTR_FORCE

```
PROCEDURE DROP_ATTR_FORCE (ATTR_NAME IN VARCHAR2);
PROCEDURE DROP_ATTR_FORCE (ATT_ID IN NUMBER);
```

Zruší atribut s daným jménem či id i v případě, že v úložišti existuje instance objektu, která má daný atribut definovaný. Pokud je nad atributem definovaný uživatelský index, procedura selže.

A.3 Balík RS_CLSS

Tento balík obsahuje procedury pro vytváření, modifikaci a mazání pojmenovaných tříd. V balíku jsou obsažené další rutiny pro práci s maskami pojmenovaných tříd či zjištění hierarchie tříd.

A.3.1 CREATE_CLSS

```
PROCEDURE CREATE_CLSS (CLSS_NAME IN VARCHAR2);
```

Interně volá *CREATE_CLSS_RET_ID*.

A.3.2 CREATE_CLSS_RET_ID

```
FUNCTION CREATE_CLSS_RET_ID (CLSS_NAME IN VARCHAR2) RETURN NUMBER;
```

Funkce vytvoří novou pojmenovanou třídu, jejíž id vrací. Pokud třída s daným jménem již existuje, vyhazuje výjimku *CLSS_EXISTS*.

A.3.3 LAST_CREATED_CLSS

```
FUNCTION LAST_CREATED_CLSS RETURN NUMBER;
```

Vrací id naposledy vytvořené pojmenované třídy ve stávající session. Pokud v aktuální session žádná třída vytvořena nebyla, vrací hodnotu 0.

A.3.4 DROP_CLSS

```
PROCEDURE DROP_CLSS (CLSS_NAME IN VARCHAR2);
PROCEDURE DROP_CLSS_BY_ID (CLSS_ID IN NUMBER);
```

Zruší pojmenovanou třídu s daným jménem či id.

A.3.5 COPY

```
PROCEDURE COPY (CLSS_NAME IN VARCHAR2, NEW_CLSS_NAME IN VARCHAR2);
PROCEDURE COPY_BY_ID (CLSS_ID IN NUMBER, NEW_CLSS_NAME IN VARCHAR2);
```

Procedura interně volá funkci *COPY_RET_ID*.

A.3.6 COPY_RET_ID

```
FUNCTION COPY_RET_ID (CLSS_NAME IN VARCHAR2, NEW_CLSS_NAME IN VARCHAR2)
RETURN NUMBER;
FUNCTION COPY_RET_ID_BY_ID (CLSS_ID IN NUMBER, NEW_CLSS_NAME IN VARCHAR2)
RETURN NUMBER;
```

Vytvoří novou pojmenovanou třídu se jménem *NEW_CLSS_NAME*. Nová třída bude definovaná stejnou množinou atributů jako třída *CLSS_NAME* či *CLSS_ID*. Vrací id nově vytvořené třídy.

A.3.7 ADD_ATTR

```
PROCEDURE ADD_ATTR (CLSS_NAME IN VARCHAR2, ATTR_NAME IN VARCHAR2);
PROCEDURE ADD_ATTR (CLSS_NAME IN VARCHAR2, ATT_ID IN NUMBER);
PROCEDURE ADD_ATTR (CLSS_ID IN NUMBER, ATTR_NAME IN VARCHAR2);
PROCEDURE ADD_ATTR (CLSS_ID IN NUMBER, ATT_ID IN NUMBER);
```

Přidá do množiny atributů pojmenované třídy daný atribut. Pokud je atribut již v množině definující třídu obsažen, vyhazuje výjimku *ATTR_HAS_ALREADY_ADDED*.

A.3.8 DEL_ATTR

```
PROCEDURE DEL_ATTR (CLSS_NAME IN VARCHAR2, ATTR_NAME IN VARCHAR2);
PROCEDURE DEL_ATTR (CLSS_NAME IN VARCHAR2, ATT_ID IN NUMBER);
PROCEDURE DEL_ATTR (CLSS_ID IN NUMBER, ATTR_NAME IN VARCHAR2);
PROCEDURE DEL_ATTR (CLSS_ID IN NUMBER, ATT_ID IN NUMBER);
```

Odstraní z množiny atributů definující pojmenovanou třídu atribut.

A.3.9 MIN_ATTRS_CLSS

```
FUNCTION MIN_ATTRS_CLSS(CLSS_NAME1 IN VARCHAR2, CLSS_NAME2 IN VARCHAR2)
RETURN VARCHAR2;

FUNCTION MIN_ATTRS_CLSS(CLSS_NAME1 IN VARCHAR2, CLSS_ID2 IN NUMBER) RETURN
VARCHAR2;

FUNCTION MIN_ATTRS_CLSS(CLSS_ID1 IN NUMBER, CLSS_NAME2 IN VARCHAR2) RETURN
VARCHAR2;

FUNCTION MIN_ATTRS_CLSS(CLSS_ID1 IN NUMBER, CLSS_ID2 IN NUMBER) RETURN
VARCHAR2;
```

Vrací množinu atributů definující nejmenšího společného předka pojmenovaných tříd. Vstupem jsou dvě existující pojmenované třídy.

A.3.10 MIN_MASK_CLSS

```
FUNCTION MIN_MASK_CLSS(CLSS_NAME1 IN VARCHAR2, CLSS_NAME2 IN VARCHAR2)
RETURN VARCHAR2;

FUNCTION MIN_MASK_CLSS(CLSS_NAME1 IN VARCHAR2, CLSS_ID2 IN NUMBER) RETURN
VARCHAR2;

FUNCTION MIN_MASK_CLSS(CLSS_ID1 IN NUMBER, CLSS_NAME2 IN VARCHAR2) RETURN
VARCHAR2;

FUNCTION MIN_MASK_CLSS(CLSS_ID1 IN NUMBER, CLSS_ID2 IN NUMBER) RETURN
VARCHAR2;
```

Vrací masku nejmenšího společného předka pojmenovaných tříd. Vstupem jsou dvě existující pojmenované třídy.

A.3.11 MAX_ATTRS_CLSS

```
FUNCTION MAX_ATTRS_CLSS(CLSS_NAME1 IN VARCHAR2, CLSS_NAME2 IN VARCHAR2)
RETURN VARCHAR2;

FUNCTION MAX_ATTRS_CLSS(CLSS_NAME1 IN VARCHAR2, CLSS_ID2 IN NUMBER) RETURN
VARCHAR2;

FUNCTION MAX_ATTRS_CLSS(CLSS_ID1 IN NUMBER, CLSS_NAME2 IN VARCHAR2) RETURN
VARCHAR2;

FUNCTION MAX_ATTRS_CLSS(CLSS_ID1 IN NUMBER, CLSS_ID2 IN NUMBER) RETURN
VARCHAR2;
```

Vrací množinu atributů definující nejmenšího společného potomka pojmenovaných tříd. Vstupem jsou dvě existující pojmenované třídy.

A.3.12 MAX_MASK_CLSS

```
FUNCTION MAX_MASK_CLSS(CLSS_NAME1 IN VARCHAR2, CLSS_NAME2 IN VARCHAR2)
RETURN VARCHAR2;

FUNCTION MAX_MASK_CLSS(CLSS_NAME1 IN VARCHAR2, CLSS_ID2 IN NUMBER) RETURN
VARCHAR2;
```

```
FUNCTION MAX_MASK_CLSS(CLSS_ID1 IN NUMBER, CLSS_NAME2 IN VARCHAR2) RETURN
VARCHAR2;

FUNCTION MAX_MASK_CLSS(CLSS_ID1 IN NUMBER, CLSS_ID2 IN NUMBER) RETURN
VARCHAR2;
```

Vrací masku nejmenšího společného potomka pojmenovaných tříd. Vstupem jsou dvě existující pojmenované třídy.

A.3.13 CLSS_TO_ATTRS

```
FUNCTION CLSS_TO_ATTRS(CLSS_NAME IN VARCHAR2) RETURN VARCHAR2;
```

Vrací množinu atributů definující již existující pojmenovanou třídu se jménem *CLSS_NAME*.

A.3.14 CLSS_TO_MASK

```
FUNCTION CLSS_TO_MASK(CLSS_NAME IN VARCHAR2) RETURN VARCHAR2;
```

Vrací masku existující pojmenované třídy se jménem *CLSS_NAME*.

A.3.15 ATTRS_TO_CLSS

```
FUNCTION ATTRS_TO_CLSS(ATTRS_STR IN VARCHAR2) RETURN VARCHAR2;
```

Vrací jméno první pojmenované třídy definované vstupní množinou atributů.

A.3.16 ATTRS_TO_MASK

```
FUNCTION ATTRS_TO_MASK(ATTRS IN VARCHAR2) RETURN VARCHAR2;
```

Vrací masku ke vstupní množině atributů.

A.3.17 MASK_TO_ATTRS

```
FUNCTION MASK_TO_ATTRS(MASK IN VARCHAR2) RETURN VARCHAR2;
```

Vrací množinu atributů vstupní masky.

A.3.18 MASK_TO_CLSS

```
FUNCTION MASK_TO_CLSS(MASK_STR IN VARCHAR2) RETURN VARCHAR2;
```

Vrací jméno první vyhovující pojmenované třídy vstupní masce.

A.3.19 HIERARCHY_MASK

```
FUNCTION HIERARCHY_MASK(CLSS_MASK1 IN VARCHAR2, CLSS_MASK2 IN VARCHAR2)
RETURN NUMBER;
```

Vrací vzájemný vztah dvou vstupních masek tříd dle hierarchie dědění. Hodnota -2 znamená, že masky nejsou v žádném vztahu. Hodnota 1 indikuje, že maska *CLSS_MASK1* je

rodičem masky *CLSS_MASK2*. Hodnota 0 znamená, že maska *CLSS_MASK1* je rovna masce *CLSS_MASK2*. Hodnota -1 znamená, že maska *CLSS_MASK1* je potokem masky *CLSS_MASK2*.

A.3.20 PARENT_MASK

```
FUNCTION PARENT_MASK(CLSS_MASK1 IN VARCHAR2, CLSS_MASK2 IN VARCHAR2)
RETURN NUMBER;
```

Vrací hodnotu 1 pokud je maska *CLSS_MASK1* rodičem masky *CLSS_MASK2*, jinak vrací číslo 0.

A.3.21 SAME_MASK

```
FUNCTION SAME_MASK(CLSS_MASK1 IN VARCHAR2, CLSS_MASK2 IN VARCHAR2) RETURN
NUMBER;
```

Vrací hodnotu 1 pokud je maska *CLSS_MASK1* ekvivalentní masce *CLSS_MASK2*, jinak vrací číslo 0.

A.3.22 CHILD_MASK

```
FUNCTION CHILD_MASK(CLSS_MASK1 IN VARCHAR2, CLSS_MASK2 IN VARCHAR2) RETURN
NUMBER;
```

Vrací hodnotu 1, pokud je maska *CLSS_MASK1* potomkem masky *CLSS_MASK2*, jinak vrací číslo 0.

A.3.23 CHECK_SOME_REDUNDANT_ATTRS

```
FUNCTION CHECK_SOME_REDUNDANT_ATTRS RETURN VARCHAR2;
```

Pokud v úložišti existuje více pojmenovaných tříd definujících stejnou množinou atributů, potom tato funkce vrací jméno libovolné pojmenované třídy, která je v konfliktu s nějakou jinou pojmenovanou třídou.

A.4 Balík RS_OBJ

Balík pro kompletní zprávu objektů jako je vytváření, modifikace či rušení instancí objektů. Pojem modifikace znamená, že k objektu lze přidávat atributy s hodnotami či jednotlivé atributy u instancí objektů mazat. Dále lze využívat pojmenovaných tříd, podle nichž lze přistupovat k jednotlivým instancím objektů. V tomto balíku je pod procedurami a funkcemi skryta největší část logiky úložiště.

A.4.1 CREATE_OBJ

```
PROCEDURE CREATE_OBJ(OBJ_IDENTIFICATOR IN VARCHAR2);
```

Procedura interně volá funkci *CREATE_OBJ_RET_ID*.

A.4.2 CREATE_OBJ_RET_ID

```
FUNCTION CREATE_OBJ_RET_ID(OBJ_IDENTIFICATOR IN VARCHAR2) RETURN NUMBER;
```

Funkce vytvoří novou instanci objektu v úložišti a vrátí id nově vytvořeného objektu. Parametr je jednoznačný identifikátor objektu v úložišti v podobě textového popisku. Pokud popisek není vyplněný, bude se identifikátor automaticky nastavovat na hodnotu id objektu.

A.4.3 COPY_SNAPSHOT_AT_TIME

```
PROCEDURE COPY_SNAPSHOT_AT_TIME(OBJ_IDENTIFICATOR IN VARCHAR2, VALID_TIME
IN TIMESTAMP, NEW_VALID_FROM IN TIMESTAMP, NEW_VALID_TO IN TIMESTAMP);

PROCEDURE COPY_SNAPSHOT_AT_TIME(OBJ_ID IN NUMBER, VALID_TIME IN TIMESTAMP,
NEW_VALID_FROM IN TIMESTAMP, NEW_VALID_TO IN TIMESTAMP);
```

Procedura interně volá funkci *COPY_SNAPSHOT_AT_TIME_RET_ID*.

A.4.4 COPY_SNAPSHOT_AT_TIME_RET_ID

```
FUNCTION COPY_SNAPSHOT_AT_TIME_RET_ID(OBJ_IDENTIFICATOR IN VARCHAR2,
VALID_TIME IN TIMESTAMP, NEW_VALID_FROM IN TIMESTAMP, NEW_VALID_TO IN
TIMESTAMP) RETURN NUMBER;

FUNCTION COPY_SNAPSHOT_AT_TIME_RET_ID(OBJ_ID IN NUMBER, VALID_TIME IN
TIMESTAMP, NEW_VALID_FROM IN TIMESTAMP, NEW_VALID_TO IN TIMESTAMP) RETURN
NUMBER
```

Funkce vytvoří novou instanci objektu v úložišti a vrátí id nově vytvořeného objektu. Nový objekt bude mít definované všechny atributy, které měl objekt *OBJ_IDENTIFIKATOR* či *OBJ_ID* v časový okamžik *VALID_TIME*. Platnost všech atributů nového objektu bude v intervalu mezi *NEW_VALID_FROM* a *NEW_VALID_TO*.

A.4.5 COPY_MOVE

```
PROCEDURE COPY_MOVE(OBJ_IDENTIFICATOR IN VARCHAR2, TIME_MOVE IN NUMBER);

PROCEDURE COPY_MOVE(OBJ_ID IN NUMBER, TIME_MOVE IN NUMBER);
```

Procedura interně volá funkci *COPY_MOVE_RET_ID*.

A.4.6 COPY_MOVE_RET_ID

```
FUNCTION COPY_MOVE_RET_ID(OBJ_IDENTIFICATOR IN VARCHAR2, TIME_MOVE IN
NUMBER) RETURN NUMBER;

FUNCTION COPY_MOVE_RET_ID(OBJ_ID IN NUMBER, TIME_MOVE IN NUMBER) RETURN
NUMBER;
```

Funkce vytvoří novou instanci objektu v úložišti a vrátí id nově vytvořeného objektu. Nový objekt bude klonem objektu *OBJ_IDENTIFICATOR* či *OBJ_ID* se všemi atributy definovanými po celou dobu života objektu *OBJ_IDENTIFICATOR* či *OBJ_ID*. Parametr *TIME_MOVE* umožňuje vytvořit kopii posunutou v čase. Posunutí je uvedeno ve dnech.

A.4.7 COPY

```
PROCEDURE COPY(OBJ_IDENTIFICATOR IN VARCHAR2);
PROCEDURE COPY(OBJ_ID IN NUMBER);
```

Procedura interně volá funkci *COPY_RET_ID*.

A.4.8 COPY_RET_ID

```
FUNCTION COPY_RET_ID(OBJ_IDENTIFICATOR IN VARCHAR2) RETURN NUMBER;
FUNCTION COPY_RET_ID(OBJ_ID IN NUMBER) RETURN NUMBER;
```

Ekvivalentní funkce funkci *COPY_MOVE_RET_ID* bez posunu nově vytvořeného objektu v čase.

A.4.9 LAST_CREATED_OBJECT

```
FUNCTION LAST_CREATED_OBJECT RETURN NUMBER;
```

Funkce vrací id naposledy vytvořené instance objektu v rámci dané session. Pokud nebyla doposud vytvořena žádná instance objektu, vyhazuje výjimku *OBJ_NOT_FOUND*.

A.4.10 DROP_OBJ

```
PROCEDURE DROP_OBJ(OBJ_IDENTIFICATOR IN VARCHAR2);
PROCEDURE DROP_OBJ(OBJ_ID IN NUMBER);
```

Smaže instanci objektu, která nemá definované žádné atributy. Při pokusu o smazání zamknutého objektu vyhazuje výjimku *OBJ_LOCKED*. Pokud má atribut definovaný alespoň jeden atribut vyhazuje výjimku *OBJECT_ISNT_EMPTY*.

A.4.11 DROP_OBJ_FORCE

```
PROCEDURE DROP_OBJ_FORCE(OBJ_IDENTIFICATOR IN VARCHAR2);
PROCEDURE DROP_OBJ_FORCE(OBJ_ID IN NUMBER);
```

Smaže instanci objektu i se všemi definovanými atributy.

A.4.12 LOCK_OBJ

```
PROCEDURE LOCK_OBJ(OBJ_IDENTIFICATOR IN VARCHAR2);
PROCEDURE LOCK_OBJ(OBJ_ID IN NUMBER);
```

Procedura zamkne instanci objektu.

A.4.13 UNLOCK_OBJ

```
PROCEDURE UNLOCK_OBJ;
```

Procedura odemkne zamčenou instanci objektu v aktuální session a pro odemčenou instanci objektu přepočítá hodnoty modifikovaných atributů.

A.4.14 LOCK_ID

```
FUNCTION LOCK_ID RETURN NUMBER;
```

Funkce vrací id uzamčené instance objektu v aktuální session. Pokud takový objekt neexistuje vrací 0.

A.4.15 SET_VALUE_L

```
PROCEDURE SET_VALUE_L (ATTR_NAME IN VARCHAR2, VAL IN VARCHAR2, VALID_FROM
IN TIMESTAMP, VALID_TO IN TIMESTAMP);

PROCEDURE SET_VALUE_L (ATT_ID IN NUMBER, VAL IN VARCHAR2, VALID_FROM IN
TIMESTAMP, VALID_TO IN TIMESTAMP);
```

Procedury definují na uzamčeném objektu atribut *ATTR_NAME* či *ATT_ID* s hodnotou *VAL* a platností od *VALID_FROM* do *VALID_TO*. Pokud objekt již má definovaný atribut přepíše starou hodnotu novou.

A.4.16 DEL_VALUE_L

```
PROCEDURE DEL_VALUE_L (ATTR_NAME IN VARCHAR2, VALID_FROM IN TIMESTAMP,
VALID_TO IN TIMESTAMP);

PROCEDURE DEL_VALUE_L (ATT_ID IN NUMBER, VALID_FROM IN TIMESTAMP,
VALID_TO IN TIMESTAMP);
```

Procedura smaže na uzamčeném objektu atribut *ATTR_NAME* či *ATT_ID* v rozmezí od *VALID_FROM* do *VALID_TO*.

A.4.17 GET_VALUE_L

```
FUNCTION GET_VALUE_L(ATTR_NAME IN VARCHAR2, VALID_TIME IN TIMESTAMP)
RETURN VARCHAR2;

FUNCTION GET_VALUE_L(ATT_ID IN NUMBER, VALID_TIME IN TIMESTAMP) RETURN
VARCHAR2;
```

Funkce vrátí hodnotu zobrazení $val(o, a_i)^t$ právě uzamčené instance objektu. V případech, kdy v session není žádný uzamčený objekt nebo atribut neexistuje a nebo je atribut na instanci v daný časový okamžik nedefinovaný, vrací hodnotu *null*.

A.4.18 SET_VALUE

```
PROCEDURE SET_VALUE (OBJ_IDENTIFICATOR IN VARCHAR2, ATTR_NAME IN VARCHAR2,
VAL IN VARCHAR2, VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP);

PROCEDURE SET_VALUE (OBJ_ID IN NUMBER, ATTR_NAME IN VARCHAR2, VAL IN
VARCHAR2, VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP);

PROCEDURE SET_VALUE (OBJ_IDENTIFICATOR IN VARCHAR2, ATT_ID IN NUMBER, VAL
IN VARCHAR2, VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP);

PROCEDURE SET_VALUE (OBJ_ID IN NUMBER, ATT_ID IN NUMBER, VAL IN VARCHAR2,
VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP);
```

Na objektu definuje atribut s hodnotou *VAL* a platností od *VALID_FROM* do *VALID_TO*. Pokud objekt již má definovaný atribut, přepíše starou hodnotu novou. Interně objekt zamkne upraví a poté zase odemkne.

A.4.19 DEL_VALUE

```
PROCEDURE DEL_VALUE (OBJ_IDENTIFICATOR IN VARCHAR2, ATTR_NAME IN VARCHAR2,
VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP);

PROCEDURE DEL_VALUE (OBJ_ID IN NUMBER, ATTR_NAME IN VARCHAR2, VALID_FROM
IN TIMESTAMP, VALID_TO IN TIMESTAMP);

PROCEDURE DEL_VALUE (OBJ_IDENTIFICATOR IN VARCHAR2, ATT_ID IN NUMBER,
VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP);

PROCEDURE DEL_VALUE (OBJ_ID IN NUMBER, ATT_ID IN NUMBER, VALID_FROM IN
TIMESTAMP, VALID_TO IN TIMESTAMP);
```

Objektu smaže atribut v rozmezí od *VALID_FROM* do *VALID_TO*. Interně objekt zamkne, upraví a poté zase odemkne.

A.4.20 GET_VALUE

```
FUNCTION GET_VALUE(OBJ_IDENTIFICATOR IN VARCHAR2, ATTR_NAME IN VARCHAR2,
VALID_TIME IN TIMESTAMP) RETURN VARCHAR2;

FUNCTION GET_VALUE(OBJ_ID IN NUMBER, ATTR_NAME IN VARCHAR2, VALID_TIME IN
TIMESTAMP) RETURN VARCHAR2;

FUNCTION GET_VALUE(OBJ_IDENTIFICATOR IN VARCHAR2, ATT_ID IN NUMBER,
VALID_TIME IN TIMESTAMP) RETURN VARCHAR2;

FUNCTION GET_VALUE(OBJ_ID IN NUMBER, ATT_ID IN NUMBER, VALID_TIME IN
TIMESTAMP) RETURN VARCHAR2;
```

Funkce vrátí hodnotu zobrazení $val(o, a_i)^t$ instance objektu. V případech, kdy atribut neexistuje a nebo je atribut na instanci v daný časový okamžik nedefinovaný, vrací hodnotu *null*.

A.4.21 TRIM_OBJ

```
PROCEDURE TRIM_OBJ(OBJ_IDENTIFICATOR IN VARCHAR2, VALID_FROM IN TIMESTAMP,
VALID_TO IN TIMESTAMP);

PROCEDURE TRIM_OBJ(OBJ_ID IN NUMBER, VALID_FROM IN TIMESTAMP, VALID_TO IN
```

```
TIMESTAMP);
```

Procedura smaže u objektu všechny atributy mimo rozsah *VALID_FROM* a *VALID_TO*. Pokud je zadaný interval chybný, vyhazuje výjimku *INVALID_TIME_INTERVAL*.

A.4.22 BORN_TIME

```
FUNCTION BORN_TIME(OBJ_IDENTIFICATOR IN VARCHAR2) RETURN TIMESTAMP;
FUNCTION BORN_TIME(OBJ_ID IN NUMBER) RETURN TIMESTAMP;
```

Funkce vrací čas narození instance objektu. Pokud objekt nemá definovanými žádný atribut, vrací *null*.

A.4.23 DEATH_TIME

```
FUNCTION DEATH_TIME(OBJ_IDENTIFICATOR IN VARCHAR2) RETURN TIMESTAMP;
FUNCTION DEATH_TIME(OBJ_ID IN NUMBER) RETURN TIMESTAMP;
```

Funkce vrací čas úmrtí instance objektu. Pokud objekt nemá definovanými žádný atribut, vrací *null*.

A.4.24 ATTRS_MAX

```
FUNCTION ATTRS_MAX(OBJ_IDENTIFICATOR IN VARCHAR2) RETURN VARCHAR2;
FUNCTION ATTRS_MAX(OBJ_ID IN NUMBER) RETURN VARCHAR2;
```

Funkce vrací $\overline{def}(o)$ pro instanci objektu.

A.4.25 ATTRS_MIN

```
FUNCTION ATTRS_MIN(OBJ_IDENTIFICATOR IN VARCHAR2) RETURN VARCHAR2;
FUNCTION ATTRS_MIN(OBJ_ID IN NUMBER) RETURN VARCHAR2;
```

Funkce vrací $\underline{def}(o)$ pro instanci objektu.

A.4.26 ATTRS_AT_TIME

```
FUNCTION ATTRS_AT_TIME(OBJ_IDENTIFICATOR IN VARCHAR2, VALID_TIME IN
TIMESTAMP) RETURN VARCHAR2;
FUNCTION ATTRS_AT_TIME(OBJ_ID IN NUMBER, VALID_TIME IN TIMESTAMP) RETURN
VARCHAR2;
```

Funkce vrací $def^{(t)}(o)$ pro instanci objektu.

A.4.27 MASK_MAX

```
FUNCTION MASK_MAX(OBJ_IDENTIFICATOR IN VARCHAR2) RETURN VARCHAR2;
FUNCTION MASK_MAX(OBJ_ID IN NUMBER) RETURN VARCHAR2;
```

Funkce vrací masku $\overline{def}(o)$ pro instanci objektu.

A.4.28 MASK_MIN

```
FUNCTION MASK_MIN(OBJ_IDENTIFICATOR IN VARCHAR2) RETURN VARCHAR2;
FUNCTION MASK_MIN(OBJ_ID IN NUMBER) RETURN VARCHAR2;
```

Funkce vrací masku $\underline{def}(o)$ pro instanci objektu.

A.4.29 MASK_AT_TIME

```
FUNCTION MASK_AT_TIME(OBJ_IDENTIFICATOR IN VARCHAR2, VALID_TIME IN
TIMESTAMP) RETURN VARCHAR2;
FUNCTION MASK_AT_TIME(OBJ_ID IN NUMBER, VALID_TIME IN TIMESTAMP) RETURN
VARCHAR2;
```

Funkce vrací masku $def^{(t)}(o)$ pro instanci objektu.

A.4.30 NEXT_OBJECT_BY_ATTRS

```
FUNCTION NEXT_OBJECT_BY_ATTRS(ATTRS IN VARCHAR2, VALID_TIME IN TIMESTAMP,
OBJ_ID IN NUMBER) RETURN NUMBER;
```

Funkce vrací id instance objektu, který odpovídá třídě definované množinou atributů *ATTRS* v časový okamžik *VALID_TIME*. Funkce prochází pouze objekty, které právě odpovídají dané třídě. Funkce prochází takové instance, které mají větší id než *OBJ_ID*, pokud v úložišti neexistuje žádný vyhovující objekt, vrací 0.

A.4.31 NEXT_OBJECT_BY_MASK

```
FUNCTION NEXT_OBJECT_BY_MASK(MASK IN VARCHAR2, VALID_TIME IN TIMESTAMP,
OBJ_ID IN NUMBER) RETURN NUMBER;
```

Funkce vrací id instance objektu, který odpovídá třídě definované množinou atributů masky *MASK* v časový okamžik *VALID_TIME*. Funkce prochází objekty, které odpovídají dané třídě či jejím potomkům. Funkce prochází takové instance, které mají větší id než *OBJ_ID*, pokud v úložišti neexistuje žádný vyhovující objekt, vrací 0.

A.4.32 OBJECTS_AGE

```
FUNCTION OBJECTS_AGE(VALID_TIME IN TIMESTAMP) RETURN OBJ_AGE_TABLE_TYPE
PIPELINED;
```

Funkce vrátí věk objektů v časový okamžik VALID_TIME. Objekty, které by v tuto dobu ještě nežily, jsou ignorovány.

A.4.33 OBJ_TO_IDENT

```
FUNCTION OBJ_TO_IDENT(OBJ_ID IN NUMBER) RETURN VARCHAR2;
```

Funkce vrátí jednoznačný identifikátor objektu podle id. Pokud takový objekt neexistuje, vrací hodnotu *null*.

A.4.34 IDENT_TO_OBJ

```
FUNCTION IDENT_TO_OBJ(OBJ_IDENTIFICATOR IN VARCHAR2) RETURN NUMBER;
```

Funkce vrátí id objektu s daným identifikátorem. Pokud takový objekt neexistuje, vrací hodnotu 0.

A.5 Balík RS_INX

Balík s podporou indexů nad rozvolněnými objekty. V tomto balíku lze vytvářet oba dva druhy implementovaných indexů a přistupovat k nim stejně. Liší se pouze syntaxe selekce podle druhu použitého indexu.

A.5.1 CREATE_INX_ ABSTRACT

```
PROCEDURE CREATE_INX_ABSTRACT(INX_NAME IN VARCHAR2)
```

Procedura interně volá funkci *CREATE_INX_ABSTRACT_RET_ID*.

A.5.2 CREATE_INX_ ABSTRACT _RET_ID

```
FUNCTION CREATE_INX_ ABSTRACT _RET_ID (INX_NAME IN VARCHAR2) RETURN NUMBER
```

Funkce vytvoří abstraktní, zatím nepoužitelný index. Vrací id nově vytvořeného indexu. Pokud index s daným jménem již existuje vyhazuje, výjimku *INDEX_EXISTS*.

A.5.3 LAST_CREATED_INX

```
FUNCTION LAST_CREATED_INX RETURN NUMBER
```

Funkce vrací id naposledy vytvořeného indexu v dané session. Pokud zatím žádný index nebyl vytvořen vrací hodnotu 0.

A.5.4 CREATE_INX_LEGAL

```
PROCEDURE CREATE_INX_LEGAL (INX_NAME IN VARCHAR2, RS_INX_TYPE NUMBER,
INDEX_ALL IN NUMBER);

PROCEDURE CREATE_INX_LEGAL (INX_ID IN NUMBER, RS_INX_TYPE NUMBER,
INDEX_ALL IN NUMBER);
```

Procedura vytvoří z *abstraktního indexu* skutečný index nad atributy. Hodnota *INX_TYPE_RO* (9) parametru *RS_IX_TYPE* značí, zda se má vytvořit *externí index* v úložišti, jinak bude vytvořena tabulka z indexovaných hodnot a nad touto tabulkou se vytvoří *interní index* Oracle. Pokud není *abstraktní index* definován na žádném atributu, vyhazuje výjimku

ATTR_NOT_FOUND. V případě vytvoření skutečného indexu nad více než čtyři atributy vyhazuje výjimku *TOO_MANY_INDEXED_ATTRS*. Při pokusu vytvořit nad určitým atributem více než čtyři skutečné indexy se vyhazuje výjimka *TOO_MANY_INDEXES_OVER_ATTR*. Poslední parametr *INDEX_ALL* indikuje, zda se mají indexovat pouze hodnoty modifikovaných atributů od vzniku indexu či se mají indexovat i hodnoty, které byly vloženy do úložiště dříve. Chování se indikuje podle konstant *INX_CREATE_REINDEX_ALL* (9) a *INX_CREATE_NO_REINDEX_ALL*(9).

A.5.5 DROP_INX

```
PROCEDURE DROP_INX (INX_NAME IN VARCHAR2);
PROCEDURE DROP_INX (INX_ID IN NUMBER);
```

Smaže jak *abstraktní* tak i *skutečný index* s daným jménem.

A.5.6 ADD_ATTR

```
PROCEDURE ADD_ATTR (INX_NAME IN VARCHAR2, ATTR_NAME IN VARCHAR2);
PROCEDURE ADD_ATTR (INX_NAME IN VARCHAR2, ATT_ID IN NUMBER);
PROCEDURE ADD_ATTR (INX_ID IN NUMBER, ATTR_NAME IN VARCHAR2);
PROCEDURE ADD_ATTR (INX_ID IN NUMBER, ATT_ID IN NUMBER);
```

Přiřadí k *abstraktnímu indexu* daný atribut. Pokud je již atribut k *abstraktnímu indexu* přiřazen vyhazuje výjimku *ATTR_HAS_ALREADY_ADDED*.

A.5.7 DEL_ATTR

```
PROCEDURE DEL_ATTR (INX_NAME IN VARCHAR2, ATTR_NAME IN VARCHAR2);
PROCEDURE DEL_ATTR (INX_NAME IN VARCHAR2, ATT_ID IN NUMBER);
PROCEDURE DEL_ATTR (INX_ID IN NUMBER, ATTR_NAME IN VARCHAR2);
PROCEDURE DEL_ATTR (INX_ID IN NUMBER, ATT_ID IN NUMBER);
```

Odstraní od *abstraktního indexu* atribut indexu.

A.5.8 ADD_CLSS

```
PROCEDURE ADD_CLSS(INX_NAME IN VARCHAR2, CLSS_NAME IN VARCHAR2);
PROCEDURE ADD_CLSS(INX_NAME IN VARCHAR2, CLSS_ID IN NUMBER);
PROCEDURE ADD_CLSS(INX_ID IN NUMBER, CLSS_NAME IN VARCHAR2);
PROCEDURE ADD_CLSS(INX_ID IN NUMBER, CLSS_ID IN NUMBER);
```

K *abstraktnímu indexu* přidá všechny atributy dané třídy. Pokud je již nějaký atribut přidán vyhazuje výjimku *ATTR_HAS_ALREADY_ADDED*.

A.5.9 ADD_CLSS_OVER

```
PROCEDURE ADD_CLSS_OVER(INX_NAME IN VARCHAR2, CLSS_NAME IN VARCHAR2);
PROCEDURE ADD_CLSS_OVER(INX_NAME IN VARCHAR2, CLSS_ID IN NUMBER);
PROCEDURE ADD_CLSS_OVER(INX_ID IN NUMBER, CLSS_NAME IN VARCHAR2);
PROCEDURE ADD_CLSS_OVER(INX_ID IN NUMBER, CLSS_ID IN NUMBER);
```

K indexu přidá všechny atributy dané třídy. Pokud je již nějaký atribut přidán, bude ignorován.

A.5.10 DEL_CLSS

```
PROCEDURE DEL_CLSS(INX_NAME IN VARCHAR2, CLSS_NAME IN VARCHAR2);
PROCEDURE DEL_CLSS(INX_NAME IN VARCHAR2, CLSS_ID IN NUMBER);
PROCEDURE DEL_CLSS(INX_ID IN NUMBER, CLSS_NAME IN VARCHAR2);
PROCEDURE DEL_CLSS(INX_ID IN NUMBER, CLSS_ID IN NUMBER);
```

Smaže u *abstraktního indexu* všechny atributy, které definují nějakou pojmenovanou třídu.

A.5.11 SELECT_EQUAL

```
PROCEDURE SELECT_EQUAL(INX_NAME IN VARCHAR2, ATTR1_VALUE IN VARCHAR2,
ATTR2_VALUE IN VARCHAR2, VALID_TIME IN TIMESTAMP);
PROCEDURE SELECT_EQUAL(INX_NAME IN VARCHAR2, ATTR1_VALUE IN VARCHAR2,
ATTR2_VALUE IN VARCHAR2, VALID_TIME IN TIMESTAMP, TABLE_MOD IN VARCHAR2);
PROCEDURE SELECT_EQUAL(INX_ID IN NUMBER, ATTR1_VALUE IN VARCHAR2,
ATTR2_VALUE IN VARCHAR2, VALID_TIME IN TIMESTAMP);
PROCEDURE SELECT_EQUAL(INX_ID IN NUMBER, ATTR1_VALUE IN VARCHAR2,
ATTR2_VALUE IN VARCHAR2, VALID_TIME IN TIMESTAMP, TABLE_MOD IN VARCHAR2);
PROCEDURE SELECT_EQUAL(INX_NAME IN VARCHAR2, ATTR1_VALUE IN VARCHAR2,
ATTR2_VALUE IN VARCHAR2, VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP);
PROCEDURE SELECT_EQUAL(INX_NAME IN VARCHAR2, ATTR1_VALUE IN VARCHAR2,
ATTR2_VALUE IN VARCHAR2, VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP,
TABLE_MOD IN VARCHAR2);
PROCEDURE SELECT_EQUAL(INX_ID IN NUMBER, ATTR1_VALUE IN VARCHAR2,
ATTR2_VALUE IN VARCHAR2, VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP);
PROCEDURE SELECT_EQUAL(INX_ID IN NUMBER, ATTR1_VALUE IN VARCHAR2,
ATTR2_VALUE IN VARCHAR2, VALID_FROM IN TIMESTAMP, VALID_TO IN TIMESTAMP,
TABLE_MOD IN VARCHAR2);
```

Procedura v externím indexu nalezne instance objektů podle hodnot atributů. Vyhledání je zkrácením intervalového dotazu. *ATTR1_VALUE*, *ATTR2_VALUE* s *VALID_TIME* (případně dvojice *VALID_FROM* a *VALID_TO*) je hledaný klíč. *TABLE_MOD* je přepínač, zda se mají vyhledané výsledky přepsat či nikoliv podle modifikátoru *INX_MOD_CLEAR* (9) a *INX_MOC_APPEND* (9). *Null* hodnota proměnné *VALID_TIME* je nahrazena aktuálním časovým razítkem, *null* hodnota v proměnné

VALID_FROM je nahrazena minimálním časovým razítkem v úložišti a *null* hodnota proměnné *VALID_TO* je nahrazena maximálním časovým razítkem.

A.5.12 SELECT_IN

```
PROCEDURE SELECT_IN(INX_NAME IN VARCHAR2, ATTR1_FROM IN VARCHAR2, ATTR1_TO
IN VARCHAR2, ATTR2_FROM IN VARCHAR2, ATTR2_TO IN VARCHAR2, VALID_TIME IN
TIMESTAMP);

PROCEDURE SELECT_IN(INX_NAME IN VARCHAR2, ATTR1_FROM IN VARCHAR2, ATTR1_TO
IN VARCHAR2, ATTR2_FROM IN VARCHAR2, ATTR2_TO IN VARCHAR2, VALID_TIME IN
TIMESTAMP, TABLE_MOD IN VARCHAR2);

PROCEDURE SELECT_IN(INX_ID IN NUMBER, ATTR1_FROM IN VARCHAR2, ATTR1_TO IN
VARCHAR2, ATTR2_FROM IN VARCHAR2, ATTR2_TO IN VARCHAR2, VALID_TIME IN
TIMESTAMP);

PROCEDURE SELECT_IN(INX_ID IN NUMBER, ATTR1_FROM IN VARCHAR2, ATTR1_TO IN
VARCHAR2, ATTR2_FROM IN VARCHAR2, ATTR2_TO IN VARCHAR2, VALID_TIME IN
TIMESTAMP, TABLE_MOD IN VARCHAR2);

PROCEDURE SELECT_IN(INX_NAME IN VARCHAR2, ATTR1_FROM IN VARCHAR2, ATTR1_TO
IN VARCHAR2, ATTR2_FROM IN VARCHAR2, ATTR2_TO IN VARCHAR2, VALID_FROM IN
TIMESTAMP, VALID_TO IN TIMESTAMP);

PROCEDURE SELECT_IN(INX_NAME IN VARCHAR2, ATTR1_FROM IN VARCHAR2, ATTR1_TO
IN VARCHAR2, ATTR2_FROM IN VARCHAR2, ATTR2_TO IN VARCHAR2, VALID_FROM IN
TIMESTAMP, VALID_TO IN TIMESTAMP, TABLE_MOD IN VARCHAR2);

PROCEDURE SELECT_IN(INX_ID IN NUMBER, ATTR1_FROM IN VARCHAR2, ATTR1_TO IN
VARCHAR2, ATTR2_FROM IN VARCHAR2, ATTR2_TO IN VARCHAR2, VALID_FROM IN
TIMESTAMP, VALID_TO IN TIMESTAMP);

PROCEDURE SELECT_IN(INX_ID IN NUMBER, ATTR1_FROM IN VARCHAR2, ATTR1_TO IN
VARCHAR2, ATTR2_FROM IN VARCHAR2, ATTR2_TO IN VARCHAR2, VALID_FROM IN
TIMESTAMP, VALID_TO IN TIMESTAMP, TABLE_MOD IN VARCHAR2);
```

Procedura v externím indexu nalezne instance objektů podle hodnot atributů. Vyhledání je podle zadaného intervalu hodnot. *ATTR1_FROM*, *ATTR1_TO*, *ATTR2_FROM* a *ATTR2_TO* s *VALID_TIME* (případně dvojice *VALID_FROM* a *VALID_TO*) je hledaný klíč. *TABLE_MOD* je přepínač zda se mají vyhledané výsledky přepsat či nikoliv podle modifikátoru *INX_MOD_CLEAR* (9) a *INX_MOC_APPEND* (9). *Null* hodnota proměnné *VALID_TIME* je nahrazena aktuálním časovým razítkem, *null* hodnota v proměnné *VALID_FROM* je nahrazena minimálním časovým razítkem v úložišti a *null* hodnota proměnné *VALID_TO* je nahrazena maximálním časovým razítkem.

Příloha B Vlastní rozhraní nad OCI

Nad OCI bylo vytvořeno omezené rozhraní, která umožňuje pohodlnější práci programátora při komunikaci systému Oracle a externí procedurou. Vytvořené rozhraní je obdobou objektového rozhraní OCCI, které však pro nedostatečnou dokumentaci nebylo možné použít. Především nebylo možné z předáváného OCI service kontext parametru získat odpovídající OCCI instance pro existující spojení s databází.

Definované rozhraní se stává ze čtyř obalujících objektů použitých při implementaci externího indexu. Jsou to objekty tříd *RSBuzy, RSStatement, RSBlob* a *RSTimeStamp*. Funkce všech objektových wrapperů mají návratový typ *integer*, jehož hodnota rovna nule indikuje, že se daná operace provedla korektně. Všechny třídy vlastního rozhraní jsou deklarované v souboru *rsbuzy.h*, kde je tedy kompletní seznam hlaviček všech funkcí.

Třída *RSBuzy* je obalující třída pro kontext, všechny handly a návratové hodnoty. Výpis nejdůležitějších hlaviček funkcí je uveden v kódu (Kód 10). Funkce *newBuzy* vytvoří a inicializuje kompletně rozhraní. Funkce *delBuzy* rozhraní ukončí a vrátí hodnotu číselného typu Oracle. Předpokládá se, že všechny externí procedury externího indexu v Oracle budou vracet číselné hodnoty.

```
class RSBuzy {
public:
     RSBuzy(RSBuzyData *data);
     ~RSBuzy();

     int newStatement(const char *stmtText, RSStatement **statement);
     int newBlob(RSBlob **blob);
     int newTimeStamp(RSTimeStamp **timeStamp);

     int delStatement(RSStatement **stmt);
     int delBlob(RSBlob **blob);
     int delTimeStamp(RSTimeStamp **timeStamp);
};

int newBuzy(OCIExtProcContext *ctx, RSBuzy **buzy);
OCINumber *delBuzy(RSBuzy ** buzy);
```

Kód 10 – třída *RSBuzy*

Třída *RSStatement*, jak již název napovídá, obaluje funkce pro výkonný blok. Nejvýznamnější funkce jsou uvedené v kódu (Kód 11).

```
class RSStatement {
public:
     RSStatement(RSBuzyData *buzyData);
     ~RSStatement();
```

```
      int bindString(const int pos, char *var, const int maxSize);
      int bindInt(const int pos, int *var);
      int bindBlob(const int pos, RSBlob *blob);
      int bindTimestamp(const int pos, RSTimeStamp *timeStamp);

      int defineString(const int pos, char *var, const int maxSize);
      int defineInt(const int pos, int *var);
      int defineBlob(const int pos, RSBlob *blob);
      int defineTimestamp(const int pos, RSTimeStamp *timeStamp);

      int execute();

      int readFetchNext();
      int readStop();
};
```

Kód 11 – třída *RSStatement*

Pro práci s bloby je vytvořena třída *RSBlob* s nejvýznamnějšími funkcemi deklarovaných v kódu (Kód 12).

```
class RSBlob {
public:
      RSBlob(RSBuzyData *buzyData);
      ~RSBlob();

      int open(const ub1 mode = OCI_LOB_READONLY);
      int close();

      int write(const int offSet, const char *buff, const int buffSize);
      int read(const int offSet, char *buff, const int buffSize);
};
```

Kód 12 – třída *RSBlob*

Pro práci s typem Oracle časových razítek je vytvořena třída *RSTimeStamp*. Nejvýznamnější funkce této třídy jsou uvedené v kódu (Kód 13).

```
class RSTimeStamp {
public:
      RSTimeStamp(RSBuzyData *buzyData, OCIDateTime *dateTime = NULL);
      ~RSTimeStamp();

      int getDateTime(int *year, int *mounth, int *day, int *hour, int
            *minute, int *second, int *miliSecond);
      int setDateTime(int year, int mounth, int day, int hour, int minute,
            int second, int miliSecond);

      int compare(RSTimeStamp *timeStamp, sword *res);
      int toString(char *buff, const int buffSize);

      bool isNull();
};
```

Kód 13 – třída *RSTimeStamp*

V kódu (Kód 14) je ukázka použití při výběru všech id klientů s křestním jménem *Karel*. Syntaxe je jednotná ať pro načtení jednoho či více řádků. Funkce *execute* třídy *RSStatement* vykoná dotaz a v případně selectu vrátí první řádek požadovaných dat. Funkce *readFetchNext* již jenom dotahuje data pro další řádky, pokud takové existují.

```
OCINumber *test1(OCIExtProcContext *ctx) {

     RSBuzy *buzy = NULL;
     RSStatement *stmt = NULL;

     if (newBuzy(ctx, &buzy))
          return delBuzy(&buzy);

     char sql[2000];
     sprintf_s(sql, sizeof(sql), "SELECT ID FROM TABLE USERS WHERE
          FIRSTNAME = :FIRSTNAME FOR UPDATE");

     if (buzy->newStatement(sql, &stmt))
          return cleanUp(&buzy, &stmt);

     char *firstName = "Karel";
     if (stmt->bindString(1, firstName, strlen(firstName)))
          return cleanUp(&buzy, &stmt);

     int id;
     if (stmt->defineInt(1, *id))
          return cleanUp(&buzy, &stmt);

     if (stmt->execute() == 0)
     do {
          cout << id << endl;
     } while (stmt->readFetchNext() == 0);

     return cleanUp(&buzy, &stmt);
}
```

Kód 14 - test1

Ukázka vložení struktury do blobu je v kódu (Kód 15).

```
struct QQQ {
     int a,b;
};

OCINumber *test2(OCIExtProcContext *ctx) {
     RSBuzy *buzy = NULL;
     RSStatement *stmt = NULL;
     RSBlob *blob = NULL;

     if (newBuzy(ctx, &buzy))
          return delBuzy(&buzy);

     char sql[2000];
     sprintf_s(sql, sizeof(sql), "INSERT INTO BLOBY VALUES(1,
          EMPTY_BLOB()) RETURNING B INTO :B");
```

```
     if (buzy->newStatement(sql, &stmt))
           return cleanUp(&buzy, &stmt, &blob);

     if (buzy->newBlob(&blob))
           return cleanUp(&buzy, &stmt, &blob);

     if (stmt->bindBlob(1, blob))
           return cleanUp(&buzy, &stmt, &blob);

     if (stmt->execute())
           return cleanUp(&buzy, &stmt, &blob);

     if (blob->open(OCI_LOB_READWRITE))
           return cleanUp(&buzy, &stmt, &blob);

     QQQ qqq;
     if (blob->write(1, (char *)&qqq, sizeof(qqq)))
           return cleanUp(&buzy, &stmt, &blob);

     if (blob->close())
           return cleanUp(&buzy, &stmt, &blob);

     return cleanUp(&buzy, &stmt, &blob);
}
```

Kód 15 - test2

Ukázka použití typu pro časová razítka je v kódu (Kód 16).

```
OCINumber *test3(OCIExtProcContext *ctx) {
     RSBuzy *buzy = NULL;
     RSStatement *stmt = NULL;
     RSTimeStamp *st= NULL;

     if (newBuzy(ctx, &buzy))
           return delBuzy(&buzy);

     char sql[2000];
     sprintf_s(sql, sizeof(sql), "BEGIN :1 := SYS_TIMESTAMP(); END;");

     if (buzy->newStatement(sql, &stmt))
           return cleanUp(&buzy, &stmt, &ts);

     if (buzy->newTimeStamp(&ts))
           return cleanUp(&buzy, &stmt, &ts);

     if (stmt->bindTimeStamp(1, ts))
           return cleanUp(&buzy, &stmt, &ts);

     if (stmt->execute())
           return cleanUp(&buzy, &stmt, &ts);

     cout << "EVERYTIME FALSE: " << ts->isNull() << endl;

     return cleanUp(&buzy, &stmt, &ts);
}
```

Kód 16 - test3

Příloha C Mapa CD

\bin	Zkompilované kódy úložiště
\bin\samples	Ukázkové příklady
\soft	Freewarový software použitý při vývoji
\source\cpp	Zdrojový kód knihovny v C++
\source\ora	Zdrojový kód databázových objektů
\source\samples	Zdrojové kódy ukázkových příkladů
\text	Elektronická verze diplomové práce