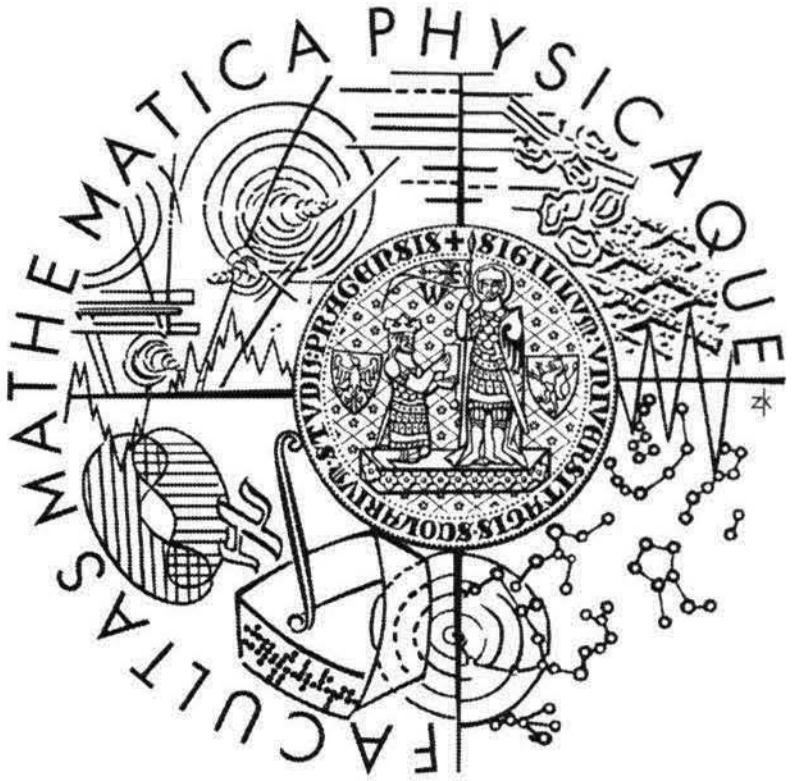
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Martin Paľo

Přínosy objektově-relační technologie

Katedra softwarového inženýrství
Vedoucí diplomové práce: *RNDr. Antonín Říha, CSc.*
Studijní program: *Informatika, Datové inženýrství*

Na tomto místě bych rád poděkoval vedoucímu své diplomové práce RNDr. Antonínu Říhovi, CSc. za odborné rady a náměty, které přispěly k napsání této práce.

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

V Praze dne 11. srpna 2006

Martin Paľo

OBSAH

Název práce: Přínosy objektově-relační technologie
Autor: Martin Paľo
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Antonín Říha, CSc.
e-mail vedoucího: Antonin.Riha@mff.cuni.cz

Abstrakt

Objektově-relační databáze se v poslední době těší veliké popularitě. Oproti tradičním přístupům poskytují mnoho nových možností. Práce popisuje obecné požadavky na objektově-relační systémy, nová objektově-relační rozšíření systému Oracle a obdobná rozšíření v dalších databázových systémech a jejich srovnání s požadavky databázového standardu SQL:1999. Podstatnou částí práce je srovnání relační a objektově-relační implementace studijního informačního systému, provedené s použitím databázového serveru Oracle 9i.

Klíčová slova: objektově-relační databáze, hnízděná tabulka, SQL:1999, Oracle

Title: Contributions of the object-relational technology
Author: Martin Paľo
Department: Department of Software Engineering
Supervisor: RNDr. Antonín Říha, CSc.
Supervisor's email adress: Antonin.Riha@mff.cuni.cz

Abstract

Lately, object-relational databases have become increasingly popular. Compared to the traditional approach, they provide many new possibilities. This thesis describes the general requirements on object-relational systems and the new object-relational enhancement of Oracle, as well as similar enhancements of other well-known database systems and their comparison with the SQL:1999 database standard requirements. An essential part of this thesis is concerned with the comparison of the relational and object-relational implementation of the student information system using the Oracle 9i database server.

Keywords: object-relational database, nested table, SQL:1999, Oracle

1 Úvod

1.1 Cíl práce

Diplomová práce si klade za cíl posoudit skutečné přínosy objektově-relační (dále jen OR) technologie v současných databázových systémech a popsat hlavní výhody a nedostatky spojené s používáním OR technologie v porovnání s čistě relační technologií. Za tímto účelem byla s využitím OR databáze Oracle vytvořena testovací implementace studijního informačního systému jak s využitím relační datové vrstvy, tak s využitím možností nabízených OR rozšířením a obě varianty byly následně otestovány. Snaží se ověřit, nakolik jsou teoretické předpoklady OR přístupu použitelné v praxi, nakolik je jejich výhodnost závislá na konkrétní implementaci typu aplikace a použitých strukturách při implementaci objektového návrhu aplikace.

1.2 Obsah práce

Práce je tématicky rozdělena do osmi kapitol. Druhá kapitola je věnována stručnému shrnutí nevýhod relačního datového modelu, tak jak jej najdeme v literatuře. Třetí kapitola popisuje rozdělení databázových systémů podle způsobu jejich přístupu k datům a manipulaci s nimi. Zatímco čtvrtá kapitola se věnuje obecnému popisu vlastností, požadovaných od OR databází, pátá kapitola popisuje skutečnou implementaci nových OR rysů v databázi Oracle. Vzhledem k faktu, že se implementace objektově-relačních rozšíření v jednotlivých existujících databázích značně liší, uvádí šestá kapitola pro srovnání, jak definuje tato rozšíření samotná norma SQL, a jak tato rozšíření realizují databáze DB2 a Informix. Sedmá kapitola popisuje databázovou aplikaci, na které byly prakticky otestovány nové postupy pro OR verzi a porovnávány s přístupy relačními. Obsahuje rovněž naměřené charakteristiky a výsledky, získané analýzou prováděných dotazů nad oběma verzemi.

2 Současný stav

Ještě v nedávné minulosti měly dominantní postavení ve světě databázových systémů databázové systémy založené na relačním modelu. Tyto systémy jsou na scéně poměrně dlouhou dobu a mají oproti dříve používaným modelům řadu výhod.
Důležitou výhodou, kterou relační model přinesl, je oddělení fyzického a logického pohledu na data, což umožňuje pracovat s daty nezávisle na jejich fyzickém uložení. Navíc existuje propracovaná teorie relačního modelu.

2.1 Nevýhody relačního datového modelu

To, jaká data budou použita, závisí na charakteru vytvářené aplikace a její složitosti. U složitějších aplikací, kdy je vhodné v aplikační vrstvě použít složitějších datových struktur, nejsou relační systémy vyhovující. Podstatné nedostatky jsou uvedeny v následujícím přehledu:

- Malá modelovací síla neumožňující reprezentovat hodnoty prostřednictvím složitých interních struktur. SŘDB (systém řízení báze dat) uchovává data v databázi v normalizovaných tabulkách. Řádky odpovídají záznamům a sloupce atributům. Každý sloupec má přiřazen jeden z atomických datových typů, jejichž počet je omezen. Nejsou povolena pole proměnné délky ani jinak strukturované hodnoty. Vztahy mezi tabulkami nejsou explicitní, ale jsou realizovány prostřednictvím hodnot ve specifických sloupcích, přes cizí klíče. K realizaci mnoha vztahů je nutno použít spojovacích tabulek.
- Neefektivní transakční zpracovávání. Relační databázové systémy jsou optimalizovány pro transakční zpracování jednoduchých dat.
- Problémy s identifikací. Všechna zpracování jsou založena na hodnotách polí v záznamech. Záznamy nemají jednoznačné identifikátory, které by byly neměnné během jejich života. Není přitom možné provádět odkazování z jednoho záznamu na jiný.
- Problémy s výpočetní silou. Omezením dotazovacího jazyka SQL v relačním modelu je např. neexistence rekurzivních dotazů.
- Neefektivní podpora datové hierarchie.
- Nemožnost definovat vlastní operace.
- Nedostatečná podpora pro nestandardní aplikace – obrázky, prostorová data …
- Nedostatečná podpora integritních omezení.

Vzhledem k omezeným možnostem relačního modelu, je vhodné relační systémy nasazovat především v aplikacích využívajících relativně jednoduchá data, mezi kterými existují jednoduché vztahy. Složitější vztahy se realizují pomocnými vztahovými relacemi a spojeními mezi nimi. Čím více vztahů mezi daty, tím víc spojení je zapotřebí.
Z uvedeného plyne, že relační databázové systémy nejsou příliš vhodné pro aplikace jako:

- CASE systémy
- MCAD systémy
- GIS systémy
- Medicínské aplikace

Všechny uvedené aplikace pracují se složitými daty.

Mezi hlavní požadavky současných aplikací patří:

- Rozšiřitelnost, resp. možnost definovat nové datové typy.
- Podpora zpracování multimediálních dat. Vytváření složitých objektů a manipulace s mimi.
- Aplikace musí být zásobeny primitivy manipulujícími s objekty jako s celky a primitivy umožňujícími manipulovat s komponentami těchto objektů.
- Zabezpečovací a přístupový mechanismus musí být založen na objektech.

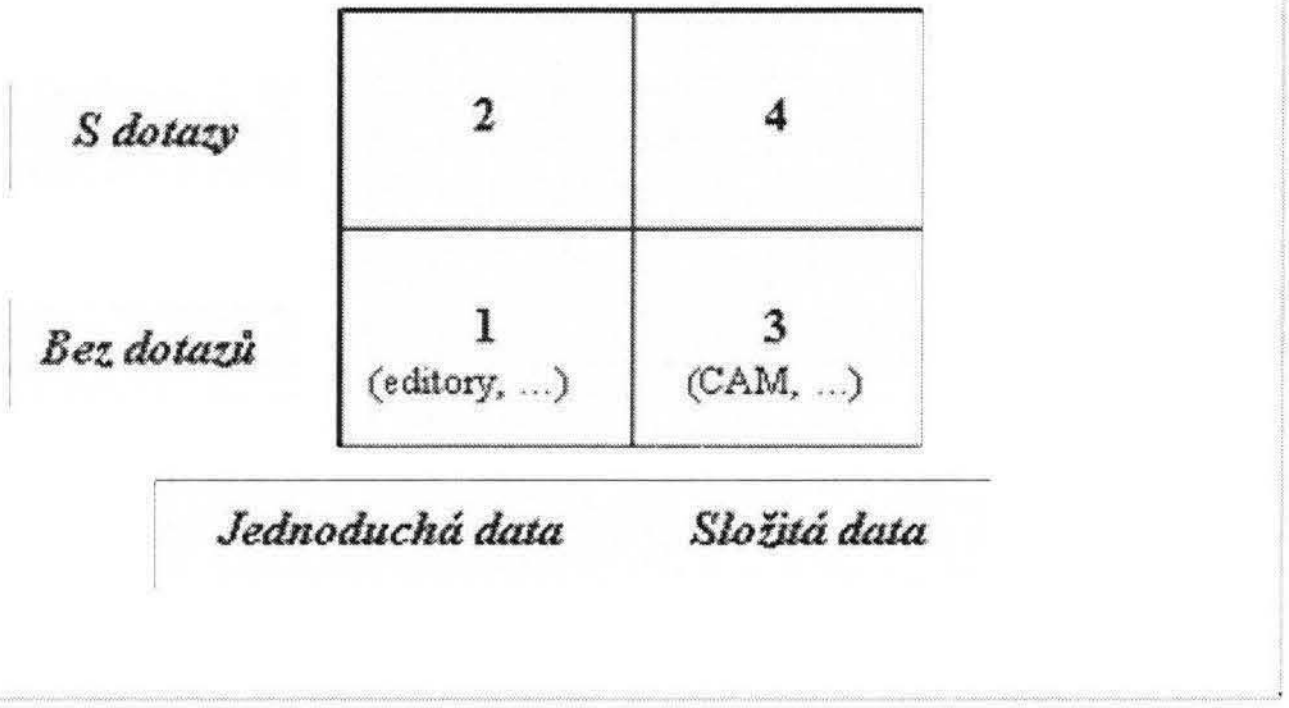
Rostoucí nároky uživatelů a stále větší počet aplikací, pro které je relační technologie nedostatečná, vedly k prosazování nových databázových technologií a jejich postupné adopci relačními databázemi.

3 Nové trendy

V současnosti existují dva nové přístupy k databázím.
Prvním z nich je OR přístup, jehož cílem je zlepšovat a zjednodušovat datové modelování a dotazování při práci se složitými datovými strukturami. Aplikace vhodné pro tento typ databází jsou zaměřeny na uchovávání složitých dat včetně multimediálních a dotazování nad nimi.
Druhý přístup představují objektově-orientované (dále jen OO) databázové systémy, které se zaměřují na aplikace s vysokými požadavky na výkonnost.

3.1 Klasifikace databázových systémů

V následujících dvou kapitolách jsou rozsáhle využívány poznatky knihy [1].
Její autoři rozdělují aplikace do čtyř obecných typů, jak zachycuje následující matice.



Obr. 3.1 Klasifikace databázových aplikací

Svislá osa rozlišuje, zda aplikace vyžaduje dotazovací schopnost, vodorovná osa zobrazuje jednoduchá resp. složitá data. Přitom samozřejmě neexistuje pevná hranice mezi jednoduchými a složitými daty, podobně je tomu i u schopnosti dotazování.
Aplikace tak lze zařadit na základě jejich vlastností do jednoho z uvedených kvadrantů, přičemž některé aplikace mohou náležet do více kvadrantů současně. Pro účely této práce budou zajímavé především systémy spadající do kategorie 2. a 4., tedy systémy, vyžadující ukládání jednoduchých a složitých dat s dotazy.
K popisu aplikací spadajících do druhého kvadrantu reprezentujícího *jednoduchá data s dotazy* může posloužit jednoduchý databázový příklad. Předpokládejme, že máme databázi obsahující dvě tabulky reprezentující studenty a školy, na kterých studují. U studentů máme kromě jména zaznamenanou adresu bydliště, datum začátku studia a průměr všech známek. Záznamy školy obsahují název školy a adresu. Za těchto předpokladů můžeme prostřednictvím SQL sestavovat např. následující dotazy:

1. Najdi všechny studenty, kteří nastoupili na školu v roce 2002 a jejich průměr známek nepřevyšuje 1.5.
2. Najdi počty žáků na jednotlivých školách.

Do čtvrté kategorie, vyžadující *složitá data s dotazy*, lze zařadit plnohodnotný *studijní informační systém* (SIS), jehož základ byl implementován s využitím relačního i OR datového modelu v této práci. Na jedné straně je potřebné evidovat studenty, vyučující, vyučované předměty a studijní plány jednotlivých studentů, na straně druhé je potřeba poskytnout studentům možnost vyhledávat přednášky, semináře a cvičení s ohledem na jejich návaznost, na potřebu splnit požadavky pro úspěšné ukončení studia a řadu dalších aspektů. S podporou OR rozšíření je navíc snadné rozšířit popis přednášek o sylaby s možnosti full-textového vyhledávání v nich, rozšířit evidenci studentů a vyučujících o jejich fotografie, a podobně.
Jiným příkladem takové aplikace může být cestovní kancelář, poskytující zájezdy na různá místa po celém světě, a která o každé destinaci eviduje fotografický materiál. Zákazníci si pak mohou jednotlivé destinace vybírat na základě obsahu obrázků, patřících k jednotlivým destinacím. Např. podle přístupu k moři resp. k horám. Datová struktura udržující informace o jednotlivých obrázcích by mohla obsahovat kromě identifikátoru a vlastního obrázku textový popisek obsahující atributy obrázku.
Ruční klasifikace obrázků je ale nepraktická. Krom toho by při ní mohl být vynechán právě atribut zajímající zákazníka. Lepším způsobem by bylo použití některé z technik porovnávání vzorů.

3.2 OO databázové systémy

OO databázové systémy byly navrženy jako další generace databázových systémů překonávající omezení tradičního relačního databázového modelu a bez ohledu na zpětnou kompatibilitu s nim. Se složitými informacemi pracují efektivněji než tradiční relační systémy, protože jsou tyto informace uchovány jako celek v databázi. OO systémy navíc umožňují navigaci mezi objekty a vztahy mezi nimi s využitím referencí.
Objektová databázová technologie je založena na následujících konceptech:

- **Identifikace objektů** – Každému objektu je přidělen jednoznačný identifikátor – *objektový identifikátor OID*, který se během života objektu nemění.
- **Složené objekty** – Každý objekt obsahuje množinu atributů, jejichž hodnotou může být znovu objekt resp. množina objektů.
- **Zapouzdření** – Stav objektu je reprezentován hodnotami jeho atributů. Tento stav se mění prostřednictvím zpráv vyvolávajících odpovídající objektové metody.
- **Třídy** – Objekty stejné struktury a chování tvoří třídu reprezentující šablonu objektů ve třídě. Každý objekt je instancí nějaké třídy.
- **Dědičnost** – Jednotlivé třídy mohou být odvozeny od jiných tříd. Podtřída je definovaná jako speciální případ, který dědí všechny atributy a metody svojí nadtřídy. Dědičnost může přitom být jednoduchá nebo vícenásobná. Dědičnost tvoří základ hierarchie tříd.
- **Přetížení** – Různé metody nemusí mít vždy rozdílná jména. To v případech, kdy se tyto metody liší počtem resp. typem svých parametrů.

Vývoj OO databázových systémů začal přibližně před 15 lety. Podporují aplikace, pro které bylo relační řešení nedostačující. To ale neznamená, že jsou ve všech směrech lepší než relační databázové systémy. Své využití nacházejí hlavně u aplikací pracujícími se složitými strukturami jako jsou:

- CAD systémy
- CASE systémy
- GIS systémy
- Systémy v medicíně
- Systémy pro uchovávání a získávání dokumentů
- Multimediální aplikace

a u některých dalších.

3.3 OR databázové systémy

Jedná se o databázové systémy, které se snaží sjednotit rysy jak relačních, tak objektových databází. OR SŘBD je specifikován v rozšíření SQL standardu — SQL:1999 (viz kap. 6.1). Podrobnějšímu popisu těchto systémů se věnuje následující kapitola.

4 Objektově-relační SŘBD – obecné požadavky

OR databáze jsou zatím posledním vývojovým článkem v historii databází. Produkty výrobců relačních databázových systémů jako jsou IBM, Informix, Oracle, UniSQL a jiných zahrnují OO rysy do svých produktů jako odpověď na vzrůstající požadavky ukládat objekty aplikačního modelu do databáze s využitím užitečných prvků OO technologie.
Pro OO modelování je charakteristická především bohatost typů objektů, které jsou k dispozici. Složitost těchto objektů plyne nejenom z jejich struktury, ale i z hlediska vzájemných vztahů. V relačních databázích založených na normalizovaných tabulkách šlo modelovat takový svět pouze prostřednictvím jednoduchých datových struktur a přístupů za cenu často složitého a neefektivního přístupu k datům. Silnou stránkou relačních systémů přitom je možnost práce s jednoduchými daty s relativně silnými dotazovacími prostředky. Přitom existuje velké množství aplikací, které jsou implementovány pomocí relační technologie a není nutné či potřebné je rekonstruovat do technologie jiné.

4.1 Základy objektově-relačních databází

OR databázové systémy rozšiřují tradiční relační databázový model. Přidávají možnosti pro uchovávání, manipulaci a získávání složitých datových typů jako jsou obrázky, video, časové události a webové stránky. Podporují množství nových datových typů, které mohly být předtím realizovány jenom pomocí binárních objektů. Povolují také definovat nové datové typy z již existujících jednodušších, čímž zapouzdřují jejich interní strukturu a atributy. Kromě toho přinášejí i nové způsoby manipulace s daty a efektivního vyhledávání.
Kromě konceptů OO databází, uvedených v kapitole 3.2, objektově relační systémy poskytují i následující rozšíření oproti databázím relačním:

- **Uživatelsky definované funkce** umožňují definovat metody pro vytváření, uchovávání a přístup k novým datovým typům.
- **Uživatelsky definované indexové struktury** slouží k efektivnějšímu získávání uložených dat jako jsou textová data, obrázky atd. Tradiční relační indexové metody jsou pro tyto typy dat nedostačující.
- **Rozšířené optimalizátory**, které umožňují na základě ohodnocení ceny uživatelsky definovaných funkcí a indexových struktur určit nejefektivnější způsob vyhodnocení dotazu.

Mezi další důležité rysy OR databázových systémů patří schopnost přistupovat ke všem datům a funkcím prostřednictvím jednoduchých dotazů.

V následující části budou podrobněji popsány tyto rysy OR SŘBD

1) rozšíření základních datových typů v kontextu SQL
2) podpora složitých objektů
3) dědičnost
4) podpora systému pravidel.

4.2 Rozšíření základních datových typů

Nezbytnou vlastností při řešení problémů patřících do čtvrtého kvadrantu, reprezentujícího složitá data s dotazy, je rozšíření základních datových typů. Díky ní lze řešit problémy, které se v SQL-92[1] řeší jen obtížně.

[1] Standard SQL-92: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt/

Typy, které lze v SQL-92 použít při definování sloupců tabulky jsou omezeny na následující:

- celé číslo
- číslo v pohyblivé řádové čárce
- datum, čas
- numerický a dekadický typ
- znakový řetězec

Pro každý z těchto typů jsou definovány funkce a operátory, které s instancemi těchto typů pracují. Tyto funkce a operátory jsou potřebné pro řadu výpočtů, v některých případech ale nejsou zcela vhodné. Jsou tedy zapotřebí další, v SQL-92 nedefinované funkce, pomocí nichž by se řada problémů dala mnohem snáze naprogramovat.
Prostřednictvím rozšířených datových typů v OR SŘBD lze eliminovat simulaci datových typů (např. simulace datového typu „zeměpisný bod“) a tím zvýšit efektivitu aplikací pracujících s těmito typy.
Při vytváření nových datových typů vycházíme ze základních typů jako jsou *integer*, *string*, *date*. Použitím typových konstruktorů lze z těchto typů vytvářet složitější datové typy. Mezi typické konstruktory patří:

- **Struktura záznamu**: Je-li k dispozici seznam typů T_1, T_2, ..., T_n a odpovídající seznam názvů polí f_1, f_2, ..., f_n , je možné vytvořit záznam s n komponentami.
- **Typové kolekce**: Aplikací operátoru kolekce na předem zvolený datový typ T je možné vytvořit typ kolekce jako pole položek typu T - **array**, seznam položek typu T - **list**, množinu položek typu T - **set**.
- **Referenční typy**: Odkaz na typ T je typ, jehož hodnota je vhodná pro lokalizaci hodnoty typu T. Referenční typ používaný v databázových systémech je oproti odkazům známým z programovacích jazyků jako jsou C a C++ složitější. Zatímco pointer představuje adresu do virtuální paměti, reference v sobě zahrnuje odkaz na fyzickou oblast paměti plus přídavné informace.

Po vytvoření nového datového typu je nutné provést ještě jeden krok – specifikovat způsob převodu znakového řetězce na instance nového datového typu. Tento převod zajišťuje uživatelsky definovaná funkce. K vlastnímu převodu na znakovou reprezentaci a zpět se v systémech rozšiřitelných datových typů používají konverzní rutiny poskytující poměrně velkou pružnost. Konverzní rutiny mohou provádět libovolnou transformaci.
Při definování datového typu mu lze přiřadit určitá omezení (omezení se může např. týkat maximálně povoleného průměru, který nemůže být překročen při výběru studentů). Nejvhodnějším místem pro provádění takových kontrol jsou právě konverzní rutiny.
Datový typ v OR systémech tvoří narozdíl od relačního systému jak informace, tak možné operace nad tímto typem. Uvedená flexibilita je základním předpokladem k modelování složitých databázových aplikací.

4.3 Uživatelsky definované funkce a operátory

Uživateli musí být umožněno přidávat k nově definovaným typům operace pracující s instancemi těchto typů. Systém tedy musí být rozšiřitelný o nové funkce a operátory. Prostředkem k psaní funkcí může být v OR systému OR SQL resp. programovací jazyk třetí generace. Takto vytvořené funkce je pak nutno v systému zaregistrovat - uvést jméno nové funkce, její argumenty, návratový typ a vlastní kód. Funkce napsané v OR SQL se při vykonávání dotazu mohou rozvinout, čímž se zjednodušují uživatelské dotazy bez snížení výkonnosti.
Naproti tomu funkce napsané v jazyku třetí generace jsou „neprůhledné“, tzn. že je exekutivní stroj nemůže rozvinout a při zpracování dotazu je musí zavolat.

Jak již bylo řečeno, systém je možné rozšířit i o nové operátory. Tradiční SQL podporuje celou řadu operátorů (např. relační operátory porovnání). Při bližším pohledu na jakýkoliv operátor zjistíme, že se vlastně jedná o funkci se dvěma argumenty. Chceme-li tedy zavést do systému nový operátor, musíme vytvořit novou funkci, kterou následně svážeme s odpovídajícím operátorem. Při použití nového operátoru pak OR SŘBD volá funkci odpovídající tomuto operátoru.
Kromě funkcí a operátorů, je možné definovat v OR systému i vlastní agregace. Ty je nutno zavádět v případech, kdy standardní agregace (sum, avg, min, max) nestačí. Systém SQL-92 má při tvorbě nových agregací následující požadavky [1]:

1) **Inicializace**. Dříve, než se začne počítat vlastní agregace je nutné inicializovat výpočet. Inicializace se skládá z deklarací dvou proměnných, *počet* a *suma*, a jejich nastavení na nulu.
2) **Iterace**. Pro každou hodnotu ze zkoumané množiny je nutné provést částkovou aritmetickou operaci nutnou k získání požadovaného výsledku.
3) **Dokončení**. Po přečtení poslední hodnoty z množiny je třeba z dosud získané hodnoty v kroku 2) vypočítat výslednou hodnotu agregace.

Tzn., má-li OR SŘBD podporovat uživatelsky definované agregace, musí uživateli umožnit definovat tři funkce:

```
initialize(), iterate(hodnota, stav), finalize(stav)
```

K definování obecnějších agregací by mohl uživatel použít i jiný způsob. V takovém případě by si musel nejdřív vybrat množinu hodnot, které ho zajímají, předložit ji uživatelskému programu, který tuto agregaci spočítá. Toto řešení ale představuje nezanedbatelný úzký profil z hlediska výkonnosti, protože se mezi serverem a klientem musí přenášet značný objem dat a navíc je toto řešení zdrojem častých chyb.

4.4 Složené objekty

Druhým nezbytným požadavkem kladeným na OR systémy je podpora složených objektů, složených ze základních resp. z uživatelsky definovaných typů.
Základními kameny pro tvorbu složených typů jsou následující konstruktory:

- složené datové typy (řádky)
- množiny
- odkazy

Prvním uvedeným složeným datovým typem je typ řádek. Jedná se o záznam obsahující složky, jejichž typy jsou známy databázovému systému. Je vhodný jako typ pro kontejnery (např. tabulky, do kterých lze umísťovat instance typu řádek) a typy obsahující hodnoty, ke kterým se často přistupuje. Datový typ řádek má velký význam při definování nových datových typů prostřednictvím dědičnosti.
Dalším složeným datovým typem jsou různé druhy *množin (kolekcí)* – množiny, seznamy a multimnožiny. Vytvoříme-li nějakou kolekci obsahující prvky stejného typu, pak i tato kolekce musí být sama o sobě typ. Musí být přitom umožněno vytvářet kolekce ze všech druhů datových typů včetně kolekcí samotných.
Efektivní využití nacházejí kolekce při předávání dat mezi OR SŘBD a klientským programem. S jejich pomocí je možné snížit objem všech předávaných dat. Kolekce zajistí, že se jako výsledek dotazu nevrátí množina řádků obsahujících v některých svých sloupcích stejné informace, ale pouze jeden řádek, u kterého bude stejná informace uvedena jenom jednou a jeho další složkou bude množina hodnot obsahující jednu hodnotu za každý řádek zdrojové tabulky.

Posledním ze zmíněných typů je *odkaz*, který reprezentuje adresu instance typu řádek či kolekce. Odkazy lze použít jako typ libovolného sloupce tabulky.
Kromě výše uvedených existují ještě další dva konstrukty, které v OR SŘBD přicházejí v úvahu – *časové řady* a *multidimenzionální data*. Jejich využitelnost je sice omezenější, přesto ale jejich dostupnost umocňuje funkcionalitu systému řízení databáze.
Časové řady jsou nejčastější ve finančních resp. vědecko-technických aplikacích. Slouží k shromažďování dat ve stanovených časových intervalech. Takovéto aplikace časem nashromáždí velké množství dat.
Zatímco při standardním přístupu se s daty získanými v každém časovém okamžiku pracuje jako s řádky tabulky, efektivnější přístup spočívá v zřetězení těchto časových okamžiků prostřednictvím konstruktoru časové řady. Data tak budou uložena do jediného spojitého bloku, čímž se zjednoduší práce funkcí pracujících s těmito daty.
K zodpovězení dotazů na časová data je nutno použít jak data časových řad tak „obyčejná data".
Existuje několik důvodů k tomu, aby bylo uživateli OR systému umožněno rozšířit tento systém o nově definované jednoduché a složené datové typy a funkce. Těmi nejdůležitějšími důvody jsou:

- přirozenost
- zapouzdření
- OID
- datové konverze a řazení

Přirozenost souvisí s chápáním datových typů z hlediska uživatele. Řádkové typy je možno vytvářet s atributy různých typů. Každý řádek tabulky je přirozeně instancí typu řádek.
Zabýváme-li se otázkou *zapouzdření*, je nutno říci, že základní typy jsou zcela zapouzdřené.To znamená, že je lze používat k předávání argumentů při volání funkcí. Objekty typu řádek se naproti tomu skládají z několika složek, které jsou všechny dostupné dotazovacímu jazyku. Zveřejnění jen některých polí objektu typu řádek a ostatní ponechat soukromé představuje třetí možnost zapouzdření.
Každému složenému objektu je pro jeho jednoznačnou identifikaci přiděleno tzv. *OID*, což s sebou přináší určitou režii. Tento odkaz identifikující objekt jednoduché datové typy nemají.
Pro každý nově definovaný datový typ lze definovat operátory, které s nim budou umět pracovat. S jejich pomocí je možné instance nových typů ukládat do indexových struktur.
Některé nově definované jednoduché datové typy vyžadují odlišný vnitřní a vnější formát. Konverze těchto formátů provádějí funkce přetypování při vstupu a výstupu jednoduchých datových typů. Tyto konverze a uživatelsky definované operátory představují velmi mocný rys jednoduchých typů.

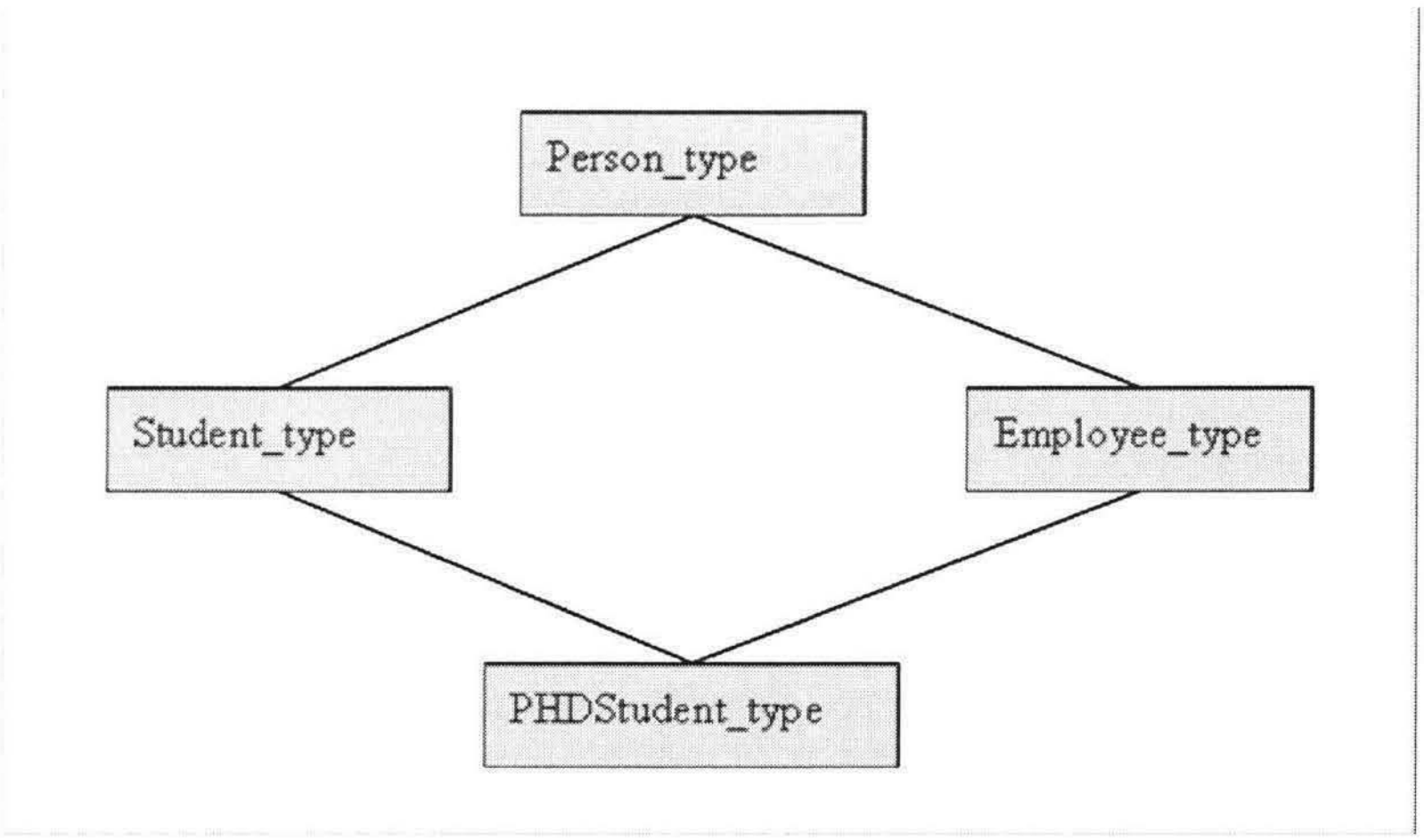
4.5 Dědičnost

Podpora dědičnosti je dalším nezbytným předpokladem dobrého OO SŘBD. Umožňuje uživateli, podobně jako první dvě charakteristiky, definovat nové datové typy. Podpora dědičnosti od rodiče k potomku se vztahuje pouze na složené datové typy. Při dědění zdědí potomek od rodiče všechna jeho datová pole a může k nim přidat své vlastní atributy. Složené typy tak lze sdružovat do hierarchie dědičnosti.
V některých případech se s výhodou využívá tzv. vícenásobná dědičnost, která umožňuje potomkovi dědit datové prvky od více rodičů. Přitom obsahují-li oba rodiče stejný atribut, zděděný od jejich společného rodiče, je tento atribut zděděn od tohoto předka. V případě, kdy potomek zdědil svá pole od více rodičů, může dojít k nejednoznačné situaci. Ta nastává, když oba rodiče obsahují pole stejného jména, ale mající pro každého rodiče jiný význam – tzv. diamond problem [10]. Tento problém lze řešit dvěma způsoby. První spočívá v tom, že administrátor nadefinuje jistá pravidla, která budou nejednoznačnost řešit. Jedná se ale o

poměrně složité řešení. Jiným a jednodušším řešením je prosté zakázání definice potomka způsobujícího nejednoznačnost. Neboli je nezbytné zajistit změnu definice problémových polí od rodičů.
K využití dědičnosti při definování tabulek je nezbytné nejprve vytvořit typy, které jsou následně přiřazovány novým tabulkám. Na základě této dědičnosti je pak možné vytvářet tzv. *tabulkové hierarchie*. V nich se pak, při dotazu na rodičovskou tabulku první úrovně, prohledávají všechny tabulky ležící pod ní. Tato hierarchie tedy umožňuje určovat obor hodnot příkazů SQL tak, aby se jediným příkazem SQL vybíraly výsledky z tabulky i všech jejich následníků. Samozřejmě je možno zajistit, aby se v případě potřeby prohledávala jenom tabulka nejvyšší úrovně.
S dědičností souvisí i následující detail. V případě, kdy se SQL dotaz ptá na všechny atributy nějaké tabulky, která je rodičovskou tabulkou tabulky jiné, s největší pravděpodobností jsou množiny atributů obou tabulek různé. Vyvstává tedy problém, které atributy má dotaz vrátit. Logicky by se mohlo zdát, že by měl vybírat jenom sloupce obsažené v nadřazené tabulce. Musí být ale zajištěna i možnost, kdy by vybíral všechny atributy obou tabulek.

4.5.1 Dědičnost metod

V případě dotazu na instance typu T, využívajícího definovanou metodu tohoto objektového typu, a zúženého jen na záznamy odpovídající tabulky, se dotaz vyhodnotí očekávaným způsobem. Co se ale stane v případě, kdy budou oborem hodnot dotazu i všechny zděděné tabulky? V takovém případě lze nově definovanou funkci s jistotou vyčíslit pro tabulku první úrovně. Problém nastává, když tato nová funkce není definována i pro odvozené tabulky. Řešení spočívá v zdědění této funkce od rodiče, které zajišťuje OR systém. Přitom je možné nadefinovat pro každý typ vlastní metodu stejného jména, jakou má definovanou rodičovský typ. Poté má tato metoda jinou implementaci než ta původní. Tzn. je přípustné předefinovat jednu metodu více různými způsoby. V takovém případě budou existovat dotazy, u kterých bude nutné použít v rámci jednoho dotazu více funkcí.
OR SŘBD obdrží žádost o vyčíslení metody, která není na dané úrovni hierarchie definována, je prohledána hierarchie typů od skutečného typu instance směrem ke kořeni hierarchie, a hledá se místo, ve kterém je požadovaná metoda definována. V takovém případě je metoda zavolána virtuálně se zajištěním polymorfního chování objektů. Pokud by hledání začínalo nikoli u skutečného typu instance, ale u typu T, volala by se pro všechny potomky původní varianta metody nevirtuálně.



Obr. 4.1 Příklad hierarchie s vícenásobnou dědičností

Při použití vícenásobné dědičnosti, jako je tomu na předchozím obrázku 4.1, by ale mohla nastat situace, kdy potomek třídy PHDStudent_type zdědí dvě různé implementace metody od svých dvou rodičů Student_type a Employee_type. V takovém případě nelze jednoznačně určit, která metoda se má použít. Tuto nejednoznačnost pak musí odstranit sám uživatel, a to buď přejmenováním jedné z metod, nebo napsáním další implementace funkce, řešící případ pro Ph.D. studenty.

4.6 Pravidla

Prostředkem zajišťujícím integritu dat uložených v databázi a jejich údržbu jsou tzv. pravidla. Dají se s výhodou používat i při modelování toku práce při tvorbě nové databázové aplikace. Oproti tradičnímu přístupu, u kterého byla pravidla realizována prostřednictvím triggerů mají několik výhod.
Pravidlo má následující obecný tvar:

při <události>
kde <podmínka>
proveď <akci>

V závislosti na typu události a provedené akci rozlišujeme čtyři druhy pravidel:

3. pravidla aktualizace-aktualizace
4. pravidla dotaz-aktualizace
5. pravidla aktualizace-dotaz
6. pravidla dotaz-dotaz

Akci by mělo být možno vyvolat jak před tak po zpracování události.
U prvního typu pravidel je jak událostí tak akcí provedení jisté aktualizace. Tzn. je-li definováno pravidlo tohoto druhu a nastala první aktualizace některého ze záznamů v tabulce, je automaticky provedena i druhá aktualizace měnící obsah jiného záznamu.
Druhým typem jsou pravidla dotaz-aktualizace. Tzn. událostí, která pravidlo vyvolá je dotaz a následně provedená akce je aktualizace v pravidle určeného záznamu.

U pravidel typu aktualizace-dotaz je situace obracená než v předchozím případě. Událostí je zde aktualizace a akcí je odezva vygenerovaná uživateli. Pro pravidla tohoto typu je někdy používáno označení *varování*. Varování se tedy používají k hlídání změn prováděných na záznamech tabulky a provádí akci, kterou tvoří upozornění pro rutinu výstrahy.
Posledním typem pravidel jsou pravidla, u kterých je jak událostí tak akcí, která se má provést výběr. V některých případech je možno pomocí tohoto typu pravidla nahradit pravidlo typu aktualizace-aktualizace.

Pravidla s sebou ale přinášejí i několik problémů, které je dobré mít na paměti při jejich používání. Nejčastějšími jsou [1]:

- více pravidel spuštěných stejnou událostí občas způsobuje nepředvídatelné výsledky
- zřetězená pravidla mohou vyvolat nekonečný cyklus
- ukončení transakce předčasně ukončí také akci tvořící součást pravidla
- načasování aktivace pravidla může mít vliv na konečný stav databáze

Počet definovaných pravidel není nijak omezen. Může tak nastat situace, kdy je pro jednu událost definováno více pravidel. Přitom pořadí, ve kterém budou tyto pravidla vztahující se ke stejné události prováděné, nelze ovlivnit a určuje ho systém. Tedy výsledky takových pravidel jsou zcela nepředvídatelné.
Při používání pravidel může nastat situace, kdy je akcí jednoho pravidla vyvolané jiné pravidlo. Takové situace jsou označovány jako *zřetězená pravidla*. Tímto způsobem je možno vytvořit nekonečnou smyčku zřetězených pravidel. Takové situace musí být systém schopný rozpoznat a vyřešit. Samozřejmě nejlepším řešením je vyhnout se podobným pravidlům.
K vyvolání akce pravidla může dojít i uvnitř právě prováděné transakce. Tato akce se pak vykonává v aktuální transakci. Problém nastává v případě zrušení transakce. V takovém případě dojde i k přerušení akce pravidla. Existují případy, ve kterých je tento postup nežádoucí – např. přerušení transakce dotazující se na některý záznam tabulky může znemožnit uložení položky do žurnálovacího souboru, ve kterém se ukládají všechny přístupy k záznamům tabulky. K zajištění správné funkčnosti operace je nezbytné zajistit, aby se akce pravidla provedla v samostatné transakci, která je pak nezávislá na původní uživatelské transakci.
U každého pravidla je navíc nutno určit, ve kterém okamžiku se akce pravidla provede. Tedy jestli se provede bezprostředně před nebo po zpracování události – *okamžité vykonávání*, resp. v momentu kdy je potvrzena aktivující transakce – *odložené vykonávání*. K získání správného výsledku se musí některá pravidla vykonávat okamžitě, některá na konci transakce. Tzn., že z hlediska konečného stavu databáze po skončení transakce je časový okamžik, ve kterém se pravidlo aktivuje, významný.

4.7 Relační optimalizátory

Při zadání jakéhokoliv dotazu je spuštěn optimalizátor, který na jeho základě vygeneruje množinu všech možných způsobů vykonání tohoto dotazu. Ne každý plán vyhodnocení je ale stejně dobrý. K výběru nejlepšího z nich používá optimalizátor následující nákladovou funkci odhadující celkové náklady potřebné ke zpracování dotazu [1]:

náklady=očekávaný počet prohlížených záznamů +
(opravný faktor*(předpokládaný počet přečtených stránek))

Očekávaný počet prohlížených záznamů zde zastupuje předpokládané využití zdrojů CPU v dotazu. Odhad nároků na vstupní a výstupní funkce vyjadřuje druhý člen, který je násobený opravným faktorem, který zastupuje důležitost zdroje CPU v porovnání se vstupně-výstupními zdroji. Na základě získaných hodnot pro všechny uvažované plány zpracování pak

optimalizátor vybere ten s nejnižšími náklady. K minimalizaci počtu zkoumaných plánů slouží různé vnitřní heuristiky používané optimalizátorem.
V každém dotazu jsou pro tabulku obsažena jistá omezení. Ty se dají realizovat více způsoby, z kterých je dobré vybrat ten nejvhodnější.
Implementačně nejjednodušší způsob je *sekvenční prohlížení tabulky* a vylučování všech záznamů nevyhovujících omezením z dotazu. Náklady spojené s touto operací jsou:

náklady=počet_záznamů + opravný faktor*počet_stránek

kde počet_záznamů je celkový počet záznamů v tabulce a počet_stránek je počet stránek, ve kterých jsou tyto záznamy uloženy.
K prohlížení tabulky je možno dále použít *index na bázi B-stromu* nad odpovídajícím sloupcem, využívající indexsekvenční přístup. Vkládání, rušení a správné seřazení záznamů v B-stromu zajišťují speciální rutiny.
Existují zde uzly dvou typů: vnitřní a listové. Listové uzly jsou ukládány ve tvaru:

(indexovaná hodnota, ukazatel na záznam v tabulce)

Vnitřní uzly obsahují záznamy ve tvaru:

(hodnota, ukazatel na následující stránku)

Vyhledávání konkrétní hodnoty v takové struktuře probíhá standardním způsobem, kdy je potřeba nejdřív vyhledat v kořenovém uzlu nejmenší hodnotu, která je větší než hledaná hodnota. Pak probíhá vyhledávání na uzlech nižší úrovně.
Kromě přesně dané hodnoty je možně provádět i intervalové vyhledávání, ve kterém se pro vyhledání počáteční hodnoty intervalu použije stejný postup jako při vyhledávání jedné hodnoty a od tohoto listu pak prohlížet další záznamy na listové úrovni indexu ležící uvnitř daného intervalu.
V tomto případě vyhledávání, musí optimalizátor odhadnout počet záznamů spadajících do hledaného intervalu a ke každému takovému záznamu vyhodnotit zbývající podmínky dotazu. Optimalizátor zde využije odhad založený na statistické evidenci, přičemž celkové vstupní a výstupní náklady závisí i na tom, zda je index seskupený resp. neseskupený.
K vyšší efektivnosti vyhledání dotazovaných záznamů může přispět i sestavení plánu dotazu využívajícího současně indexy nad více sloupci. Požadované záznamy se pak získají průnikem seznamů dílčích vyhledávání.

4.7.1 Spojení tabulek

Pracujeme-li najednou se dvěma tabulkami **Tab1** a **Tab2**, u kterých chceme provést jejich spojení, máme k dispozici tři různé postupy. U obou tabulek musí být znám očekávaný počet záznamů a použitých stránek.

1) vnořené cyklické spojení
2) spojení zatříděním
3) hašované spojení

Při použití první techniky se zvolí jedna z tabulek. Její záznamy jsou pak procházeny v cyklu a dosazovány do dotazu.
Náklady spojené s touto realizací spojení jsou:

náklady=počet_Tab1 + opravný faktor*stránky_Tab1
+
počet_Tab1*(očekávané náklady výsledného dotazu nad Tab2)

Očekávané náklady dotazu nad druhou tabulkou určí optimalizátor z množiny dotazů nad touto tabulkou.
U druhé varianty – *spojení zatříděním* - se nejprve provede setřídění tabulek **Tab1** a **Tab2** podle pole použitého ke spojení. Pak je možné obě tabulky zatřídit a provést jejich spojení. Tato technika je vhodná zejména tam, kde je už alespoň jedna tabulka setříděná.
Náklady pro tento případ spojení pak jsou:

náklady=náklady(třídění Tab1) + náklady(třídění Tab2) + náklady(zatřídění)

Výběr třídícího algoritmu může náklady spojené s tříděním obou tabulek výrazně ovlivnit.

Náklady spojené se zatříděním tabulek jsou:

náklady=počet_Tab1 + počet_Tab2 + opravný faktor*(stránky_Tab1 + stránky_Tab2)

Poslední možností používanou ke spojení dvou tabulek je *hašované spojení* [1], kdy je zvolena jedna tabulka, která je následně rozdělena do **H** hašovacích segmentů. Záznamy z druhé tabulky jsou pak sekvenčně vybírány pro každý záznam hašovaný podle spojovacího sloupce do segmentu. V hašovacím segmentu pak probíhá hledání záznamu se stejnými hodnotami.
Tento typ spojení se podobně jako předchozí typ spojení zatříděním dá použít jen u spojení založených na rovnosti. V opačném případě je možné použít pouze spojení vnořeným cyklem
Náklady u tohoto typu spojení jsou:

náklady=počet_Tab1 + počet_Tab2*(počet záznamů v hašovacím segmentu)

Náklady spojené se vstupem a výstupem se rovnají celkové velikosti obou tabulek:

náklady=stránky_Tab1 + stránky_Tab2

Náklady na vstup a výstup hašovaného spojení:

náklady=(stránky_Tab1*stránky_Tab2) / H

4.8 Objektově-relační optimalizátory

OR systémy obsahují naproti relačním systémům řadu rozšíření. Aby dokázal tradiční relační optimalizátor pracovat v novém prostředí, je nezbytné v něm provést jistá rozšíření. Jsou to následující:

1) **zápis příkazu v notaci používající jak operátory, tak funkce**
   Při zadávání dotazu v OR SŘBD lze omezující podmínku vyjádřit dvěma způsoby. A to buď pomocí operátoru nebo ve tvaru volání funkce. Oba dotazy dávají naprosto stejný výsledek a v obou případech by měl být vygenerován stejný plán dotazu proveden stejnou rychlostí.
2) **generické B-stromy**
3) **uživatelsky definované logické operátory**
   Generické B-stromy a uživatelsky definované logické operátory spolu souvisí. V případě dotazu využívajícího uživatelsky definovaný operátor (viz metody MAP a ORDER v kap. 5.3), jehož argument je uživatelsky definovaného typu, by mohl optimalizátor postupovat dvěma způsoby. Za prvé by mohl k odpovědi použít sekvenční prohlížení. Rychlejším způsobem, který povede k efektivnějšímu plánu dotazu, je použití indexu na bázi B-stromu. K sestavení takového indexu jsou ale nutné dvě rozšíření:

- Přepracovat B-stromy na generické, tedy takové, které je možné vytvářet jak nad standardními číselnými a řetězcovými datovými typy, tak i nad jakýmkoli jiným datovým typem.
- Možnost vytvářet uživatelsky definované relační operátory

  K tomu, aby bylo možné u takového dotazu provádět indexové prohledávání je nutné zajistit kromě standardních alfanumerických logických operátorů i podporu uživatelsky definovaných logických operátorů pracujících s nově definovanými uživatelskými typy.

4) **uživatelsky definované funkce selektivity**
   Ve spojení s možností umožnit uživateli definovat nové datové typy a operátory vyvstává nový problém. Optimalizátor totiž nemá informace o selektivitě těchto nových operátorů. Při tvorbě indexu na bázi B-stromu jsou využívány operátory, z kterých každému odpovídá určitá binární funkce, tzv. *funkce selektivity*. Pro každou nově vytvářenou funkci je možné definovat funkci tohoto typu. Optimalizátor pak s jejich pomocí vypočítává selektivitu operací nezbytnou k řádnému vyhodnocení plánu dotazu.

5) **uživatelsky definované operátory negace**
   K tomu, aby mohl být dotaz vyhodnocen indexovým prohlížením i v případě zadání operátoru negace *not* před uživatelsky definovaným operátorem, je nezbytné s každým takto nově definovaným operátorem uvést i k němu náležící operátor negace, aby mohl optimalizátor změnit původní dotaz na dotaz využívající přímo negovaného operátoru.

6) **uživatelsky definované komutativní operátory**
   Ke každému operátoru je možné v univerzálním serveru specifikovat komutativní operátor, přičemž rychlost jejich zpracování by měla být naprosto stejná.

7) **přístupové metody pro funkce vracející data**
   Aby bylo možno při zpracování dotazu využít indexové prohlížení i v případě, kdy dotaz obsahuje uživatelsky definovanou funkci, musí OR systém podporovat i indexy tvořené nad těmito funkcemi.

8) **inteligentní řazení klauzulí v predikátu**
   Není-li nad tabulkou vytvořen žádný index nezbývá nic jiného, než při zadání dotazu použít sekvenční prohlížení záznamů. Tento postup používá relační SŘBD, protože lze říci, že všechny klauzule jsou poměrně jednoduché. Tento postup ale není možno použít v OR SŘBD, kde se mohou v dotazu nacházet i uživatelsky definované funkce, jejichž vyhodnocení vyžaduje obecně více instrukcí. OR optimalizátor tedy musí být schopen rozlišit různé řazení klauzulí predikátu, aby se nejdříve vyhodnocovaly klauzule méně náročné, čímž se zmenší množina přípustných záznamů a následně byly vyhodnoceny náročnější klauzule.
   Při určování očekávaného počtu načítaných stránek lze postupovat dvěma způsoby. Tradiční optimalizátor předpokládá, že se každý záznam bude načítat jako celek. Existuje ale lepší řešení zajišťující snížení požadavků funkce na vstup a výstup. Místo čtení celých záznamů stačí, aby funkce přečetla pouze nějakou přídavnou informaci o daném záznamu, v případě, že taková existuje.
   OR optimalizátor musí mít k dispozici propracovanou nákladovou funkci, aby měl s její pomocí přehled o požadavcích funkcí majících vyšší nároky na zdroje CPU či na vstup resp. výstup. Při specifikaci takových funkcí se musí brát v úvahu:

- náklady CPU na volání funkce
- očekávané procento bytů argumentu, které funkce bude číst
- náklady CPU na jeden přečtený byte

Pro každou permutaci klauzulí predikátu dotazu vypočítá optimalizátor náklady.

9) optimalizace nákladných funkcí

Vyskytuje-li se v dotazu spojení realizované pomocí standardních operátorů a navíc dotaz obsahuje alespoň jednu nákladovou klauzuli, je vhodnější provést nejdřív spojení. Nákladové funkce pak stačí vyhodnotit pro menší počet přípustných záznamů.

10) uživatelsky definované přístupové metody

U dotazů požadujících dvou a vícedimenzionální vyhledávání je jednodimenzionální přístupová metoda bezcenná. V takovém případě je zapotřebí použít multidimenzionální přístupovou metodu, např. R-strom.

Řada aplikací z různých oborů (zdravotnictví, policie) vyžaduje vlastní speciální přístupové metody urychlující vyhledávání. Tzn. vyžadují přístupové metody, které vyhovují datovému typu, se kterým tyto aplikace pracují. Vzhledem ke skutečnosti, že OR systém umožňuje přidávat nové datové typy, musí zde existovat možnost přidat pro tyto nové typy vhodnou přístupovou metodu.

Přístupová metoda [1] je sada funkcí, které exekutivní stroj volá na příslušných místech během vykonávání plánu dotazu. Jejich úkolem je např. vykonávat:

- zahájení prohledávání indexu
- dodání dalšího prohlíženého záznamu
- vložení záznamu
- nahrazení záznamu
- uzavření prohlížení

Při tvorbě přístupové metody je nezbytné mít na paměti její schopnost zpracovat následující problémy:

- *Zamykání.* V některých situacích je nezbytné umístit na indexové objekty zámky. Po provedení příslušné akce je nutno následně tyto zámky odstranit. To vyžaduje spolupráci s modulem zajišťujícím v SŘBD správu zamykání.
- *Zotavení.* V případě havárie musí být přístupová metoda schopna zajistit zotavení používaných datových struktur. Toho lze dosáhnout několika způsoby. První možnost spočívá v provádění žurnálování všech událostí, které se týkají indexování. Jinou možností je provést zakódování programu přístupové metody. Někdy se s výhodou používá kombinace obou metod.
- *Správa stránkování.* Zde je nezbytná spolupráce s manažerem vyrovnávacích pamětí SŘBD. Pečlivá manipulace se stránkami ve vyrovnávací paměti umožňuje přístupové metodě využívat stránky ze společné oblasti této paměti. Tím se také předejde nutnosti kopírovat stránky do odděleného prostoru.

11) „uhlazení“ dotazů vykonávaných nad složitými objekty

Definujeme-li nějakou funkci, která jako svůj výsledek vrací množinu záznamů, např. v jazyce C, je ji možno použít v SQL dotazu na místech, kde je přípustné uvést tabulku, např. v klauzuli *from*. Při zpracovávání funkcí tohoto druhu nemá optimalizátor jinou volbu, než materializovat jejich výsledek. Navíc, protože jsou funkce napsané v jazyku C neprůhledné, je možná jen minimální optimalizace.

Jinak je tomu při práci s funkcemi napsanými v SQL, do kterých může optimalizátor nahlédnout. Dotazy s těmito funkcemi pak může optimalizátor, všude tam, kde je to

možné, uhlazovat. Uhlazení může být provedeno např. indexovým prohlížením v případě existence indexu na bázi B-stromu pro některý atribut v dotazu.

12) reprezentace množiny „v řádku“

Obsahuje-li záznam tabulky atribut typu množina, přičemž se v dotazu ptáme na prvky tohoto atributu pro některý záznam určený hodnotou jiného atributu záznamu, je výhodné, jsou-li tyto prvky uloženy ve stejném záznamu jako hodnota určujícího atributu. V takovém případě je vyhledání o mnoho rychlejší než když jsou množiny uloženy v samostatném záznamu a přistupuje se k nim nepřímo. Při ukládání množin do samostatného prostoru až v případě přetečení znamená zvýšení výkonnosti oproti systémům ukládajícím množiny odděleně.

13) indexy nad atributy množin

Systém by měl podporovat možnost vytvoření indexu nad atributem množiny. V případě existence takového indexu se vyhneme sekvenčnímu čtení prvků množiny.

14) optimalizace prohlížení hierarchií dědičnosti

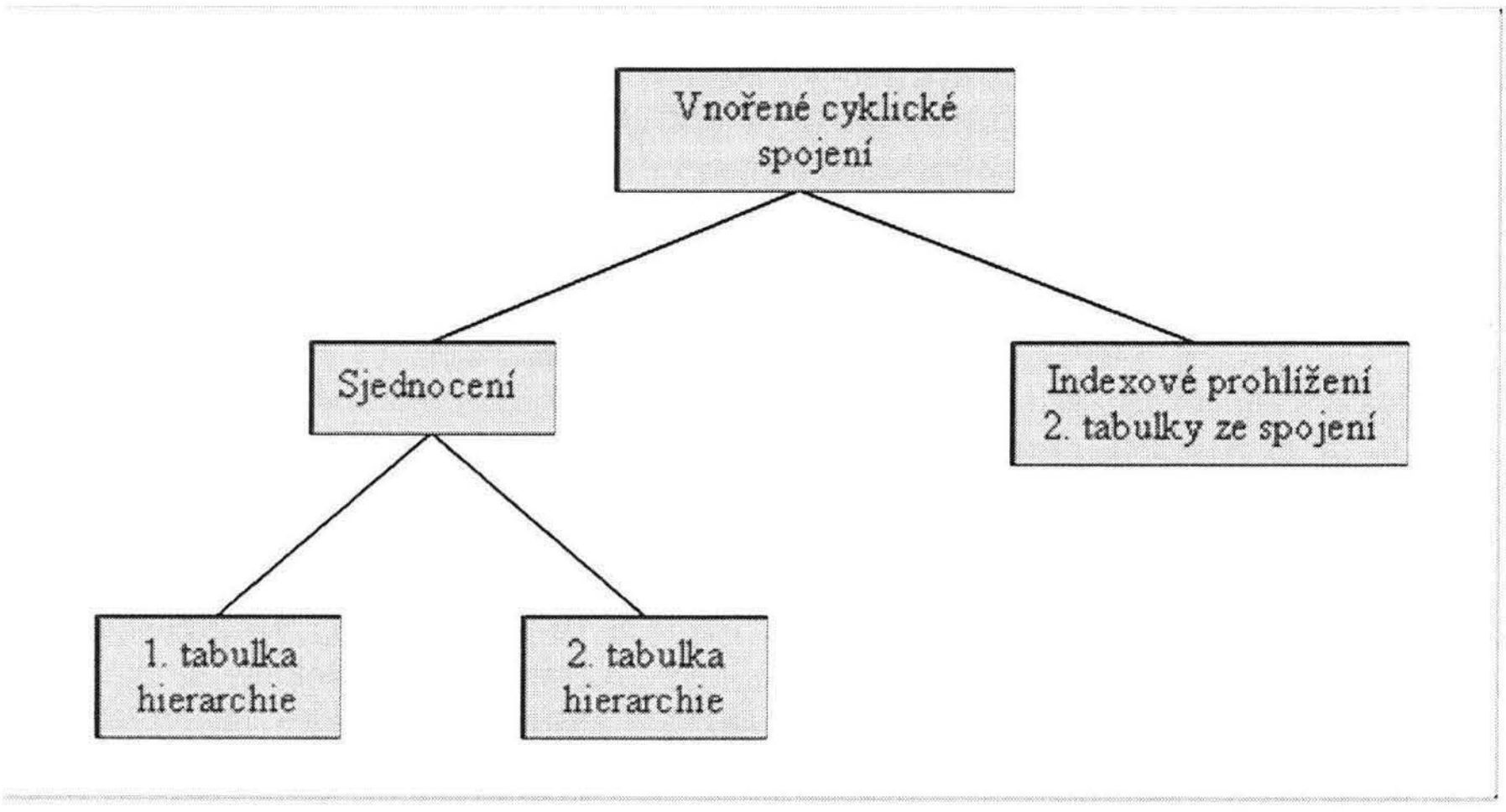
V případě existence tabulkové hierarchie se po zadání dotazu na tabulku nejvyšší úrovně náklady na vyřízení dotazu násobí. Jiným a rychlejším postupem je použít pro všechny tabulky v hierarchii jeden plán dotazu. Tímto způsobem se lze vyhnout režii spojené s dotazy vykonávanými navíc.

Jeden plán dotazu je ale nevhodný v případě existence indexu jedné tabulky hierarchie nad jedním ze sloupců a neexistence stejného indexu u zbývajících tabulek.

15) optimalizace operací spojení v hierarchiích dědičnosti

Obsahuje-li dotaz spojení dvou tabulek, přičemž jedna tabulka je součástí hierarchie dědičnosti tabulek, je tento dotaz při naivním přístupu zpracován tak, že se nahradí vykonáním více dotazů – pro každou tabulku z hierarchie jeden dotaz. Je-li nad druhou tabulkou ze spojení definován index, je možné, že optimalizátor provede indexové prohlížení této tabulky, postupně následované vnořeným cyklickým spojením vytvořeného výsledku s tabulkami z hierarchie. To ale znamená, že se totéž indexové prohlížení provádí vícekrát.

Efektivnější postup spočívá v tom, že se do stromové struktury plánu uloží pod spojení jediný uzel „sjednocené tabulky“, čímž se předejde duplikování práce.



Obr. 4.2 Použití sjednocené tabulky

16) podpora uživatelsky definovaných agregací

Aby mohl uživatel rozšířit systém o vlastní agregaci, musí systém pro každou novou agregaci, umožňovat specifikaci funkce pro inicializaci, iteraci a finalizaci. Velkým přínosem, se kterým je ale spojeno zvýšení složitosti, při zpracování agregací může být paralelismus. Ten ale nejde použít v každém případě. – příkladem by mohl být paralelní výpočet mediánu z několika množin, kdy se výsledný medián množiny dílčích mediánů nerovná mediánu celé množiny.

5 Objektově-relační rysy v Oracle

Porovnání relačního a OR přístupu je realizováno nad databází Oracle. Tato kapitola se proto věnuje popisu implementovaných OO rozšíření, které tato databáze nabízí, a které je možné při implementaci databáze použít.

5.1 Porovnání s obecnými požadavky

Systém Oracle postupně implementuje OR rysy obsažené v obecných požadavcích na OR systémy. I když je většina požadavků v Oracle již obsažena, pořád je zde možné najít rozdíly i když v některých jde jen o různý způsob zápisu stejné věci.
Tak je tomu např. u kolekcí, kde SQL standard uvádí možnost vytvářet kolekce typu pole položek ARRAY, seznam položek LIST resp. množinu položek SET. Oracle implementuje kolekce prostřednictvím typu pro pole VARRAY a hnízděných tabulek NESTED TABLE, které mohou být indexované nebo neindexované. Dále je zde možné nalézt rozdílnou syntaxi při vytváření nových typů a objektů.
Rozdíl lze nalézt i při tvorbě uživatelsky definovaných agregačních funkcí, kdy Oracle přidává navíc možnost provést spojení dvou agregací (viz odstavec v následující kapitole věnovaný agregacím).
Rozdíly v syntaxi je vidět i při definici nových objektových typů.

5.2 Nová funkcionalita

Vzhledem k tomu, že jedním z hlavních cílů nutně byla zpětná kompatibilita databáze se stávajícími relačními aplikacemi, je OR systém v Oracle implementovaný jako rozšíření relačního modelu a jeho předchozích procedurálních rozšíření. Důsledkem je ne vždy stoprocentní shoda syntaxe s ANSI SQL normou[2] a tím zapříčiněná nepřenositelnost aplikace na jiné databázové platformy.
Pokud porovnáváme nové OR rysy obsažené v systému Oracle 9i[3] s obecnými požadavky kapitoly 4, zjistíme, že ve větší či menší míře implementuje většinu uvedených prvků. Mezi výjimky patří datový typ řádky. Způsob implementace rysů, podstatných pro implementaci, je uveden v následujících sekcích.

5.2.1 Uživatelsky definované objektové typy.

Tyto typy mohou být používány pro modelování entit reálného světa. Každý objektový typ může obsahovat jak atributy, které odrážejí strukturu dané entity, tak metody, implementující operace nad těmito atributy. Jako typy jednotlivých atributů mohou vystupovat jak vestavěné tak i dříve definované objektové typy. Instance objektových typů mohou být uchovány jako takzvané řádkové objekty v objektových tabulkách, resp. jako sloupcové objekty ve sloupcích běžných tabulek.
V práci se budeme držet, pokud to bude nezbytné, pojmů sloupcový resp. řádkový typ, zavedených v SQL:1999. Názvosloví Oracle v obou případech používá označení objektový typ.

Příklad vytvoření typu:

---

[2] Kompatibilita s normou se v jednotlivých verzích zvyšuje, zpravidla umožněním obou variant zápisu, ale k dosažení plné kompatibility bude zřejmě zapotřebí ještě dlouhá doba.
[3] Na této verzi byla provedena implementace a také testovány výsledky. V současnosti je k dispozici již i novější verze Oracle 10g.

```
CREATE TYPE AddrType AS OBJECT (
    street      CHAR(20),
    city        CHAR(20),
    state       CHAR(20),
    zip         NUMBER(5)
);

CREATE TYPE BarType AS OBJECT (
    name        CHAR(20),
    addr        AddrType
);
```

S každým novým UDT vytváří Oracle implicitně stejnojmennou metodu, tzv. konstruktor, jehož počet parametrů, jejich pořadí a typ jsou shodné s definicí datových položek typu. Konstruktory se používají pro vytváření instancí daného typu, potřebných při vkládaní, změně resp. mazání záznamů z tabulek založených na UDT. Tyto tabulky se odlišují od relačních tabulek v několika věcech:

- Každý řádek tabulky, tedy ne každá instance řádkového objektového typu, má vlastní OID – object identifier.
- Na tyto řádky je možné se odkazovat z jiných objektů v databázi prostřednictvím referencí.

Příklad vytvoření objektové tabulky:

```
CREATE TABLE Addrs OF AddrType;

CREATE TABLE Bars OF BarType;
```

Vložení nových záznamů do objektové tabulky se s využitím konstruktoru provede příkazem:

```
INSERT INTO Bars VALUES (
    BarType(
          'Novy bar',
          AddrType('Lazarska', 'Praha', 'CZ', 12000)
    )
);
```

Výběr záznamů z takových tabulek je o něco složitější. Zadáním jednoduchého dotazu

```
SELECT * FROM Bars;
```

získáme vícesloupcový výstup , obsahující případné konstruktory UDT. Např.

```
'Novy bar'  AddrType('Lazarska', 'Praha', 'CZ', 12000)
```

Chceme-li získat jednotlivé atomické hodnoty, je nutné použít tečkovou notaci pro přístup k jednotlivým položkám:

```
SELECT b.name, b.addr.street FROM Bars b;
```

Přitom je nutné použít při přístupu k atributům objektového typu alias, jinak daný dotaz nebude funkční.

5.2.2 Uživatelsky definované konstruktory.

Uživatelsky definované konstruktory jsou, stejně jako konstruktor implicitní, používány k vytváření instancí uživatelsky definovaných typů. Při změně definice nového datového typu (například změně počtu atributů, nebo jejich typů) dojde zároveň ke změně implicitního konstruktoru. Tzn., že všechna volání implicitních konstruktorů v existujícím kódu nepůjdou zkompilovat.
Řešení uvedeného problému spočívá v použití uživatelsky definovaných konstruktorů, které explicitně nepotřebují množinu hodnot pro všechny atributy. Mohou mít libovolný počet atributů libovolných typů. V definici konstruktoru je totiž možné inicializovat hodnoty

zbývajících atributů nějakou defaultní, či spočtenou hodnotou. Při změně datového typu není vždy nutné měnit volání uživatelsky definovaného konstruktoru. Často stačí změnit definici konstruktoru tak, aby zohlednil posledně udělané změny na atributech.
Konstruktor musí mít vždy stejný název, jako je název typu. Uživatelsky definovaný konstruktor přitom může být přetížen v závislosti na použitých atributech:

```
CREATE OR REPLACE TYPE shape AS OBJECT
(
    name  VARCHAR2(20),
    area  NUMBER,
    CONSTRUCTOR FUNCTION shape(
          name VARCHAR2
          ) RETURN SELF AS RESULT;
    CONSTRUCTOR FUNCTION shape(
          name VARCHAR2,
          area NUMBER
          ) RETURN SELF AS RESULT

) NOT FINAL;

CREATE OR REPLACE TYPE BODY shape IS
    CONSTRUCTOR FUNCTION shape(
          name VARCHAR2
          ) RETURN SELF AS RESULT
    IS
    BEGIN
          SELF.name := name;
          SELF.area  := 0;
          RETURN;
    END;
    CONSTRUCTOR FUNCTION shape(
          name VARCHAR2,
          area NUMBER
          ) RETURN SELF AS RESULT
    IS
    BEGIN
          SELF.name := name;
          SELF.area  := area;
          RETURN;
    END;
END;
```

Uživatelsky definované konstruktory lze volat stejným způsobem jako ostatní funkce.

5.2.3 Typová dědičnost.

Prostřednictvím objektových typů lze snadno modelovat entity reálného světa. Od verze 9 je možné vytvářet hierarchie typů s využitím jednoduché dědičnosti (viz kap. 4.5). Potomek zdědí všechny atributy a metody od svého rodiče, přičemž se na něm automaticky provádějí i všechny změny prováděné na rodiči Potomek přitom může předefinovat nebo přetížit metodu stejného jména resp. předefinovat zděděnou metodu. Stejně tak může přidat své nové metody nebo atributy.
Při vytváření objektových typů je možné určit, zda může mít daný typ potomky, či nikoli. To se specifikuje pomoci klíčových slov ***NOT FINAL*** při deklaraci typu. Implicitně je nastavena hodnota ***FINAL***. Tzn., že potomky není možné vytvářet. Změnu možnosti dědit lze provést i později pomoci příkazu ***ALTER TYPE***. Přitom změna z NOT FINAL na FINAL je možná jen v tom případě, že daný typ nemá z něj odvozené typy.
Podobně je to s metodami. I u nich lze explicitně uvést ***FINAL***, takže je nebude možné změnit v odvozených typech.

```
CREATE TYPE Student_typ UNDER Person_type(
    peptid      NUMBER,
    major       VARCHAR2(30)
) FINAL;
```

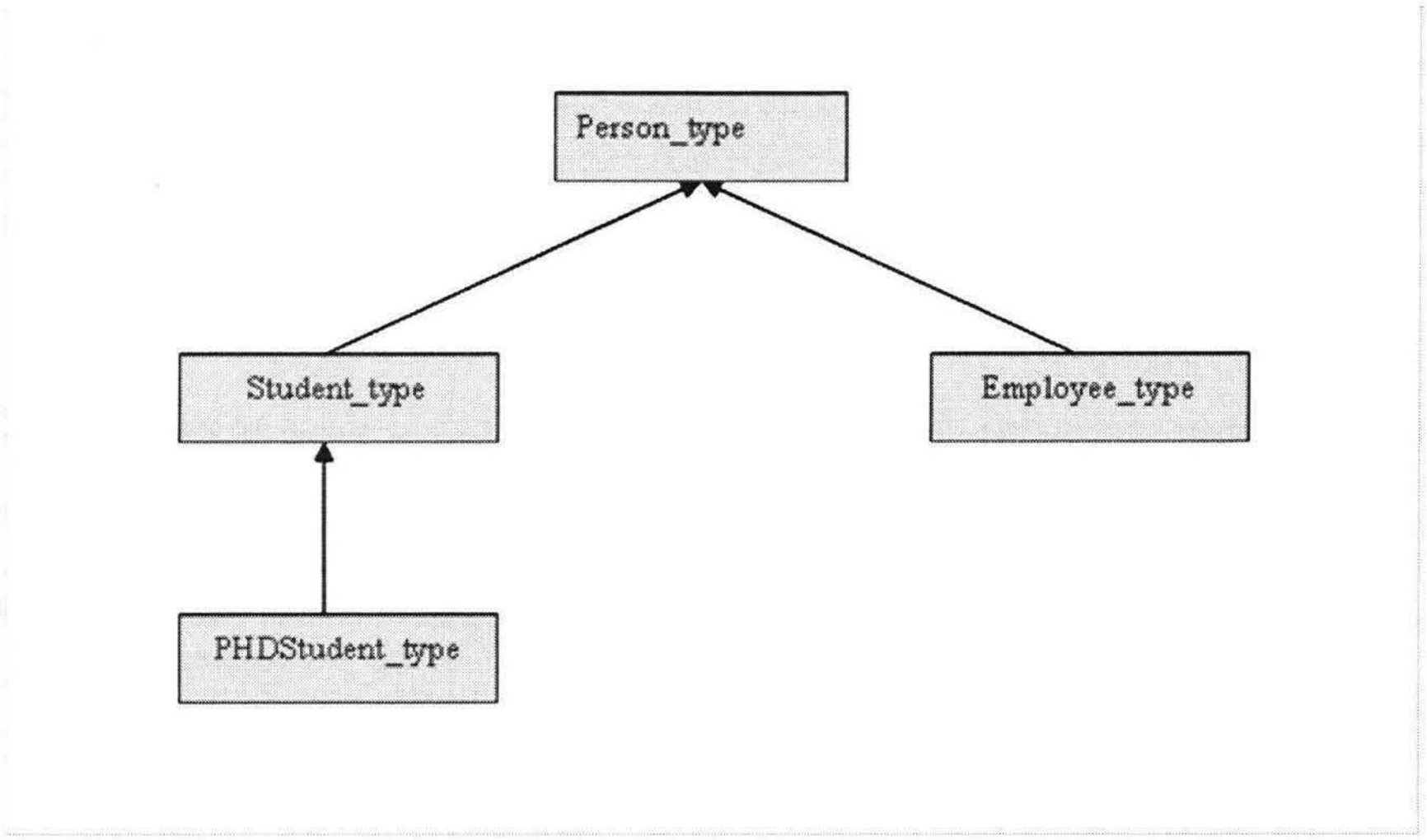
Objektový model Oracle podporuje rovněž tvorbu tzv. abstraktních typů, ze kterých není možné vytvářet instance. Tyto typy pak mohou sloužit jako společný předek pro nově odvozené typy.

```
CREATE TYPE Address_type AS OBJECT( ... ) NOT INSTANTIABLE
NOT FINAL;
```

Stejným způsobem mohou být deklarovány i metody. U takových metod se pak očekává, že budou definovány u každého potomka jiným způsobem.
V typové hierarchii jsou potomci různými variantami společného základního typu. Při práci se zděděnými typy je někdy vhodnější pracovat na obecnější úrovni. Příkladem může být dotaz na výběr všech osob včetně odvozených typů pro studenty a zaměstnance. Schopnost výběru všech základních objektů včetně od nich odvozených objektů se nazývá **nahraditelnost**. Ve všeobecnosti jsou typy nahraditelné, protože potomci jsou specializovaným druhem základních typů. Např. do sloupce pro osoby je možné uložit i všechny osoby odvozené ze základního typu jako jsou studenti resp. zaměstnanci.

5.2.4 Hierarchie objektových pohledů.

Objektové pohledy jsou užitečné při přechodu na OO aplikace, protože data mohou být do pohledu brány z relačních tabulek a následně k nim může být přistupováno jako k datům z objektových tabulek. Je tedy možné vytvářet OO aplikace bez nutnosti konverze existujících tabulek na jinou fyzickou strukturu.
Podobně jako klasické pohledy jsou virtuálními tabulkami, jsou objektové pohledy virtuálními objektovými tabulkami. Každý řádek pohledu je objekt, tzn. je možné volat jeho metody resp. přistupovat k jeho atributům.
Objektové pohledy mají podobnou funkcionalitu jako objektové tabulky. Mohou mít svoje metody, být součástí kolekcí, mít objektovou identitu a s její pomocí se navzájem referencovat s jinými pohledy.
Podobně jako klasické pohledy, mohou být i objektové pohledy použité k prezentaci jenom určitého druhu dat vhodných pro různé uživatele.
Objektový pohled může být vytvořen jako podpohled jiného objektového pohledu, čímž je vytvářena pohledová hierarchie.
Hierarchie objektových pohledů je množina objektových pohledů, z kterých každý je založen na jiném typu v typové hierarchii. Každý objektový pohled obsahuje objekty jednoduchých typů. Objektové pohledy dávají jednoduchý způsob tvorby dotazů, které mohou vracet polymorfní množiny objektů na dané úrovni. V kapitole obecných požadavků 4.5.1je uveden obdobný příklad typové dědičnosti metod, vyžívající zde zakázané vícenásobné dědičnosti.



Obr. 5.1 Hierarchie objektových pohledů

Je-li vytvořená hierarchie objektových pohledů na základě uvedené typové hierarchie pro každý typ, je možné dotazovat se nad objektovými pohledy korespondujícími s požadovanou úrovní specializace. Např. je možné dotazovat se nad pohledem, vytvořeným pro Student_type a získat výslednou množinu obsahující pouze studenty a Ph.D. studenty.
Kořenový pohled hierarchie je možné založit na kterémkoliv typu v hierarchii typů. Podpohled dědí OID od nadřízeného pohledu. Pro podpohled nemůže být proto OID explicitně specifikováno. Výhodou pohledové hierarchie je, že řádky pohledu obsahují všechny řádky odpovídajícího podpohledu. Tzn. je možné se dotazovat na objekty, jejichž typy patří do této hierarchie.
Vytvoříme-li dva objektové pohledy, kdy objektový typ Employee_type je odvozen od typu Person_type,

```
CREATE VIEW Persons OF Person_type WITH OBJECT ID (name) AS
   SELECT name FROM r_persons;

CREATE VIEW Employees OF Employee_type UNDER Person_type AS
   SELECT name, salary, bonus FROM r_employees;
```

potom následující dotaz nad pohledem Persons vrátí všechny osoby včetně zaměstnanců:

```
SELECT VALUE(p) FROM Persons p;
```

5.2.5 Evoluce datových typů.

Změna uživatelsky definovaných typů se označuje jako evoluce typů. Uživatelské typy je možné měnit několika způsoby

- přidávání resp. odebírání atributů
- přidávání resp. odebírání metod
- modifikace numerických atributů zvětšením jejich rozsahu resp. přesnosti
- modifikace proměnné délky znakových atributů
- změna nastavení FINAL resp. INSTANTIABLE

Objekty, které by přímo nebo nepřímo odkazovaly na typy ovlivněné změnami se označují **objekty závislými**.

5.2.6 Uživatelsky definované agregační funkce.

Systém Oracle poskytuje řadu předefinovaných agregačních funkcí jako jsou **MIN, MAX, SUM**, které mohou být použity pouze na skalární data (viz kap. 4.3). Pro použití těchto funkcí na složitější data je nutné tyto agregační funkce buď modifikovat resp. vytvořit zcela nové vlastní agregační funkce. Tyto funkce se používají v SQL DML výrazech stejným způsobem jako vestavěné agregační funkce.
Při tvorbě nových agregačních funkcí je nutné splnit následující kroky:

1. Inicializaci
2. Iteraci
3. Dokončení
4. Spojení

Tedy postup je naprosto stejný jako u obecných požadavků na OR systémy (viz kap. 4.3) s tím, že je zde navíc nepovinná možnost spojení. Je tedy možné v případě nutnosti spojit dva kontexty dvou agregací do jednoho.

5.2.7 Indexy založené na funkcích (funkční indexy).

Jedná se o indexy založené na návratových hodnotách výrazů resp. funkcí. Jako funkce mohou vystupovat i metody objektových typů. Funkční index založený na metodě objektu provede nejdřív předvýpočet návratové hodnoty funkce pro každou indexovanou instanci a tyto hodnoty uchová. Na tyto hodnoty je pak možné se v indexu odkazovat bez nutnosti opětovně vyhodnocovat danou funkci.
Funkční indexy jsou užitečné při zvyšování výkonu dotazů.

Příklad:

```
CREATE TYPE Employee_type UNDER Person_type
(
    name         VARCHAR2,
    salary       NUMBER,
    MEMBER FUNCTION bonus RETURN NUMBER DETERMINISTIC
);

CREATE OR REPLACE TYPE BODY Employee_type IS
    MEMBER FUNCTION bonus RETURN NUMBER IS
    BEGIN
          RETURN self.salary * 0.1;
    END;
END;

CREATE TABLE emps OF Employee_type;

SELECT name FROM emps e
WHERE e.bonus() > 2000;
```

Při vyhodnocování uvedeného dotazu musí systém vyhodnotit funkci **bonus()** pro každý řádkový objekt tabulky. Je-li ale použit funkční index založený na návratové hodnotě této funkce, pak se dotaz urychlí, protože stačí, když se podívá do odpovídajícího indexu, který tyto hodnoty již obsahuje.
Návratové hodnoty funkcí mohou být užitečně indexovány jenom v případě, jsou-li tyto hodnoty konstantami, tzn. je-li zaručeno, že funkce navrací stejné hodnoty pro opakovaná volání se stejnými hodnotami parametrů. Z toho důvodu musí být u deklarace uživatelsky definované funkce uvedeno klíčové slovo **DETERMINISTIC**.
Příklad vytvoření funkčního indexu metody bonus() v tabulce emps:

```
CREATE INDEX emps_bonus_idx ON emps x (x.bonus());
```

5.2.8 Kolekce, víceúrovňové kolekce.

Pod pojmem kolekce se rozumí uspořádaná skupina elementů stejného typu. Lze je použít např. při implementaci seznamů a polí, jejichž prvky jsou přístupný prostřednictvím indexu.
Uchovávání elementů v kolekcích přináší několik výhod. V první řade zjednodušují kód. Je-li nezbytné zpracovat větší množství elementů podobného typu, jejich uložením do kolekce získáme snadný přístup k jednotlivým elementům prostřednictvím jejich indexů. Jako příklad může posoužit metoda odstraňující všechny elementy kolekce prostřednictvím jednoduchého dotazu.
Pravděpodobně největší výhodou, kterou kolekce poskytují je zvýšení výkonu aplikace. Jsou s výhodou využívány pro ukládání statických dat s opakovaným přístupem, čímž se snižuje množství přístupu do vlastní databáze.

Systém Oracle poskytuje tři typy kolekci:

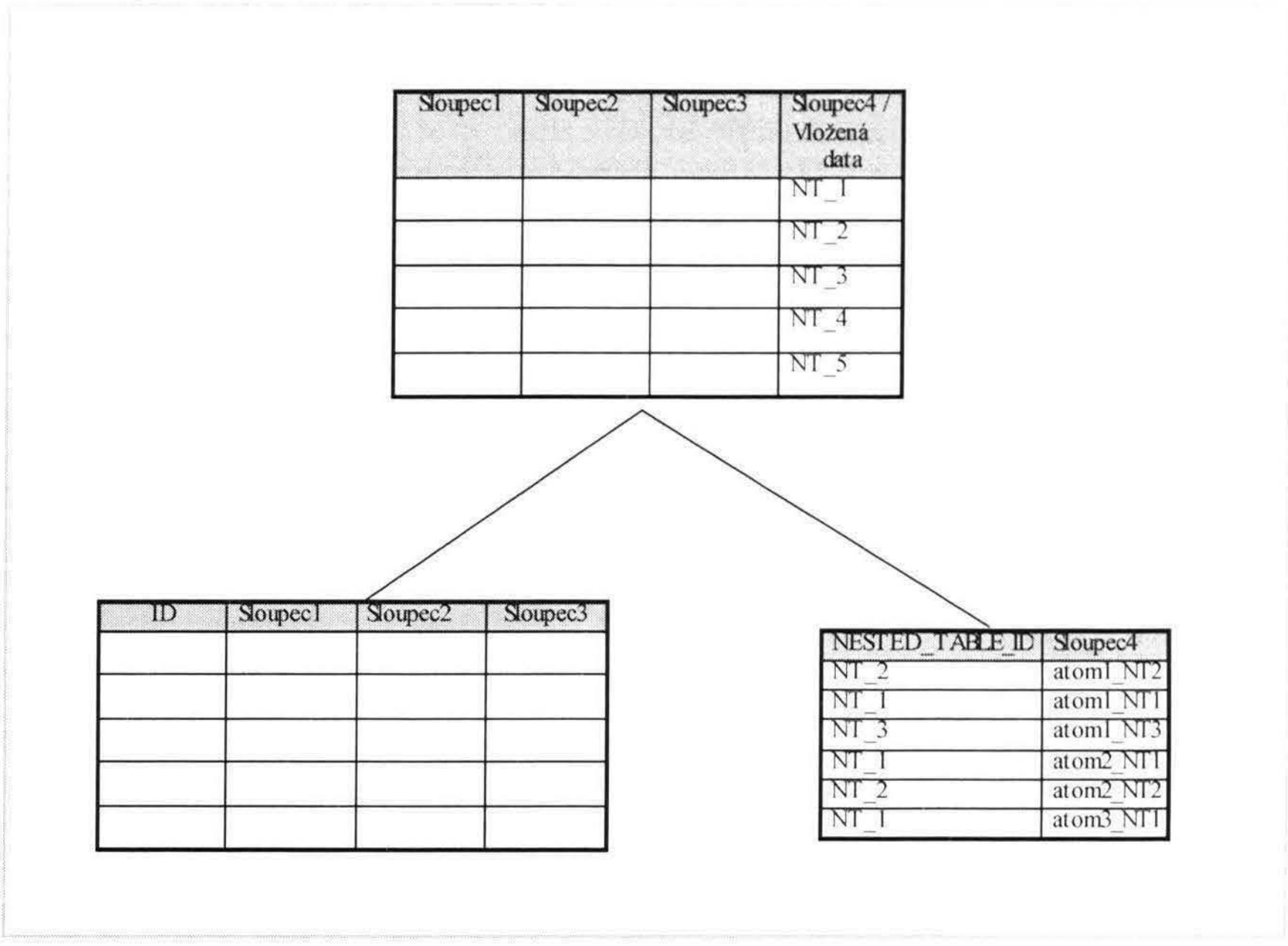
- hnízděné tabulky
- VARRAY
- asociativní pole

K modelování relací typu 1:N poskytuje Oracle dva datové typy kolekcí:
první je pole **VARRAY**, které je uspořádanou kolekcí elementů, každý s vlastní pozicí reprezentovanou indexem. Při definování pole je nutné specifikovat maximální počet elementu, které toto pole může obsahovat. Tuto velikost přitom lze později změnit. Velikost uloženého pole ale závisí pouze na aktuálním počtu prvků pole bez ohledu na jeho maximální deklarovanou velikost. Pole typu VARRAY je uloženo jako neprůhledný objekt.
Při změně resp. získávání jednotlivých elementů pole je nezbytné získat najednou všechny elementy, čímž je možné provádět požadovanou operaci se všemi prvky najednou. Tzn. pole je výhodné použít v případě, kdy hodláme manipulovat s kolekcí jako s celkem.
Druhým typem kolekcí jsou **hnízděné tabulky**.
Jedná se o podtabulky, které jsou vloženy do nadřazených tabulek na místech pro skalární hodnoty. Tyto tabulky mohou mít různý počet elementů, přičemž není určeno žádné omezující maximum. Podobně není určeno ani pořadí jednotlivých elementů. S hnízděnými tabulkami lze provádět selekci, vkládání resp. mazání stejným způsobem jako u obyčejných tabulek.
Následující výpis ukazuje vytvoření tabulky MESTO, obsahující jako sloupcové hodnoty hnízděné tabulky, uchovávající informace o obyvatelých jednotlivých měst:

```
CREATE TYPE otyp_obyvatele AS OBJECT(
    Jmeno VARCHAR2(10),
    Prijmeni    VARCHAR2(10),
    Ulice VARCHAR2(15)
);

CREATE TYPE typ_obyvatele AS TABLE OF otyp_obyvatele;

CREATE TYPE otyp_mesto AS OBJECT(
    Id    NUMBER,
    Obyvatele   typ_obyvatele,
    Nazev VARCHAR2(10)
);

CREATE TABLE mesto OF otyp_mesto(
    PRIMARY KEY (Id)
) NESTED TABLE Obyvatele STORE AS Obyvatele_nt;
```

Pro každý sloupec rodičovské tabulky, obsahující hnízděnou tabulku, vytváří systém Oracle tzv. úložnou tabulku k uchování záznamů hnízděných tabulek pro všechny řádky rodičovské

tabulky. V každé takové úložné tabulce navíc vytváří skrytý sloupec, nazývaný **NESTED_TABLE_ID**, který je používán k identifikaci záznamů hnízděné tabulky pro jednotlivé řádky rodičovské tabulky. Záznamy uchovávané v úložné tabulce mohou být indexovány.
Rozpad tabulky, obsahující sloupec typu hnízděná tabulka, na dvě ukazuje následující obrázek:



Obr. 5.2 Hnízdění tabulky

Vytvoříme-li nad úložnou tabulkou index začínající sloupcem NESTED_TABLE_ID, umožníme efektivní hledání řádek obsažených v hnízděné tabulce daného řádku.
K jednotlivým řádkům hnízděné tabulky je možno přistupovat prostřednictvím operátoru **TABLE**. V následujícím příkladu se ptáme na výskyt jména Novák v tabulce Obyvatele hnízděné do tabulky Mesto:

```
SELECT * FROM mesto m
WHERE  'Novák' IN
(SELECT prijmeni FROM TABLE(m.obyvatele));
```

Do hnízděných tabulek je možné samozřejmě přidávat nové záznamy, mazat je resp. modifikovat. I u dotazů tohoto typu, stejně tak jako u dalšího dotazu, se využívá operátoru TABLE. Vlastní vložení se provádí následujícím dotazem,

```
INSERT INTO mesto VALUES (
   1,
   typ_obyvatele(
         otyp_obyvatele('Jan', 'Novák', 'Sokolovská 21'),
         otyp_obyvatele('Petr', 'Malý', 'Letenská')
   ),
   'Praha');
```

kdy byli vloženi dva noví obyvatelé Prahy.
Chceme-li vkládat záznamy přímo do hnízděné tabulky, je nutné použít následující dotaz:

```
INSERT INTO TABLE
(SELECT m.obyvatele FROM mesto m WHERE m.id=1)
   VALUES ('Pavel', 'Vyskočil', 'J. Husa');
```

Chceme-li zrušit některou z hnízděných tabulek, stačí když nastavíme rodičovské tabulce v sloupci pro hnízděné tabulky hodnotu NULL. Od této chvíle není možné s hnízděnou tabulkou pracovat, dokud ji znovu nevytvoříme.
K znovuvytvoření hnízděné tabulky je nutné použít konstruktor jejího typu:

```
UPDATE mesto SET obyvatele = typ_obyvatele() WHERE id=1;
```

Při znovuvytvoření hnízděné tabulky je možné provést zároveň i vložení potřebných hodnot:

```
UPDATE mesto SET obyvatele =
typ_obyvatele(otyp_obyvatele('Karel', 'Vyskočil', 'Petrská
2')) WHERE id=1;
```

Při provádění operací nad hnízděnou tabulkou dochází k uzamčení rodičovského řádku, kterému tato tabulka patří. Proto je možné provádět najednou pouze jednu modifikaci nad hnízděnou tabulkou a to i v případě, kdy by se modifikace prováděla na různých řádcích této tabulky.
Simultánní modifikaci je možné provádět pouze v případech, kdy se tato modifikace týká jenom části hnízděné tabulky a to prostřednictvím operátorů REF. V takovém případě je nutné vytvořit samostatnou objektovou tabulku, ve které budou uloženy potřebné částečné informace a na které se bude odkazovat pomoci REF z hnízděné tabulky.

Ve všech dotazech tohoto odstavce byl použit TABLE operátor. Ten vrací jako svůj výsledek množinu, kterou je pak možné použít ve FROM klauzuli dotazu. Tento operátor zvyšuje výkonnost a flexibilitu napsaného kódu:

- může být volán při vícenásobném přístupu
- eliminuje potřebu uchovávat data v dočasných strukturách
- vrací získané řádky průběžně tzn. není nezbytné čekat na získání celé množiny

Od hnízděných tabulek jsou odvozeny tzv. **indexované tabulky** (asociativní pole), u kterých jsou jednotlivé elementy indexované jednoznačným klíčem (číslem resp. řetězcem). To umožňuje rychlejší přístup k elementům a to bez nutnosti znát vlastní pozici přistupovaného elementu resp. nutnosti sekvenčního procházení všech elementů.
U asociativních polí je dobré uvědomit si několik skutečností.
Tyto pole jsou určena pro uchovávání dočasných dat, tzn. není je možno použit při dotazech typu *INSERT* resp. *SELECT INTO*. Je tedy možné s nimi pracovat jako s perzistentními pouze v rámci databázového sezení a to přiřazením jejich deklarovaného typu nějaké lokální proměnné.
Na rozdíl od hnízděných tabulek je není možné uchovávat jako sloupec nějaké tabulky.
Asociativní pole jsou vhodné hlavně pro relativně malé vyhledávací tabulky, které se vejdou najednou celé do paměti.
Typy kolekcí lze použít kdekoliv kde můžou být použity ostatní datové typy. Je možné vytvářet objektové atributy typu kolekce stejně tak jako sloupce typu kolekce.
Pole typu VARRAY je vhodné použít v případě, kdy je nutné uchovat předem daný počet elementů včetně průchodu přes tyto elementy prostřednictvím cyklu.
Hnízděné tabulky jsou vhodné v případech, kdy je nezbytné provádět dobře fungující dotazy nad kolekcemi resp. provádět masové vkládaní resp. mazání elementů.
Víceúrovňové kolekce jsou kolekce, kterých jednotlivé elementy jsou taky kolekce. Mohou nastat následující možnosti:

- Hnízděná tabulka hnízděného tabulkového typu.
- Vnitřní množina hnízděných tabulek vyžaduje speciální tabulku, ve které budou uloženy sloupce všech hnízděných tabulek.
- Hnízděná tabulka typu VARRAY.
- Pole VARRAY hnízděných tabulkových typů.
- Pole je uloženo do LOBu, přičemž ve vlastním řádku je uložen pouze lokátor odkazující na místo uložení.
- Pole VARRAY typů VARRAY.
- V tomto případě je celé pole (nepřekročí-li jeho velikost danou hranici) uloženo v samotném řádku, jinak je uloženo do LOBu.
- Hnízděná tabulka resp. pole VARRAY uživatelsky definovaného typu s atributy hnízděné tabulky nebo pole VARRAY.

Podobně jako jednoduché, jednoúrovňové typy kolekcí i víceúrovňové kolekce mohou být použity jako typy sloupců v relačních tabulkách nebo jako objektové atributy v objektových tabulkách.
U kolekcí lze využít způsob jejich přiřazování. Kromě přiřazování hodnoty pro konkrétní element je¨rovněž možné hodnotu jedné kolekce přímo přiřadit do jiné za předpokladu, že jsou obě stejného typu. Tzn. shoda typů jednotlivých elementů přímému přiřazení nestačí. Na druhé straně není možné kolekce navzájem porovnávat na rovnost jinak než porovnáním všech jejich prvků postupně.
Inicializace kolekce je závislá na jejím typu. Zatímco jsou asociativní pole inicializovány hned při deklaraci, je nutné při hnízděných tabulkách a polích provést inicializaci buď explicitně pomocí konstruktoru nebo implicitně přiřazením hodnot z databáze resp. přiřazením hodnoty jiné kolekce.
Přidávání a ubírání elementů z kolekce je nejsnadnější u asociativních polí, kdy se stačí odkázat indexem na požadovaný element. U hnízděných tabulek a polí je nutné při přidávání použít funkci **EXTEND**, která vytvoří místo pro nový element.
Pro vymazání elementu z kolekce existují dvě funkce. První je funkce **DELETE**, pomocí které je možné vymazat prvek kolekce bez ohledu na jeho pozici. Druhou funkci je funkce **TRIM**, která odstraňuje elementy z konce kolekce. Funkce TRIM navíc na rozdíl od DELETE odstraňuje krom hodnoty i vlastní buňku.
Znepřístupnění obsahu celé kolekce se provádí přiřazením hodnoty NULL této kolekci. Následně je pro další práci tuto kolekci znovu inicializovat.

Pro lehčí manipulaci se všemi druhy kolekcí existuje řada vestavěných funkcí.

Metoda	Popis
funkce COUNT	Vrací aktuální počet elementů kolekce.
procedura DELETE[(i[,j])]	Odstraní z hnízděné tabulky resp. asociativního pole i-tý element resp. elementy počínaje i-tým a konče j-tým.DELETE bez parametrů odstraní všechny elementy. Tuto proceduru není možno použít u polí.
funkce EXISTS(i)	Vrací true resp. false v závislosti na existenci i-tého elementu v kolekci.
procedura EXTEND[(n[,i])]	Přidává n nových elementů do kolekce, přičemž je inicializuje hodnotou i-tého elementu. Defaultně je n=1.
funkce FIRST	Vrací nejmenší použitelný index kolekce.
funkce LAST	Vrací největší použitelný index kolekce.
funkce LIMIT	Vrací maximální počet přípustných elementů v poli.
funkce PRIOR(i)	Vrací index předchozí indexu i. Je-li i první index, vrací NULL.
funkce NEXT(i)	Vrací index následující za indexem i. Je-li i poslední index, vrací NULL.
procedura TRIM[(n)]	Odstraní n elementů z konce kolekce. Defaultní hodnota n=1. Není možno použít u asociativních polí.

Tab. 5.1 Funkce pro práci s kolekcemi.

Pro kolekce je definováno rovněž několik pseudofunkcí. Jedná se o funkce uvedené pod definici tabulek, na kterých bude ukázána jejich funkcionalita.

```
CREATE TYPE colors_tab_t IS TABLE OF VARCHAR2(32);

CREATE TABLE color_models (
    model_type  VARCHAR2(12),
    colors      colors_tab_t
) NESTED TABLE colors STORE AS  color_model_colors_tab;

CREATE TABLE birds (
    Name        VARCHAR2(40),
    Genus       VARCHAR2(30),
    species     VARCHAR2(30)
);

CREATE TABLE bird_habitats (
    city        VARCHAR2(30),
    country     VARCHAR2(30)
);
```

- **CAST**

Funkce CAST provádí mapování kolekce jednoho typu na kolekci jiného typu.

```
SELECT column_value
FROM TABLE(
          SELECT CAST(colors AS colors_tab_t)
          FROM color_models_a
          WHERE model_type = 'RGB');
```

- **MULTISET**

Funkce MULTISET mapuje databázovou tabulku do kolekce. Prostřednictvím funkcí MULTISET a CAST je možné získat řádek databázové tabulky jako sloupc typu kolekce:

```
SELECT b.genus, b.species,
    CAST(MUTLISET(
                SELECT bh.country
                FROM bird_habitats bh
                WHERE bh.genus = b.genus
                      AND bh.species = b.species)
AS contry_tab_t)
FROM birds b;
```

Uvedené funkce je možné použít i k naplnění hnízděné tabulky:

```
INSERT INTO Polygons VALUES('square',
    CAST(
          MULTISET(
                SELECT PointType(x, y)
                FROM PolyFlat
                WHERE name = 'square'
          )
    AS PolygonType
));
```

kde

```
CREATE TABLE PolyFlat (
    Name  VARCHAR2(20),
    x     NUMBER,
    y     NUMBER
);

CREATE TYPE PointType AS OBJECT (
    x     NUMBER,
    y     NUMBER
);

CREATE TYPE PolygonType AS TABLE OF PointType;

CREATE TABLE Polygons (
    name        VARCHAR2(20),
    points      PolygonType)
NESTED TABLE points STORE AS PointsTable;
```

V tomto případě byly nejdřív dotazovány souřadnice bodů z relační tabulky PolyFlat, získaná odpověď následně převedena na kolekci použitím funkce MULTISET a přetypována na kolekci typu PolygonType použitím funkce CAST.

- **TABLE**

Funkce TABLE mapuje kolekci do databázové tabulky, tedy je to funkce inverzní k MULTISET.

```
SELECT *
FROM color_models c
WHERE 'RED' IN (SELECT * FROM TABLE(c.colors));
```

V následující tabulce jsou shrnuty případy použití jednotlivých typů kolekcí.

Schopnost	VARRAY	Hnízděná tabulka	Asociativní pole
být indexován nečíselnou hodnotou	NE	NE	ANO
udržovat pořadí elementů	ANO	NE	NE
Uložený v databázi	ANO	ANO	NE
možnost výběru konkrétního elementu	ANO	ANO	---
možnosti změny konkrétního elementu	ANO	NE	---

Tab. 5.2 Typy kolekcí dle použití

Při rozhodování použití vhodného typu kolekce je dobré mít na zřeteli hlavní přednosti jednotlivých typů kolekcí.

Kolekce typu VARRAY lze s výhodou použít tam, kde

- je potřebné udržovat pořadí seznamu
- se pracuje s pevně danou množinou se známým počtem elementů.
- je potřebné kolekci uložit jako součást databáze a operovat s ni jako s celkem.

Použití hnízděných tabulek je výhodné zejména v případech, kdy

- pracujeme s neomezeným seznamem, který je nutné dynamicky zvětšovat
- je potřebné kolekci uložit jako součást databáze a pracovat s každým elementem samostatně

Asociativní pole jsou vhodná hlavně v případech, když

- není nutné uchovávat kolekci jako součást databáze. Díky své rychlosti a schopnosti indexace jsou asociativní pole ideální pro použití v interních aplikacích.

Dokud nedojde k inicializaci, má VARRAY i hnízděná tabulka hodnotu NULL. K inicializace je nutno použít konstruktor, který je systémem definovanou funkcí se stejným jménem jako má kolekce. Pomocí konstruktoru je vytvořena kolekce sestávající s elementů v něm uvedených.
V případě pole VARRAY není nutné inicializovat všechny elementy. Není-li v konstruktoru uvedena žádná hodnota, je vytvořena prázdná ale nenulová kolekce.
Jedna kolekce může být přiřazena do druhé prostřednictvím přiřazovacích dotazů INSERT, UPDATE, FETCH resp. SELECT resp. přiřazením v nějaké metodě. Aby bylo přiřazení možně provést, je nezbytné, aby byly kolekce stejného typu, tzn. nestačí, když budou mít pouze elementy stejného typu.
U kolekcí je možné provádět test, zda obsahují hodnotu NULL, není ale možné provádět vzájemné srovnávání dvou kolekcí na rovnost, byť jsou stejného typu.

Elementy pole VARRAY resp. řádky hnízděných tabulek mohou být uživatelsky definovaných objektových typů, přičemž pole VARRAY a hnízděná tabulka mohou být použity jako atributy těchto uživatelsky definovaných typů. Na druhou stranu ale systém Oracle nepodporuje vnořování kolekcí. Tzn. elementem pole VARRAY resp. řádkem hnízděné tabulky nemůže být jiné pole VARRAY ani hnízděná tabulka včetně uživatelsky definovaných objektových typů, které obsahují atributy těchto typů kolekcí.

Aplikace mají možnost ukládat používané objekty lokálně na straně klienta. V případě zaplnění *cache* paměti jsou nejstarší objekty z této paměti vypouštěny. Cache paměť na straně klienta výrazně zvyšuje výkonnost.

5.3 Výhody a nevýhody normalizace databází.

Tradiční relační databázové systémy pracují s normalizovanými tabulkami, většinou v 1NF. Normalizace sebou přináší řadu výhod, jako jsou:

- celkově lepší struktura databáze
- redukce nadbytečných dat
- konzistence dat v databázi
- flexibilnější návrh databáze
- lepší zabezpečení databáze

Výsledkem procesu normalizace databáze je vhodně uspořádaná struktura, usnadňující práci každému, kdo přijde s databází do kontaktu. Dojde k omezení nadbytečných dat, což

zjednodušuje datovou strukturu a šetří prostor na disku. Vzhledem k minimalizaci duplicity dat se také významně snižuje možnosti výskytu nekonzistentních dat.
Z normalizace a rozdělení databáze do menších tabulek plyne větší volnost při modifikaci vytvořené struktury. Je mnohem snadnější upravovat strukturu tabulky s menším množstvím dat než jednu velkou tabulku obsahující všechna nepostradatelná data databáze. Je také možno lépe zajistit bezpečnost, a to v tom smyslu, že správce databáze má u normalizované databáze lepší kontrolu nad přístupovými právy. Co je však ještě důležitější, nebude docházet k porušování integrity dat. Normalizací databáze se zjednodušuje zachovávání správnosti dat, čímž se všem zúčastněným zjednoduší práce a přispěje to ke spokojenosti koncového uživatele.
Normalizace ale sebou přináší i nevýhody. Ačkoli většina databází bývá přinejmenším do určité míry normalizována, stojí proti výhodám normalizované databáze jedna podstatná nevýhoda: snížený výkon. Normalizovaná databáze vyžaduje ke zpracování transakcí a dotazů více času procesoru, místa v paměti a vstupně-výstupních operací nežli denormalizovaná databáze. Důvodem je, že databázový systém musí kvůli získání požadovaných informací nebo zpracování požadovaných dat potřebné tabulky nejprve najít a poté k sobě přiřadit jejich data.
Denormalizace je procesem snížení úrovně normalizace databáze o jeden nebo dva stupně. Jejím provedením se sníží počet tabulek a jejich struktura se modifikuje do takové podoby, která za cenu zvýšené redundance dat zvýší výkon databáze snížením počtu tabulek, které by musel databázový stroj při výběru dat spojovat.
Větším problémem se ale jeví zajištění referenční integrity, jelikož související data se budou vyskytovat v několika tabulkách.
Závěrem odstavce věnovaného normalizaci a denormalizaci lze říct, že použitím kolekcí je možné vytvářet jednodušší a v mnoha případech (v závislosti na charakteru dat) rychlejší databázové aplikace.

5.4 Komponenty objektových typů PL/SQL

Jak je známo z OO jazyků, objektové typy zapouzdřují data a operace umožňující manipulaci s těmito daty. Kromě klasických metod je možné specifikovat i tzv. MAP a ORDER metody, které umožňují uspořádat instance objektových typů na základě uživatelem definovaného způsobu.
Objektový typ může obsahovat nejvýše jednu MAP metodu. Tato metoda nemá parametry a vrací relativní pozici daného objektu k pořadí všech objektů stejného typu. Tzn. že MAP metoda mapuje objekty na skalární uspořádatelné hodnoty, které jsou následně porovnávány standardními operátory.
Pro přímé porovnání dvou objektů slouží metoda ORDER. Podobně jako u metody MAP je možné pro každý typ definovat nejvýše jednu metodu ORDER. Ta má za parametr druhý porovnávaný objekt stejného typu. Její návratová hodnota musí být číselného typu. Při existenci jedné z metod je možné provádět porovnávání objektů jak v SQL tak v procedurálních výrazech.
Není možno deklarovat obě metody současně. V případě, kdy není definovaná ani jedna z nich, lze porovnávat objekty pouze v SQL výrazech a to pouze na rovnost.
Při třídění velkého počtu objektů je vhodnější použít metodu MAP, která jedním zavoláním namapuje všechny objekty do skalárního typu, který pak lze jednoduše setřídit. Metoda ORDER je pro takový případ méně efektivní, protože by musela být pro celé setřídění volána opakovaně.

```
CREATE TYPE Runner AS OBJECT (
    Id          NUMBER,
    weight      NUMBER,
    speed       NUMBER,
    MAP MEMBER FUNCTION convert RETURN REAL,
    ...
);

CREATE TYPE BODY Runner AS
    MAP MEMBER FUNCTION convert RETURN REAL IS
    BEGIN
          RETURN speed / weight;
    END convert;
    ...
END;

CREATE TYPE Parcel AS OBJECT (
    Id          NUMBER,
    width       NUMBER,
    height      NUMBER,
    ORDER MEMBER FUNCTION match (c Parcel) RETURN INTEGER
);

CREATE TYPE BODY Parcel AS
    ORDER MEMBER FUNCTION match (c Parcel) RETURN INTEGER IS
    BEGIN
          IF width*height < c.width*c.height THEN
                RETURN -1;
          ELSIF width*height > c.width*c.height THEN
                RETURN 1;
          ELSE
                RETURN 0;
          END IF;
    END;
END;
```

Při vytváření nových datových typů může nastat situace, kdy je jeden typ závislí na druhém a současně i druhý typ na prvním. V takovém případě lze využít výhody tzv. *dopředné typové definice* (forward type definitions), kdy se u jednoho z typů uvede pouze jeho název a jeho tělo se dodefinuje později. Takový typ se označuje jako *neúplný objektový typ* (incomplete object type).
Při práci s objektovými tabulkami, kterých položky určuje konkrétní objektový typ, se používají speciální funkce:

- **VALUE.**
  Funkce VALUE přijímá jako svůj argument řádkovou proměnnou resp. alias objektové tabulky a vrací instance objektu uloženého v dané objektové tabulce.

```
DECLARE
    st    Student;
BEGIN
    SELECT VALUE(s) INTO st FROM student s WHERE s.OID=10000;
END;
```

- **REF**
  Odkaz na konkrétní objekt je možné získat prostřednictvím funkce REF.

```
DECLARE
    ref_st      REF Student;
BEGIN
    SELECT REF(s) INTO ref_st FROM student s WHERE OID=10000;
END;
```

Dojde-li k smazání objektu, na který funkce REF odkazovala, vrací funkce REF hodnotu nepojeného objektu. Tento případ lze ošetřit pomocí SQL predikátu *IS DANGLING*.

- **DEREF**
  Není možné se k jednotlivým položkám objektu dostat prostřednictvím odkazu. Je nutné použít funkci DEREF, která zajistí dereferenci odkazu a vrátí hodnotu, na kterou bylo odkazováno. Tzn. argumentem funkce DEREF je odkaz na objekt.

5.5 Oracle Cartridge

Databázi Oracle [3] je možné dále rozšiřovat o další funkčnost pomocí komponent, nazývaných cartridge. Každý cartridge komunikuje se serverem přes definovaná rozhraní univerzálního serveru.
Mezi standardní cartridge patří:

- PL/SQL Cartridge
- Java Cartridge
- ODBC Cartridge
- WebServerOracle Cartridge
- Oracle Intermedia Cartridge, obsahující Oracle Text

5.5.1 Oracle Intermedia

Tato cartridge rozšiřuje Oracle o služby pro uchovávání, řízení a získávání obrázků, audio a video dat [4]. Umožňuje pracovat s objektovými typy:

- ORDAudio – pro audio data
- ORDImage – pro obrázkové data
- ORDVideo – pro video data
- ORDDoc – může obsahovat data všech uvedených typů

Intermedia podporují práci s více druhy objektů:

- BLOB
- File-based large objects – uchovávané lokálně v operačním systému
- URL – uchovávané na jakémkoliv http serveru
- Audio resp. video data uchovávaná na specializovaném médiu

Intermedia jsou přístupné aplikacím jak přes relační, tak přes objektové rozhraní. Je možná i spolupráce s jazyky třetí generace prostřednictvím knihovních rozhraní. Jsou podporovány všechny nejpoužívanější souborové formáty, čímž je umožněno rozšířit existující databázové aplikace o multimediální data. Multimediální data přitom mohou být uloženy běžným způsobem a to buď v podobě BLOBů uvnitř databáze resp. mimo databáze bez transakční kontroly a to v podobě externích binárních soborů (BFILE), na http serveru prostřednictvím URL resp. na uživatelsky definovaných zdrojích.
Multimediální data uložena mimo databázi mohou sloužit jako vhodný mechanismus spravování velkých mediálních skladišť z nichž pak mohou být potřebná data importovaná jako BLOBy v kterémkoli okamžiku.
Pro každý multimediální objekt jsou vytvořeny odpovídající metadata, která jsou součástí databáze.
Všechny podporované multimediální formáty je možné rozšířit.

Mezi aplikace využívající Intermedia patří:

- Internetové audio/video obchody
- Skladiště digitálních zvuků
- Digitální galerie
- Fotografické alba

5.5.2 Oracle Text

Pro práci s textem slouží součást Oracle Intermedia, nazývaná Oracle text. Rozšiřuje možnosti SQL o indexování, vyhledávání a analýzu textů, uložených v databázi, v souborech resp. na webu. Umožňuje vyhledávání v textu různými strategiemi jako vyhledávání podle klíčových slov, kontextové dotazy, boolovské vyhledávání, vyhledávání podle vzorů. Výsledek vyhledání může být ve více formátech – v podobě neformátovaného textu, HTML se zvýrazněním hledaných slov, resp. originálním formátu dokumentu.

6 SQL:1999, DB2, Informix

Implementace OR rozšíření firmou Oracle, jak byla představena v předchozí kapitole, stejně jako obdobná rozšíření v databázích jiných výrobců, nerespektuje v řadě aspektů standardizovanou podobu, tak jak ji požaduje ANSI norma SQL:1999. V této kapitole jsou m.j. uvedeny některé základní rozdíly v definicích objektových hierarchií. Výrazné rozdíly jak v množství podporovaných OR rysů, tak ve způsobu jejich implementace, by měl mít programátor, který chce pro svoji aplikaci objektový návrh databázové vrstvy použít, na paměti.

6.1 SQL:1999

Začleněním OR rysů do SQL:1999 [11] se dosáhlo standardizovaného soužití objektů a relací.
SQL:1999 definuje několik nových rysů:

- Rozšiřitelnost.

  První oblastí, která se dočkala rozšíření byla data. Přibyla možnost reprezentovat data jako velké objekty, tzv. BLOB resp. CLOB, včetně podpory pro manipulaci s těmito objekty. Jedná se hlavně o funkce umožňující vyhledávání v objektech těchto typů. Pro původní relační systémy byla vnitřní struktura takovýchto objektů neznámá.

- Nové datové typy.

Uživatelovi je dána možnost vytvářet vlastní datové typy včetně uživatelsky definovaných funkcí, čímž vytváří infrastrukturu nových objektů. Tyto nové objekty musí ale koexistovat s ostatními daty v databázi, přičemž SQL jim musí rozumět včetně funkcí a operátorů, které s nimi pracují.

➢ UDT (abstraktní datové typy, typy řádků a odlišující typy).

Zvláštní pozornost si zasluhuje možnost vytvářet *abstraktní datové typy*, s kterými lze pak dále pracovat jako s vestavěnými typy. Prostřednictvím ADT lze zapouzdřovat atributy i operace, jako jsou např. porovnávací operace. Tyto operace se označují jako *virtuální metody*. Každá taková metoda je implementovaná pomocí podprogramů a to buď přímo v SQL nebo pomocí nějakého standardního programovacího jazyka. Ve spojitosti s ADT je možné vytvářet taky uživatelsky definované funkce resp. procedury.
Příklad ADT lze ukázat na typu Student_type:

```
CREATE TYPE Student_type AS (
      Jméno        CHAR(30),
      adresa       CHAR(40),
      začátek_ studia DATE)
      UNINSTANTIABLE NOT FINAL
      METHOD absolvování_přednášky() RETURNS INTEGER;
```

U nově vytvářeného typu je možné určit, zda-li z něj bude možné vytvářet instance či nikoliv. To se provádí pomocí klíčového slova *INSTANTIABLE* resp. *UNINSTANTIABLE*. Ve druhém případě slouží typ jako společný předek odvozených typů, ze kterých se vytvoří vlastní instance. Klíčové slovo *NOT FINAL* určuje, že od definovaného typu mohou být děděním odvozeny další

potomci. Vlastní definice metod se provádí pomocí příkazu *CREATE METHOD*.

```
CREATE METHOD absolvování_přednášky() FOR Student_type
BEGIN
    ...
END;
```

Obdobným způsobem je možné definovat nové datové typy v systému Oracle (viz kap. 5.2). Při pohledu na definice u obou systému je patrná syntaktická rozdílnost, kdy u Oracle je nutné při vytváření nového typu uvádět klíčová slova AS OBJECT místo AS, jak je tomu u SQL:1999. Rovněž při vytváření abstraktních typů, které budou sloužit jen jako společní předkové pro další odvozené typy je u Oracle za definicí nového typu uvedeno NOT INSTANTIABLE, zatímco SQL:1999 vyžaduje klíčové slovo UNINSTANTIABLE. Dalším rozdílem v syntaxi je deklarace metody nového typu, kdy j Oracle používá klíčového slova MEMBER FUNCTION resp. MEMBER PROCEDURE, přičemž standard SQL:1999 definuje klíčové slovo METHOD. Rovněž u vlastní definice metody je rozdíl, u Oracle jsou všechny definice součástí těla nově deklarovaného typu, a jsou uvozené prostřednictvím klíčových slov MEMBER FUNCTION/PROCEDURE. U SQL:1999 se vytváří každá metoda samostatné pomocí příkazu CREATE METHOD název_metody FOR datový_typ jak ukazuje předchozí příklad.

Odvozený typ potomka se vytváří příkazem

```
CREATE TYPE student_pgs UNDER student_t ...
```

V SQL:1999 je požadována pouze jednoduchá dědičnost, tzn. nový typ nemůže zdědit vlastnosti od více rodičů současně.
Podobně jako u typů, lze v SQL:1999 vytvářet i odvozené tabulky, kdy nové podtabulky mohou mít přídavné sloupce. Hodnota typu řádek může být umístěna jako hodnota sloupce tabulky. Tzn. pomocí tohoto typu je možné vytvářet složené atributy. Typ řádku může přitom být pojmenovaný resp. nepojmenovaný. Pojmenovaný typ řádku je nezapouzdřený typ definovaný uživatelem. Takový typ je možné využít při definici nové tabulky:

```
CREATE ROW TYPE automobil_t
(
    číslo       INT,
    zákazník    REF(osoba_t),
    typ         CHAR(10),
    barva       CHAR(10),
    cena        INT
);

CREATE TABLE auta OF automobil_t
    (PRIMARY KEY číslo);
```

V takto vytvořené tabulce nejsou jednotlivé řádky identifikovány pomocí OID tak, jak by tomu bylo v případě použití ADT.

Prostřednictvím *typu řádky* je možné strukturovat řádky tabulek.

```
CREATE TABLE Studenti
(
      jméno ROW(
            příjmení    VARCHAR(10),
            křestní     VARCHAR(20)
      ),
      adresa      ROW(
            ulice VARCHAR(10),
            město VARCHAR(10),
            PSČ   VARCHAR(5)
      )
);
```

Zde je vidět další rozdíl v implementaci SQL standardu u Oracle, který datový typ řádek nezná. Nové typy jsou u něj deklarovány prostřednictvím TYPE.

K zajištění kompatibility mezi typy slouží *odlišující typy*. Ty umožňují rozlišovat mezi typy, jejich základní množiny jsou stejné, přitom ale spolu nesouvisejí. Jako příklad může posloužit výška platu zaměstnanců a jejich tělesná výška, které mezi sebou nijak nesouvisejí.

- Konstruktory typů pro typy řádků a typy odkazů

  Prostřednictvím odkazů je možné odkazovat se na instance ADT. Způsob, který se možné odkazování záleží na typu tohoto odkazu. Ten může být buď definovaný uživatelem, nebo generovaný systémem nebo resp. založený na nějakém jednoznačném atributu. Hodnota odkazu může ukazovat do jakékoliv tabulky, která má definované řádky pomocí daného typu. Toto odkazování je možné omezit pomocí klauzule *SCOPE* na konkrétní tabulku.

- Konstruktory typů pro typy kolekcí (ARRAY)

  V sloupcích typu kolekce ARRAY je možné uložit v jednom políčku tabulky kolekci hodnot jako pole.

```
rok VARCHAR(8)   ARRAY[12];
```

  Tabulky obsahující tento typ stejně tak jako typ pro hnízděné řádky již nesplňují požadavek pro 1NF.

  Další typy kolekcí jako jsou množina (SET), seznam (LIST) a multimnožina (MULTISET) jsou zahrnuty v definici SQL4.

  I zde je rozdíl mezi systémem SQL standardem a Oracle, který místo ARRAY používá pro pole klíčové slovo VARRAY. Při deklaraci nové proměnné typu pole je u Oracle navíc nutné nejdříve definovat nový typ pro pole a následně na jeho základě deklarovat novou proměnnou:

```
TYPE var_rok IS VARRAY(12) OF VARCHAR(8);

rok        var_rok
```

- Uživatelem definované funkce a procedury

  Kromě vestavěných funkcí využívaných v SQL, je možné definovat i vlastní funkce a procedury. Ty mohou mít vstupní (IN), výstupní (OUT) resp. vstupněvýstupní (INOUT, systém Oracle používá zápis IN OUT) parametry, přičemž funkce obsahují pouze vstupní parametry a navracejí jednu hodnotu.

  Máme-li následující tabulku studentů:

```
STUDENTI(jméno, adresa, ročník, průměr_známek)
```

  je možné vytvořit proceduru, která spočítá průměr všech studentů daného ročníku:

```
CREATE PROCEDURE PRUMER
(IN rok INTEGER, OUT průměr DOUBLE)
BEGIN
     SELECT AVG(průměr_známek) INTO průměr
     FROM STUDENTI WHERE ročník=rok;
END
```

Volání procedur se provádí pomocí příkazu *CALL*, volání funkcí lze provést přímo aplikací funkce ve výrazu.

6.2 Nové objektově-relační rysy v DB2

Abstraktní resp. strukturované datové typy slouží k modelování a uchovávání složitých objektů v relačních databázích. Strukturované typy mohou mít vícenásobná pole.

- **Strukturované typy.**
  Díky strukturovaným typům je možné vytvářet tabulky, kterých sloupce jsou těchto typů. Navíc mohou být strukturované typy hnízděné, tzn. atributy strukturovaných typů již nemusejí být omezeny na základní SQL typy.
  Je možné definovat funkce pracující těmato typy. Pro každý strukturovaný typ je navíc možné definovat metody pracující s danými typy.

- **Datové typy pro velké objekty.**
  Někdy je potřeba při modelování nového systému použít velká a složitá data jako jsou text, audio, vědecká data resp. video data. Klasické datové typy jsou pro tyto účely nepoužitelné. Systém DB2 poskytuje tři datové typy určené pro velké objekty (2 GB): BLOB, CLOB a DBCLOB (Double-Byte Character Large Objects).

- **Datový typ odkaz – REF.**
  Jedná se o vestavěný datový typ, ukazatel na objekt typu řádky. S pomocí tohoto typu je možné redukovat použití cizích klíčů ve vytvářených systémech. Umožňuje jednoduchou navigaci mezi objekty.
  Ukazatel přitom může během svého života odkazovat na více objektů, tzn. lze jej podle potřeby přesměrovat.
  Při deklaraci objektů je navíc možné definovat omezení podle kterého může pointer odkazovat pouze na specifikovanou objektovou tabulku, tzn. určit jeho rozsah. Výhodou ukazatelů tohoto druhu je, že vyžadují méně úložného prostoru a umožňují efektivnější přístup k objektům.
  V případech, kdy dojde k zrušení objektu na který bylo odkazováno, stává se takový ukazatel *volný*.
  Dereference ukazatele se provádí tečkovou notací.

- **Uživatelsky definované datové typy.**
  Uživatelsky definované typy umožňují přímo kontrolovat sémantiku vytvářených objektů. Takto vytvářené typy mohou být buď *neprůhledné* – vytvářené na základě vestavěných typů, nebo *strukturované*, kdy jde o sjednocení kolekce atributů do jednoho typu.

- **Uživatelsky definované funkce a metody.**
  Uživatelsky definované funkce je možné definovat pro manipulaci s objekty SQL dotazech.
  Uživatelsky definované metody definují chování objektů a jsou spjaty s uživatelsky definovaným strukturovaným typem.

- **Rozšíření indexů.**
  Toto rozšíření umožňuje specifikovat způsob, kterým má systém indexovat nově vytvořené strukturované a neprůhledné typy. K tomu je nutné použít výraz *CREATE INDEX EXTENSION*, který specifikuje funkci externí tabulky konvertující hodnoty strukturovaného resp. neprůhledného typu na indexový klíč a definuje způsob, optimalizace výkonu prostřednictvím těchto klíčů.

- **Transformační funkce.**
  Transformační funkce umožňují použití sloupců strukturovaných typů v uživatelských programech. Tyto funkce převádějí složitou strukturu datových typů do uspořádané množiny základních SQL typů. Dále umožňují konverzi základních atributů zpět na strukturovaná data. Tyto transformace jsou vyžadovány při přesunu strukturovaných typu z nebo do databáze.

- **SQL-Bodied funkce.**
  Tyto funkce obsahují vložené jednoduché SQL výrazy. Tím je umožněno kompilátoru optimalizovat celý SQL výraz obsahující SQL-bodied funkci.

- **Dynamické smíšené výrazy.**
  K snižování režie řízení databáze a zvýšení výkonu při vyřizování požadavku po síti slouží dynamické smíšené výrazy. Jsou ideální pro krátké skripty zahrnující logiku toku dat.
  S výrazy tohoto typu je možné deklarovat SQL proměnné, podmínky asociované s SQL výrazy a taky je možné použít při kontrole toku logiky výrazů prostřednictvím příkazů FOR, IF, ITERATE resp. WHILE.

- **Proměnné a triggery pro kontrolu toku.**
  Procedurální logiku je možné vykonávat uvnitř vložených procedur, triggerů a SQL funkcí prostřednictvím množství SQL-controlled výrazů.
  Původní přístup umožňoval pouze jednoduché řazení, tzn. bez podmínek resp. cyklů. Prostřednictvím vylepšených triggerů je možné snáze provádět migraci aplikací do DB2.
  V SQL funkcích resp. triggerech je možné použít několik výrazů pro kontrolu toku:
  - Atomické compound výrazy.
  - SQL kontrolní výrazy:
  - FOR
  - GET DIAGNOSTIC
  - IF
  - ITERATE
  - LEAVE
  - WHILE
  - Lokální proměnné SQL.

6.2.1 Koncepce rozšíření DB2 - Extenders

Extenders jsou kompletní balíčky definující speciální typy a funkce pro různé typy BLOBů, jakými jsou obrazové, audio, video, textové, prostorové a XML objekty. Díky extendrům není nutné zabývat se definováním datových typů a funkcí vhodných pro konkrétní BLOBy.
Pro práci s obrazovými daty, kdy je potřebné vyhledávat obrázky na základě jejich obsahu je určen DB2 *Image Extender*, který dále umožňuje:

- Dotazovat obrazová data na základě jejich obsahu, prostřednictvím operátoru LIKE definovaný uživatelem (barvy, barevná struktura, atd.).
- Klasifikovat obrázky podle jejich vzájemné podobnosti k dotazovanému obrázku.
- Provádět indexaci obrázků, díky které jsou uloženy číselné popisy barev a textur obrázků, které jsou ve fázi zpracování porovnávané s popisy dotazovaného obrázku.
- Snadné použití webových šablon a rozhraní, které umožňují uživateli přidávat vlastní vyhledávací metody a filtry.
- Kombinovat obrazové dotazy s dotazy na textová data.

DB2 Geodetic Extender podporuje sférické zpracování souřadnic zemského povrchu.
Tím se zjednodušuje vývoj aplikaci vyžadujících geografické analýzy.
Uchováváním, zpracováním a analyzování prostorových dat se zabývá *DB2 Spatial Extender*.
Pomocí tohoto extendru je možné generovat, analyzovat a využívat geografických prostorových informaci.

6.3 Objektově-relační rozšíření Informixu

- **Uživatelsky definované typy.**
  Tyto typy mohou být jak složité – pojmenované a nepojmenované řádkové typy, kolekce – množina, multimnožina, seznam. Dále je možno vytvářet neprůhledné datové typy, se kterými se pracuje pomocí uživatelsky definovaných rutin.
  Existující vestavěné datové typy je možné rozšířit následujícími způsoby:

  - Vytvořením složitých datových typů založených na vestavěných datových typech.
  - Vytvořením uživatelsky definovaných datových typů (distinct and opaque data types).
  - Rozšířením operací pracujících s oběma druhy datových typů.

- **Typová dědičnost.**
  Typová a tabulková dědičnost funguje na stejných principech jako u OO systémech.

- **Uživatelsky definované rutiny.**
  Mohou být napsané v jazyce C, C++, Java resp. SPL. Jsou uchovávány na straně serveru, který uchovává plány dotazů a interní datové struktury dotazu.

- **Přístupové metody.**
  Tradiční relační databázové systémy neumožňují vytvářet indexy nad daty typu text resp. obrázek. Informix umožňuje uživateli vytvářet různé druhy *přístupových metod* pro různé druhy datových typů.

6.3.1 Koncepce rozšíření INFORMIXU – datablade modul

Datablade modul [2] je softwarový balíček rozšiřující funkcionalitu Informixového dynamického serveru. Kromě možnosti psát kód programů v externích jazycích obsahuje i SQL výrazy.
Rozšiřuje SQL syntaxi o uživatelsky definované databázové objekty – datové typy, rutiny resp. databázové tabulky. Při práci s těmito uživatelskými objekty spouští databázový server odpovídající kódy poskytované datablade modulem.

Rozšíření databázového serveru spočívá hlavně v:

- **Typech dat**. Do této kategorie patří všechny nové datové typy, které nejsou vestavěny v databázovém serveru. Mohou obsahovat vícenásobné prvky jako např. řádky, kolekce, neprůhledné typy a datové typy podporující dědičnost.
- **Rutinách**. Tato kategorie zahrnuje uživatelsky definované rutiny, agregace, rutiny přetypování a rutiny podporující uživatelsky definované přístupové metody.

Hlavním důvodem vedoucím k rozšíření databázového serveru je:

- **Zvýšení výkonnosti**, které je zajištěno:
    - Optimalizovanými uživatelsky definovanými rutinami.
    - Indexy, které je možno vytvářet i nad daty, které nemohou být setříděny v standardní relační databázi.
    - Snížení objemu dat přenášených sítí. Užíváním uživatelsky definovaných rutin probíhá většina zpracování dat uvnitř databázového serveru.
- **Zjednodušení aplikací** následujícími způsoby:
    - Kód zabezpečující uchovávání a manipulaci s daty zpravuje datablade modul, čímž odlehčuje od této práce aplikace.
    - Jak k rutinám datablade modulu tak k datovým typům může být přistupováno prostřednictvím SQL.
    - Jednoduchá aktualizace existujících datablade modulů. Tu má na starost databázový server, takže při jakýchkoli změnách není nutné přelinkovávat celou aplikaci.
    - Všechny data jsou uložena a zpracovávána na jednom databázovém serveru.
    - Je zajištěna jednoduchá kombinace databázových modulů.
- **Transakční kontrola**. Datablade modul je součástí databáze. Všechny operace prováděné uvnitř datablade modulu jsou podporovány databázovými službami zajišťujícími např. zálohování, rollback, zotavení.
- **Rozšiřitelnost**. Informixový databázový server je možné rozšířit i bez vytvoření vlastního databladu pomocí individuální tvorby objektů prostřednictvím SQL.

Datablade moduly mají ale několik výhod:

- Umožňují lepší *kontrolu*. Zahrnují v sobě všechny odpovídající objekty, umožňují snadnou instalaci, aktualizaci včetně odstranění celého modulu jako celku. Jak řešení nového problému tak přidání dalšího rysu do programu probíhá na jednom místě.
- *Znovupoužití kódu* je zajištěno možností využívat jiných datablade modulů přes vytvořené rozhraní.

Tento balík obsahuje SQL příkazy a podporuje i kódy napsané v externích jazycích. Datablade může dále obsahovat klientské komponenty. Umožňuje informixovému databázovému serveru stejnou úroveň podpory pro nové datové typy jako pro vestavěné datové typy.
Uživatel přistupuje k službám poskytovaným datablade modulem stejným způsobem jako přistupuje k službám databázového serveru: pomočí SQL, SPL resp. klientské programy.

Datablade modul může dále použít Datablade API resp. SQL dotazy pro přístup k datovým typům a rutinám jiných databladů.
Informix obsahuje následující datablady:

- textový datablade – Excalibur Text Search Datablade
- obrazový datablade – Excalibur Image Datablade
- webový datablade – Web Datablade
- prostorový datablade – Spatial Datablade
- vývojový nástroj – Datablade Developers Kit Tools

Excalibur Text Search Datablade je soubor datových typů a rutin, které rozšiřují Informix Dynamic Server a umožňují vyhledávání v textu rychlejším a sofistikovanějším způsobem než poskytuje tradiční SQL. Zahrnuje dále vyhledávání frází, přesné a fuzzy vyhledávání, kompenzace pro chyby v pravopisu a porovnávání synonym.
Umožňuje provádět následující způsoby vyhledávání:

- pomocí klíčových slov
- vyhledávání frází
- booleovské vyhledávání
- přesné a přibližné vyhledávání

Excalibur Image Datablade kombinuje Excalibur image technologii s Universal data options pro uchovávání, získávání a vyhledávání obrázků v databázi. S tímto modulem je možno užívat SQL příkazy pro uchování skupiny obrázků v databázi a jejich následné vyhledávání podle jejího obsahu. Jednotlivé obrázky jsou popsané atributy umožňujícími výběr a vyhledávání podobných obrázků.
Technologie, s kterou pracuje Excalibur Image Datablade je založena na vektoru charakteristik (feature vector). Pomocí tohoto vektoru je možné provádět vyhledávání a porovnávání obrázků na základě jejího obsahu.

Web Datablade umožňuje vytvářet webové aplikace, které využívají data získaná dynamicky z databáze. V typické webové aplikaci pracující s databází, většina aplikační logiky je napsaná v jazycích jako Perl, Tcl nebo C. Tyto CGI aplikace se připojují k databázi, sestavují a provádějí SQL příkazy a formátují výsledek.
Použitím Web Datablade modulu není nutno vyvíjet CGI aplikace pro dynamické připojení databáze. Místo toho je možno vytvářet HTML stránky obsahující Web Datablade modul značky (tagy) a funkce dynamicky vykonávající SQL příkazy. Tyto HTML stránky se označují jako *Aplikační stránky* (AppPages). V případě neexistence tohoto databalde modulu by bylo možné jeho funkcionalitu implementovat jiným způsobem, jak ukazuje projekt [12].
Spatial datablade rozšiřuje Informixový server o možnost práce s geografickým informačním systémem (GIS). Umožňuje pracovat s datovými typy jako jsou umístění orientačních bodů, ulic resp. parcel v krajině.
Datablade Developers Kit Tools poskytuje uživateli grafické rozhraní pro vytváření a práci s datablade moduly. Grafické rozhraní pozůstává z následujících částí:

- BladeSmith – nástroj pro organizaci vývoje modulu
- DBDK Visual C++ - nástroj pro ladění modulu
- BladePack – nástroj pro vytváření balíků modulu
- BladeManager – nástroj pro registraci DataBlade modulu v informixové databázi

6.4 Porovnání objektově-relačních rozšíření

OR modely systémů Oracle, DB2 a Informix se liší.

- Řádkový konstruktor (ROW) obsahuje pouze Informix.
- Konstruktor referenčního typu je podporován systémem Oracle i DB2.
- Systém DB2 neposkytuje žádný typ pro kolekci.
- SQL:1999 má jako jedinou kolekci pole.
- Oracle obsahuje konstruktor pro pole stejně tak jako konstruktor pro typ tabulky, který definuje multimnožinu řádků. Tyto dva typy kolekcí ale nemůžou být do sebe navzájem vložené.
- Informix podporuje konstruktor pro množinu, multimnožinu a seznam.
- Strukturované typy jsou podporované všemi systémy.
- Oracle a DB2 následují standard SQL:1999 a podporují abstraktní datové typy, jejichž instance jsou objekty s vlastními identifikátory a metodami. V systému DB2 mohou strukturované typy dědit atributy a metody od jiných strukturovaných typů.
- Informix poskytuje pojmenované řádkové typy jako strukturované typy. Tyto typy nejsou asociovány s objektovými identifikátory a metodami. Přitom je ale podporovaná typová hierarchie pro řádkové typy, tzn. pojmenované řádkové typy mohou zdědit atributy od jiného řádkového typu.
- Všechny tři systémy podporují typové tabulky definované na základě strukturovaných typů. Tabulkovou hierarchii podporuje pouze systém DB2 a Informix.
- Omezení NOT NULL může být specifikováno na sloupcích strukturovaných typů stejným způsobem jako na obyčejné sloupce.
- V systémech Oracle a DB2 může být toto omezení specifikováno i individuálně pro kteroukoliv komponentu strukturovaného sloupce.

7 Výsledky

7.1 Konfigurace

Aplikace byla sestavena nad databází Oracle 9i, podrobnější konfiguraci serveru a vlastní databáze ukazuje následující výpis:

server:
GNU/Linux, Gentoo, kornel 2.6.10-r6, i686 SMP 1GB RAM
2x procesor GenuineIntel Pentium III (Coppermine) 700 Mhz, cache 256 KB

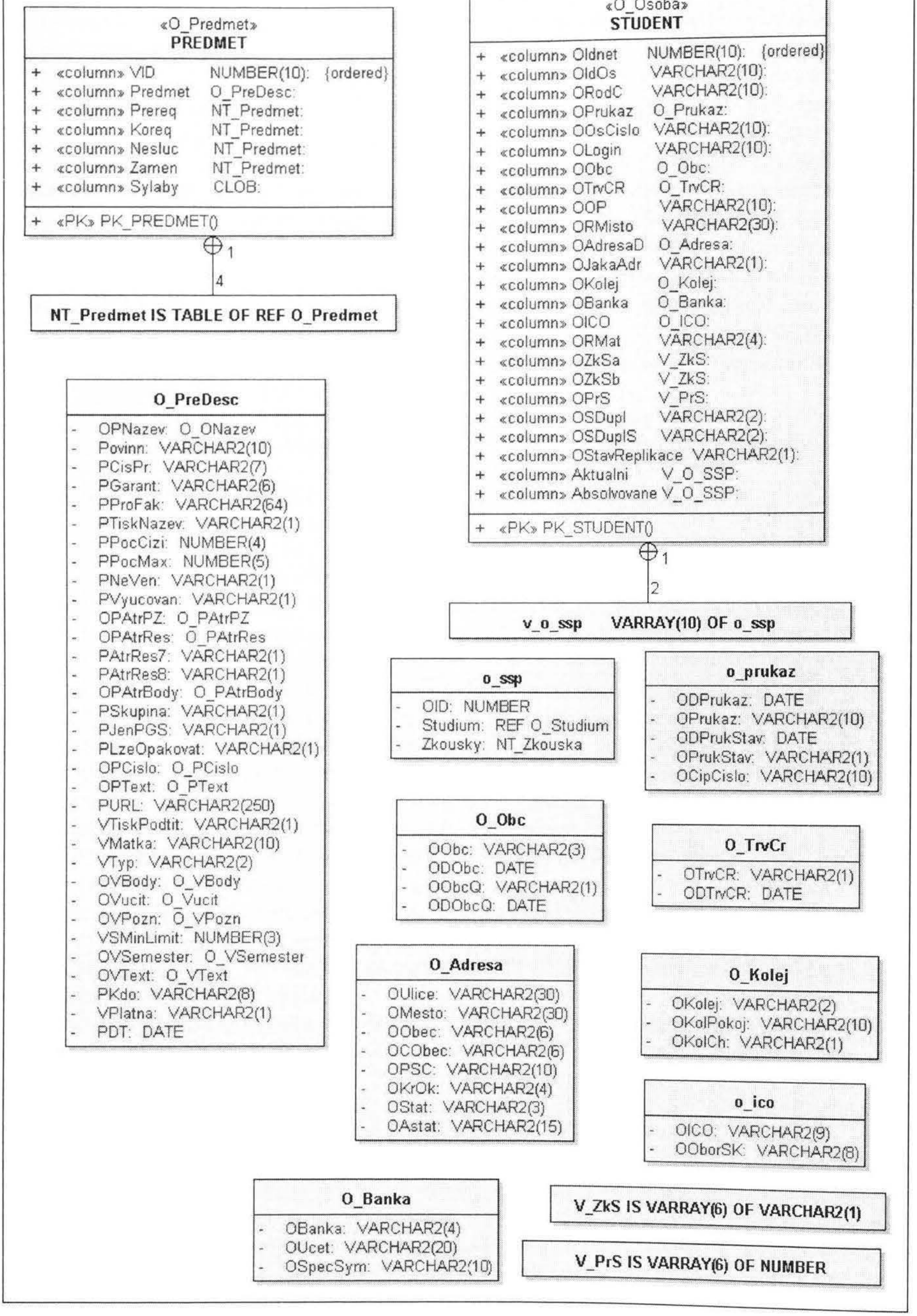
databáze:
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
PL/SQL Release 9.2.0.1.0 - Production
CORE 9.2.0.1.0 Production
TNS for Linux: Version 9.2.0.1.0 - Production
NLSRTL Version 9.2.0.1.0 - Production

7.2 Ukázková aplikace

Jako ukázkovou aplikaci, na které jsem demonstroval použití obou databázových přístupů jsem si zvolil studijní informační systém MFF. Tato aplikace je vhodná k demonstraci implementace OR rozšíření jako jsou objektové typy a objektové tabulky, hnízděné tabulky a reference. Kromě toho, že je po datové stránce dostatečně složitá, obsahuje rovněž netriviální konstrukty, jako např. rekurzivní procházení stromu všech možností. Tyto OR struktury mohou být konfrontovány s ekvivalentními klasickými relačními strukturami. Porovnána může být především časová náročnost různých typů dotazů při používání hnízděných tabulek (viz kap.5.2.8) a složitých objektů včetně referencí na ně u OR verze (viz kap.4.4) resp. bez nich u verze relační. Lze porovnávat i další detailnější charakteristiky, jakými jsou počet diskových operací, zamykání a s tím související škálovatelnost, vytížení procesoru, redundance dat posílaných klientovi, schopnost vypořádat se s prováděním opakovaných rekurzivních volání při složitějších dotazech. V neposlední řadě lze porovnat složitost zápisu dotazů, zpracování těchto dotazů optimalizátorem (viz kap. 4.7, 4.8), náročnost vytvoření datového modelu a paměťová náročnost vytvořených tabulek. Dále je zkoumána výhodnost reprezentace uložených procedur oproti procedurám, které jsou součástí balíku a rozdíl v provádění dotazů pro OR verzi při použití hnízděných tabulek a ekvivalentních dotazech bez nich.
Účelem studijního informačního systému je uchovávat, zpřístupňovat a zpracovávat údaje o studentech, jejich zapsaných a absolvovaných předmětech včetně dosažených výsledků, informací o jednotlivých předmětech, jejich vzájemné návaznosti, informace o učitelích.
Každý student si může zvolit několik studií, a pro každé studium si zapsat předměty, které chce absolvovat. Při zápisu je potřeba zajistit kontrolu, aby nebyl stejný předmět zapsán ve více studiích současně. Po absolvování zkoušky z konkrétního předmětu se předmět přesune ze zapsaných mezi absolvované předměty daného studenta, včetně zapsané známky. U zápisu předmětů musí být přitom splněna pravidla rekvizit, tzn. jako první musí být zapsány předměty, které nemají žádnou prerekvizitu, tedy předmět resp. předměty, které musí být splněny, aby bylo možno si daný předmět zapsat. Dále nesmí být nově zapisovaný předmět neslučitelný s některým již absolvovaným předmětem. V případě, že je to možné, lze využít zaměnitelného předmětu, který poslouží jako rovnocenná náhrada za požadovanou prerekvizitu.
Všechna použitá data v systému (kromě údajů, týkajících se osobních dat studentů) byla získána na základě žádosti ze studijního informačního systému MFF UK.

Z celého systému je zde realizovaná asi jeho nejdůležitější a pro ukázkové účely nejvhodnější část týkající se studentů, předmětů, zapsaných zkoušek a rekvizit pro jednotlivé předměty.
K realizaci systému je použito několik hlavních a pomocných tabulek. Hned první rozdíl mezi OR a relačním přístupem je patrný při návrhu tabulek, kdy je v relační verzi nutné některé vztahy mezi tabulkami reprezentovat prostřednictvím spojovacích tabulek, jenž je nezbytné pak v dotazech spojovat, zatímco je tento vztah pro OR tabulky realizovaný prostřednictvím hnízděných tabulek.
Práce na aplikaci probíhala v pěti etapách.
V první etapě proběhl návrh objektového modelu, na základě kterého byl vytvořen návrh relačních a OR tabulek SIS. Přetváření schématu na datový model pro OR verzi bylo o poznání jednodušší, jelikož jednotlivé namodelované entity ze schématu odpovídají jedna k jedné s entitami datového modelu. O něco složitější to bylo pro relační model, kdy bylo nezbytné pro multiatributy vytvořit vazby kardinality N:N za použití většího počtu pomocných tabulek.
Následující diagram[4] ukazuje objektový datový model informačního systému.

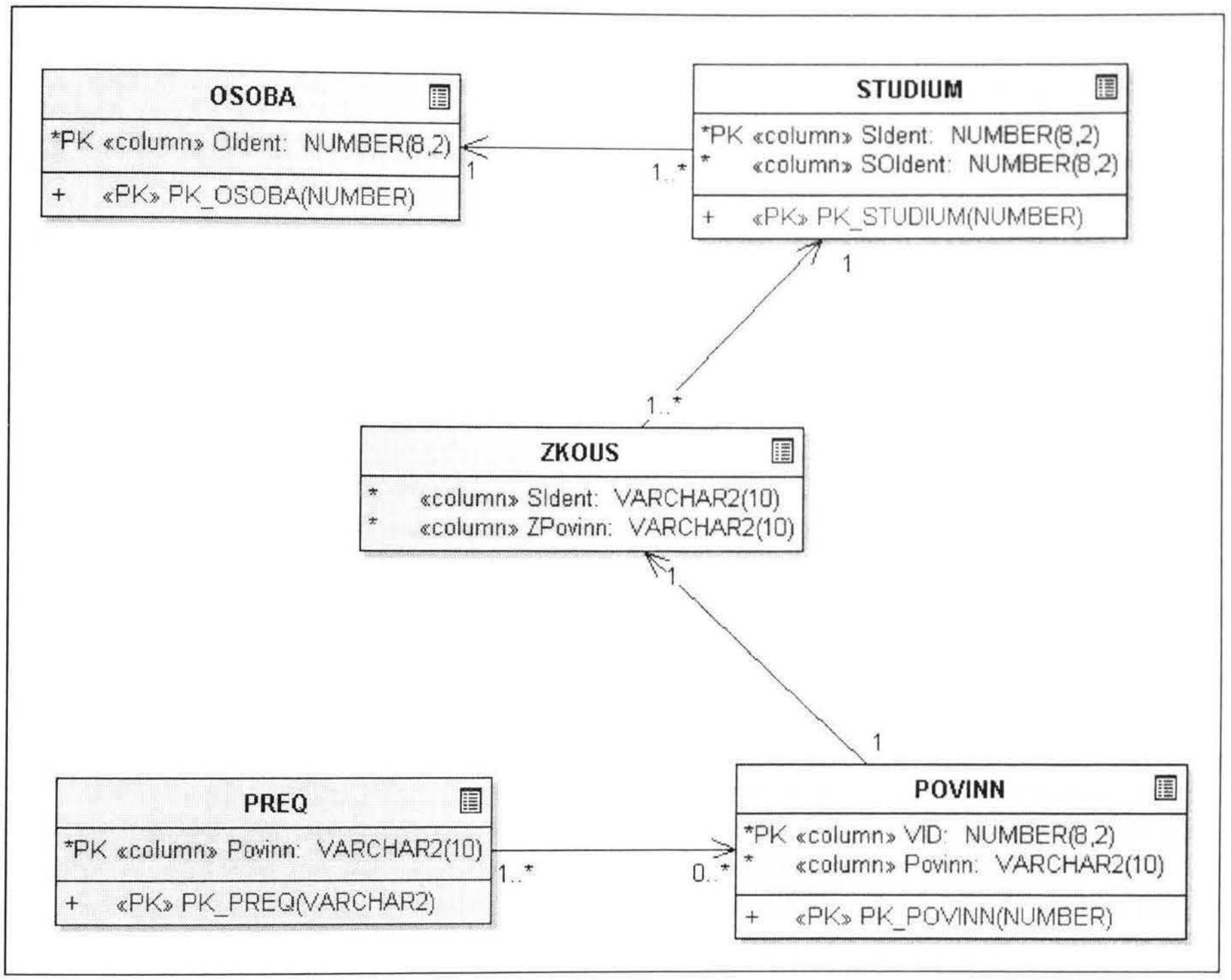
[4] Diagramy jsou vytvořené pomocí nástroje *Enterprise Architekt 5.0* - http://www.sparxsystems.com/, určenému k UML modelování



Obr. 7.1 Objektový model

Objektový model byl následně transformován do ekvivalentního relačního modelu, který ukazuje vztahy mezi relačními tabulkami, kde tabulka OSOBA reprezentuje studenty, tabulka STUDIUM různá studia jednotlivých studentů, v tabulce ZKOUS jsou záznamy pro zkoušky studentů v jednotlivých studiích. Tabulka POVINN reprezentuje všechny předměty, které je možné studovat a v tabulce PREQ jsou uloženy rekvizity všech čtyř typů k jednotlivým předmětům.

.



Obr. 7.2 Relační model

Ve druhé etapě bylo nutné vytvořit relační tabulky a ty naplnit odpovídajícími statickými daty pro studenty, předměty, učitele a další tabulky včetně číselníků. Dále byly vytvořené objektové typy a prostřednictvím nich ekvivalentní OR tabulky odpovídající relačním. Jelikož tyto OR tabulky obsahují vnořené tabulky, byly vytvořeny metody jednotlivých objektů zabezpečující jejich plnění a mazání.
V relační verzi jsou použité normalizované tabulky. K vyjádření vztahu např. zapsaného předmětu studenta resp. prerekvizity atp. jsou použité spojovací tabulky.
OR verze využívá objektů, na základě kterých jsou vytvořené objektové tabulky, u kterých je možné použití hnízděných tabulek. Tímto způsobem je realizován např. vztah zapsaných předmětů a vztah prerekvizit k předmětům.
Následující výpis zachycuje ekvivalenci mezi relačními a OR tabulkami:

- tabulka OSOBA odpovídá OR tabulce STUDENT
- STUDIUM odpovídá OSTUDIUM
- POVINN odpovídá PŘEDMET
- ZKOUS odpovídá hnízděná tabulka NT_ZKOUSKA
- PREQ odpovídá hnízděná tabulka NT_PŘEDMĚT

Vztahy z relační verze jsou v OR verzi realizovány prostřednictvím hnízděných tabulek, kdy tabulka předmětů obsahuje v záznamech položky pro rekvizity a tabulka studentů jako položky zapsané a absolvované předměty studentů. Dále jsou související položky tabulek spojeny prostřednictvím tříd do jedné položky (např. informace popisující daný předmět). Tedy jako základ jsou použity relační tabulky obsahující záznamy pro studenty, předměty, studia, učitele ..., prostřednictvím kterých se naplní odpovídající OR tabulky a následně proběhne náhodné přiřazení předem určeného počtu zkoušek pro jednotlivé studenty a rekvizit ke každému předmětu.
Ve třetí fázi byla vygenerovaná "dynamická" data, tedy data jako jsou zapsané předměty jednotlivých studentů (z tabulky předmětů) a přiřazení prerekvizit k předmětům (z tabulky rekvizit). Ty se nejdříve vygenerují pro OR verzi a následně se přesunou i do odpovídajících relačních tabulek. Přitom čistě z praktických důvodů není zachována logická souvislost mezi předmětem a prerekvizitami potřebnými pro jeho splnění.
K vlastnímu generování dat slouží procedury, které jsou součástí balíku CRUD.sql. Ten kromě pomocných funkcí (např. vracejících reference na předměty) obsahuje procedury dvou druhů. První vkládají záznamy relačních tabulek do hlavních OR tabulek (STUDENT, PREDMET), druhé provádějí generování dat do ostatních používaných tabulek.
Do první skupiny patří procedury pro vložení záznamů studentů a předmětů.
Procedury v druhé skupině zabezpečují vygenerování korekvizit, neslučitelných, zaměnitelných předmětů, u kterých je zajištěné, že navzájem zaměnitelné předměty mají stejné prerekvizity, a hierarchické závislosti prerekvizit pro každý předmět, kde jsou předměty rozděleny do n stejně velkých skupin podle vzrůstajícího VID, přičemž předměty patřící do první skupiny předmětů s nejnižšími VID nemají žádnou prerekvizitu, předměty z druhé skupiny mají za prerekvizity předměty z 1. skupiny atd.
Dále provádějí náhodný zápis předmětů studentům a přesun takto zapsaných předmětů mezi již absolvované včetně náhodného přiřazení známky. Přitom je každému studentovi vygenerován náhodný počet studií a zapsané předměty se rozdělí do těchto studií. Všechny takto vygenerované záznamy se vkládají prostřednictvím referencí na dané předměty.
Ve čtvrté fázi byly vytvořené testovací procedury, jak pro relační tak pro OR verzi.
Ty jsou vytvořené jako uložené procedury a pro relační verzi navíc i v rámci balíku. Pokrývají ukázku implementace použití objektů, referencí, hnízděných a vícenásobně hnízděných tabulek, porovnání provádění dotazů s vytvořenými indexy resp. bez nich, dotazy s agregačními funkcemi. Dotazy byly, pokud to šlo, prováděné pro různé zátěže – např. pro předměty s více a méně prerekvizitami. U některých problémovějších dotazů bylo vytvořeno více implementací.
V páté fázi probíhal sběr statistických údajů získaných při provádění jednotlivých procedur obou verzí. K tomuto sběru byly použité následující statistické prostředky:

- **autotrace**

  Výstup funkce autotrace poskytuje důležité informace o tom, co skutečně provedení dotazu vyžaduje.
  Výstup může mít dvě části: sestavu plánu dotazu a statistické informace. Ty jsou uvedeny v případě provádění dotazu optimalizátorem nákladů (CBO), kdy jsou na konci každého kroku uvedeny následující informace: [5]
  - **Cost (náklady)** - náklady přiřazené každému kroku plánu dotazu optimalizátorem CBO. Optimalizátor generuje pro jeden dotaz mnoho různých

průběhů výpočtů/plánů a každému přiřadí určitou hodnotu představující náklady na konkrétní průběh. Plán dotazu s nejnižšími náklady je vyhodnocen jako nejúspěšnější.

- **Card (mohutnost)** - představuje očekávaný počet výstupních řádků z daného kroku plánu dotazu.
- **Bytes (Bajtů)** - velikost dat v bajtech vrácená jednotlivými kroky plánu dle předpokladu optimalizátoru CBO. Hodnota je závislá na počtu řádků (Card) a očekávané šířce řádků.
- **Recursive calls** (rekurzivní volání) - Počet provedených příkazů SQL nutných k provedení uživatelského příkazu SQL. Statistické údaje o rekurzivním volání se vztahují k příkazům SQL, které jsou prováděny v zastoupení uživatele a které jsou vedlejším efektem jiného SQL příkazu.
- **Db block gets** (načtené DB bloky) - Celkový počet bloků načtených v aktuálním režimu z vyrovnávací mezipaměti. Bloky mohou být databázi Oracle načteny a používány dvěma způsoby: aktuálně a konzistentně. Načtení v aktuálním režimu znamená okamžité načtení bloku ve stávajícím stavu. Konzistentní načtení představuje načtení bloků z vyrovnávací mezipaměti v režimu "konzistentního čtení" a může zahrnovat čtení umožňující vrácení akce - příkaz UNDO(návratové segmenty). dotaz bude obvykle provádět "konzistentní načítání".
- **Consistent gets** (konzistentně načítané informace) - Počet konzistentních načítání ve vyrovnávací paměti vyžadovaných pro blok; konzistentní načítání může vyžadovat načtení informací potřebných k vrácení akce(undo - rollback), tato načítání budou počítána také.
- **Physical reads** (fyzická načtení) - Počet fyzických načtení z datových souborů do vyrovnávací mezipaměti. Statistický údaj je měřitkem skutečného počtu V/V operací neboli fyzických V/V operací, které dotaz provedl. Při fyzickém načtení tabulky nebo dat indexu je blok umístěn do vyrovnávací mezipaměti. Poté je provedena logická V/V operace pro načtení bloku. Z toho důvodu většina fyzických načítání následuje bezprostředně po logické V/V operaci.
- **Redo size** (počet opakování) - Celkový počet opakovaných akcí generovaných v bajtech během provádění příkazu. Statistický údaj udává, kolik opakování během svého zpracování příkaz vygeneroval. Tato hodnota je velmi užitečná při posuzování výkonnosti rozsáhlých hromadných operací. Nejčastěji se stává důležitou u vkládání nebo u příkazů CREATE TABLE AS SELECT.
- **Bytes sent via SQL*NET to client** (Bajty odeslané klientovi přes SQL*Net) - Celkový počet bajtů odeslaných klientovi ze serveru.
- **Bytes recieved via SQL*NET from client** (Bajty přijaté od klienta přes SQL*Net) - Celkový počet bajtů přijatých od klienta.
- **SQL*NET roundtrips to/from client** (Přenosy programu SQL*Net od/ke klientovi) - Celkový počet zpráv programu SQL*Net odeslaných klientovi a přijatých od klienta, číslo zahrnuje přenosy týkající se načítání ze sady výsledků s více řádky.
- **Sorts (memory)** (Třídění v paměti) - Počet provedených setřídění v paměti relace uživatele (oblast třídění), řízeno prostřednictvím parametru databáze sort_area.
- **Sorts (disk)** (Třídění na disku) - Počet setřídění na disku(dočasný tabulkový prostor), protože při třídění došlo k překročení velikosti oblasti třídění uživatele.
- **Rows processed** (Zpracované řádky) - Řádky zpracované úpravami nebo vrácené z příkazu SELECT.

Db block gets a consistent gets představují nejdůležitější části sestavy AUTOTRACE. Tyto parametry reprezentují logické V/V operace - kolikrát bylo nutné blokovat vyrovnávací paměť za účelem jejího prozkoumání. Čím méně je blokovacích operací, tím lépe.
Pozornost je nutno věnovat i hodnotám u SQL*NET. Cílem by měla být minimalizace dat, která jsou přijímaná od serveru. Řídit to lze např. tak, že se v dotazu vybírají pouze ty sloupce, které jsou důležité. Tím dojde ke snížení zatížení sítě nepotřebnýma daty a dále je spotřebováno daleko méně paměti RAM, což může radikálně ovlivnit efektivitu plánů dotazů. Počet přenosů lze ovlivnit např. nastavením velikosti pole - set arraysize velkost.
Pokud hodnoty Cost, Card a Bytes ve výstupu uvedeny nejsou, lze jednoznačně říci, že byl dotaz spuštěn optimalizátorem RBO.

- **trace**

Databáze Oracle umožňuje zapnout možnost podrobného trasování. Po zapnutí trasování zaznamená databáze veškeré příkazy SQL a volání PL/SQL nejvyšší úrovně, které aplikace použije, do trasovacího souboru v serveru. Trasovací soubor obsahuje kromě příkazů SQL a volání PL/SQL informace o časování, informace o událostech čekání (způsobujících zpomalení systému), počtu provedených logických V/V operací a fyzických V/V operací, o časování procesoru, o skutečně uplynulém čase a o počtu zpracovaných řádků. Obsahuje rovněž plány dotazu s počty řádků a další údaje. Tento trasovací soubor je pro člověka jen velmi obtížně čitelný. Do textového, uživatelsky přívětivého formátu je možné jej převést nástrojem tkprof.
Výstupem jsou statistické údaje týkající se dotazu zpracovaného databázi:

  - **Parse (analýza)** - fáze odpovídající nalezení dotazu systémem Oracle ve sdíleném fondu (soft parse) nebo vytvoření nového plánu pro dotaz (hard parse).
  - **Execute (provedení)** - fáze představující činnost databáze Oracle související s příkazy OPEN a EXECUTE. V mnoha případech bude pro příkaz SELECT prázdná.
  - **Fetch (načtení)** - u příkazu SELECT bude tato fáze představovat nejvíce provedených činností, v případě příkazu UPDATE nevykáže žádnou činnost (v případě aktualizace nedochází k načítání).

Pro každou fázi zpracovávaného příkazu jsou uvedené tyto údaje.
Horní část sestavy obsahuje následující záhlaví:

  - **Count (počet)** - číslo udává kolikrát byla daná fáze dotazu provedena. U správně vytvořené aplikace bude u všech příkazů SQL počet analytických fází (Parse) roven hodnotě 1 a počet fází provedení (Execute) roven hodnotě 1 či větší. Je-li to možné, analýza by neměla proběhnout více než jedenkrát.
  - **CPU (procesor)** - čas procesoru vynaložený na danou fázi příkazu v tisícinách sekundy.
  - **Elapsed (uplynulý)** - skutečný čas vynaložený na danou fázi. Je-li uplynulý čas mnohem delší než čas procesoru, znamená to, že určitý čas byl vynaložen na čekání. V databázi Oracle9i lze pomocí nástroje TKPROF snadno zjistit příčinu. V dolní části sestavy je uvedeno, že se jednalo o "db file scattered read" - čekání na dokončení fyzické V/V operace.
  - **Disk** - počet provedených fyzických V/V operací během dané fáze dotazu.
  - **Query (dotaz)** - počet provedených logických V/V operací za účelem obnovení bloků konzistentního režimu. Jedná se o bloky, které mohly být zrekonstruovány z návratových segmentů, byly by zobrazeny, pokud by existovaly při spuštění dotazu. Všechny fyzické V/V operace mají obecně za

následek logické V/V operace. Ve většině případů platí, že počet logických V/V operací převyšuje počet fyzických V/V operací. Na druhou stranu přímé čtení a zápis u dočasného prostoru toto pravidlo porušuje a mohou existovat fyzické V/V operace, které nebudou převedeny na logické V/V operace.

- **Current (aktuální)** - počet provedených logických V/V operací za účelem obnovení bloků v tomto okamžiku. Nejčastěji bude tento počet zobrazen během úpravy operací DML, např. u aktualizací nebo mazání. Blok musí být obnoven v aktuálním režimu, aby došlo ke zpracování úpravy, což je opakem situace, kdy se dotazujeme na tabulku a kdy databáze obnoví bloky v okamžiku spuštění dotazu.
- **Rows (řádky)** - počet řádků zpracovaných nebo ovlivněných danou fází. Během úpravy bude hodnota Rows zobrazena ve fázi Execute. Během dotazu SELECT se tato hodnota zobrazí ve fázi Fetch.

- **profiler**

Pomocí tohoto nástroje je možné určit část kódu, na jejíž ladění je třeba maximálně zaměřit pozornost [8]. Hlavní důvody pro použití profileru jsou tyto:

- testování kódu s cílem ujistit se, že testovací případy jsou kódem 100% pokryty
- ladění algoritmů – jelikož největší přínos poskytuje porovnání maximálně odladěných kódů

Pomocí profileru je možné snadno určit nejvhodnější procedurální algoritmus. Protože profiler není standardní součástí systému, je nutné ho nejdřív nainstalovat a vytvořit potřebné tabulky, se kterými pracuje.

Tři vytvořené tabulky obsahují důležité informace zachycující výkonnost PL/SQL.

- Tabulka **plsql_profiler_runs** obsahuje informace týkající se sezení - kdy bylo měření spuštěno, jméno uživatele, jak dlouho měření probíhalo. Tabulka obsahuje následující sloupce:
  - runid - jednoznačný identifikátor pro každé spuštění profileru
  - related_run – identifikátor odpovídajícího běhu, pomocí něhož může být volán programátorem
  - run_owner - jméno uživatele provádějícího měření
  - run_date - datum spuštění měření
  - run_comment – uživatelská poznámka k měření
  - run_total_time – celkový čas potřebný na provedení tohoto měření

- Tabulka **plsql_profiler_units** definuje PL/SQL komponentu, které byly prováděné během měření. Obsahuje následující sloupce:
  - runid - odkazy na plsql_profiler_runs (runid).
  - unit_number - interně generované číslo komponenty
  - unit_type - typ komponenty (balík, procedura, ...)
  - unit_owner - jméno vlastníka komponenty
  - unit_name - jméno komponenty
  - unit_timestamp – čas vytvoření komponenty
  - total_time – celkový čas zpracování této komponenty

- Tabulka **plsql_profiler_data** obsahuje skutečně naměřené hodnoty. Obsahuje statistiky provádění pro každý řádek kódu obsaženého v PL/SQL skriptu. Má tyto sloupce:

- runid - primární klíč
- unit_number - primární klíč pro komponentu
- line# - číslo řádky v komponentě
- total_occur - počet, kolikrát byla řádka celkově prováděna
- total_time - celkový čas provádění dané řádky v nanosekundách
- min_time - minimální čas pro provádění dané řádky v nanosekundách
- max_time - maximální čas pro provádění dané řádky v nanosekundách
- spare1, spare2, spare3, spare4 - nepoužito

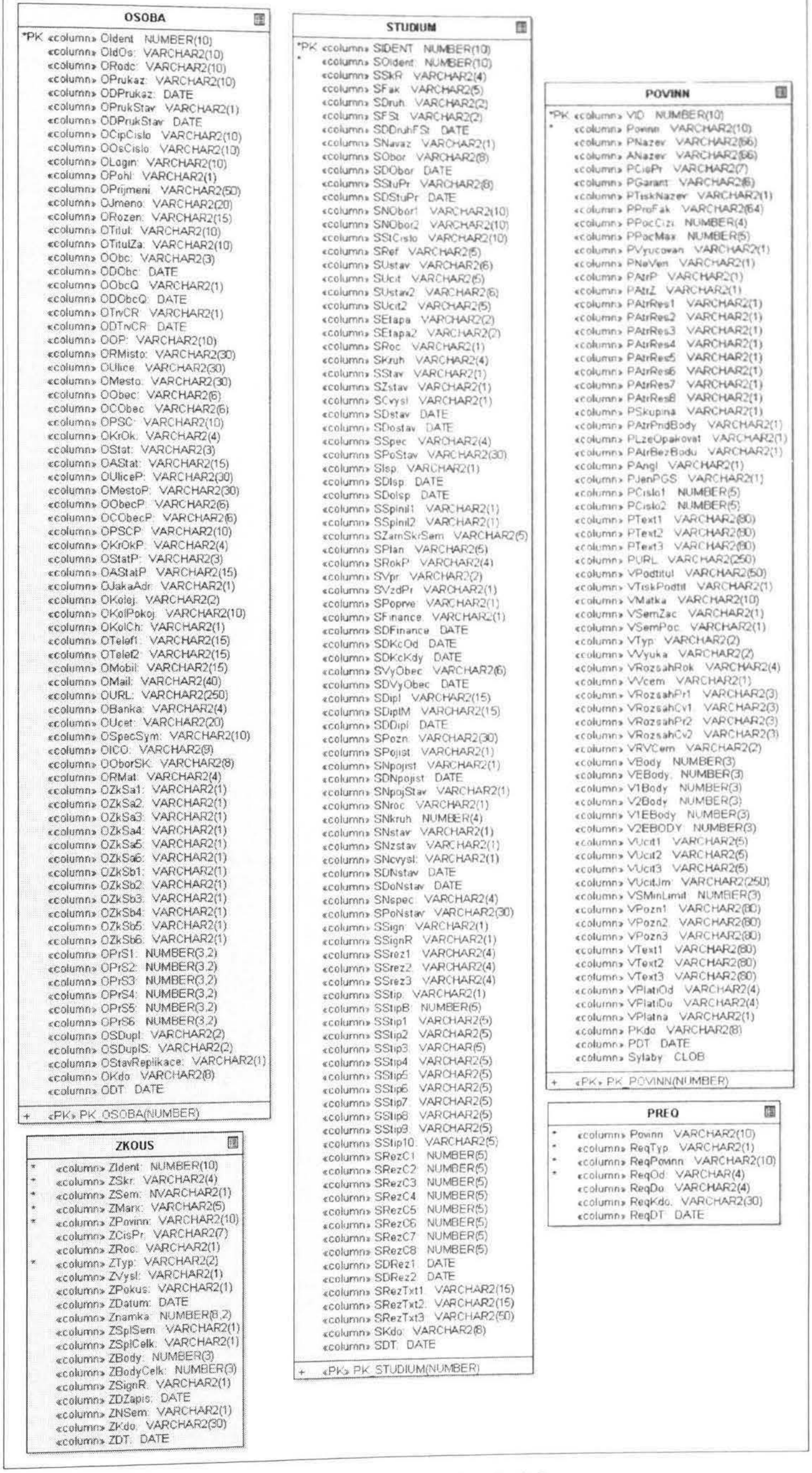
- **runstat**

  Jedná se o nástroj vyvinutý ke srovnání metod a zjištění té nejlepší z nich. Měří tři klíčové hodnoty:

  - skutečný čas nebo uplynulý čas
  - statistické údaje systému - vedle sebe zobrazuje čas, po který jednotlivým přístupům trvalo provedení určité operace a rozdíl mezi dvěma přístupy
  - blokování - klíčový výstupní údaj sestavy. Jedná se o typ slabého uzamknutí. Uzamknutí jsou serializačními zařízeními. Serializační zařízení brání paralelnímu zpracování. Přitom věci bránící paralelnímu zpracování jsou méně škálovatelné, mohou podporovat méně uživatelů a vyžadují více prostředků. Tedy čím méně blokování budou jednotlivé přístupy způsobovat, tím lépe.

Sběr statistik byl proveden jak pro jednotlivé dotazy zvlášť, tak pro vícenásobné spuštění každého z dotazů. Získání statistických hodnot pro všechny dotazy a všechny statistické nástroje probíhal stejným způsobem, s tím, že se u každého typu statistiky provedla inicializace pro tuto konkrétní statistiku. Vlastní měření bylo prováděno pro každý dotaz prostřednictvím pomocných procedur (pro každý typ statistiky jedna), kdy se po zavolání odpovídající pomocné procedury inicializuje daná statistika a následně je volán požadovaný dotaz. Takto získané hodnoty jsou (v případě nástroje trace) dále zpracovány pomocí nástroje tkprof. V případě hromadného měření jsou postupně volány všechny dotazy s předchozí inicializací statistiky.

Obrázek 7.3 obsahuje všechny atributy relačních tabulek.



Obr. 7.3 Atributy relačních tabulek

7.3 OR vs. relační dotazy – výhody a nevýhody

Výsledky uvedené v této kapitole byly získány na základě provádění dotazů obou verzí (relační – název procedury končící sufixem R a OR verze s názvem funkce končícím OR) z následujícího seznamu:

1. **returnAbsolvovane** - vrátí předměty, které má daný student již absolvované.

```
PROCEDURE returnAbsolvovaneR(in_OIdent NUMBER), přičemž vrací
výsledek prostřednictvím pomocné tabulky tab_POVINN

FUNCTION returnAbsolvovaneOR(in_OIdent NUMBER) RETURN
NT_Predmet
```

   Zpřístupnění všech hodnot hnízděné tabulky.
2. **avgZnamkyStudents** - průměr všech studentů ze všech studií.

```
PROCEDURE avgZnamkyStudentsR(), přičemž vrací výsledek
prostřednictvím pomocné tabulky tab_studentPrumer

FUNCTION avgZnamkyStudentsOR(a number) RETURN
NT_studentPrumer
```

   Hromadně prováděný dotaz z bodu 12., tedy jsou procházeny všechny hnízděné tabulky všech studentů.
3. **returnSylabysNear** - OPERATOR NEAR - vrátí sylaby, ve kterých se zadané termy vyskytují v maximálně zadané vzdálenosti.

```
PROCEDURE returnSylabysNearR(in_pocet number), přičemž vrací
výsledek prostřednictvím tabulky sylabyNearVIDR

FUNCTION returnSylabysNearOR(in_vids V_VIDS, in_pocet NUMBER)
```

   Ukázka práce s textem pomocí Oracle Text (viz kap. 5.5.2) za použití operátoru NEAR. Jelikož relační verze tento operátor neobsahuje, bylo nezbytné jej nasimulovat.
4. **countStudentExam** - vrátí počet studentů, kteří mají absolvovanou danou zkoušku historicky.

```
FUNCTION countStudentExamR(in_pov VARCHAR2) RETURN NUMBER
FUNCTION countStudentExamOR(in_pov VARCHAR2) RETURN NUMBER
```

   Při provádění dotazu je nezbytné u OR verze projít všechny záznamy všech hnízděných tabulek odkazujících se na absolvované předměty jednotlivých studentů a v nich vyhledat požadovaný předmět.
   V relační verzi stačí projít tabulku studií všech studentů spojenou s tabulkou, ve které jsou všechny absolvované i zapsané předměty studentů.
5. **plan** - vytvoří studijní plán s maximálním součtem bodů předmětů, obsahující požadované předměty tak, že zohlední zaměnitelné a neslučitelné předměty a již absolvované předměty studenta.

```
PROCEDURE planR(in_OIdent NUMBER, in_vids V_VID), přičemž
vrací výstup prostřednictvím tabulky tempRetez

FUNCTION planOR(in_OIdent NUMBER, in_vids IN V_VID, in_typ
NUMBER) RETURN NT_Predmet
```

   Ukázka rekurzívního volání a předávání parametrů při procházení stromu všech možností při tvorbě studijního plánu.
6. **nextPredmety** - pro daný předmět vrátí všechny předměty, které mají daný předmět jako prerekvizitu.

```
PROCEDURE nextPredmetyR(in_povinn VARCHAR2, in_level NUMBER),
přičemž vrací výstup prostřednictvím tabulky nextPR

FUNCTION nextPredmetyOR(in_vid NUMBER, in_level NUMBER)
RETURN NT_Predmet
```

Vícenásobný přístup k datům hnízděných tabulek, jejichž záznamy se ukládají do pomocných indexovaných tabulek, uchovávajících výslednou množinu předmětů.

7. **returnCountExams** - vrátí počty absolvovaných předmětů jednotlivých studentů.

```
PROCEDURE returnCountExamsR(), přičemž vrací výstup
prostřednictvím tabulky tab_countExams

FUNCTION returnCountExamsOR(a number) RETURN NT_countExams
```

Použití agregační funkce na všechny hnízděné tabulky s absolvovanými předměty.

8. **sameExams** - vrátí největší skupinu studentů, kteří mají alespoň požadovaný počet stejných zkoušek.

```
PROCEDURE sameExamsR(in_n NUMBER), přičemž vrací výstup
prostřednictvím tabulek maxTempPov, maxTempSt

PROCEDURE sameExamsOR(in_n NUMBER, out_maxTempPov OUT
NT_Povinn, out_maxTempSt OUT NT_OIdents)
```

Předávání parametrů prostřednictvím lokálních kolekcí mezi procedurami a jejich využívání. V relační verzi lze toto předávání uskutečnit pouze prostřednictvím hodnot v pomocných tabulkách vytvořených pro tento účel. Dále je zde vyzkoušena efektivita těchto struktur v prostředí rekurzivně se volajících procedur.

9. **returnAllAbsolvovane** - vrátí všechny absolvované předměty všech studentů.

```
PROCEDURE returnAllAbsolvovaneR(), přičemž vrací výstup
prostřednictvím tabulky tab_POVINN

FUNCTION returnAllAbsolvovaneOR(a number) RETURN NT_Predmet
```

Zpřístupnění všech hodnot všech hnízděných tabulek s absolvovanými předměty.

10. **avgZnamkyObor** - průměr známek všech studentů jednotlivých oborů.

```
PROCEDURE avgZnamkyOborR(), přičemž vrací výstup
prostřednictvím tabulky tab_oborPrumer

FUNCTION avgZnamkyOborOR(a number) RETURN nt_oborPrumer
```

K provedení dotazu je nutné přistupovat ke dvěma hnízděným tabulkám, přičemž jedna je obsažena ve sloupci druhé. Je nezbytné projít všemi daty vnořené hnízděné tabulky (reprezentujícími známky studentů), a ty následně seskupit na základě hodnoty sloupce v nadřazené hnízděné tabulky (v tomto případě podle oboru).

11. **avgZnamkyOborRok** - průměr známek všech studentů jednotlivých oborů po letech.

```
PROCEDURE avgZnamkyOborRokR(), přičemž vrací výstup
prostřednictvím tabulky aZORR

FUNCTION avgZnamkyOborRokOR(a number) RETURN aZOR
```

Jedná se o podobný typ dotazu jako v bodě 10., přičemž seskupování probíhá podle dvou hodnot nadřazené hnízděné tabulky.

12. **avgZnamky** - průměr všech známek daného studenta.

```
FUNCTION avgZnamkyR(in_OIdent NUMBER) RETURN NUMBER

FUNCTION avgZnamkyOR(in_OIdent NUMBER) RETURN NUMBER
```

Ukázka použití agregační funkce nad daty uloženými v hnízděné tabulce pro jednoho studenta.

13. **bestPrumerObor** - vrátí studenty s nejlepšími průměry v jednotlivých oborech.

```
PROCEDURE bestPrumerOborR(), přičemž vrací výstup
prostřednictvím tabulky bPR

FUNCTION bestPrumerOborOR(a number) RETURN bPOR
```

Ekvivalentní dotaz k dotazu v bodě 10. s použitím jiného typu agregační funkce.

14. **compStudent** - Porovnání dvou studentů na základě jejich známek. V případě, že nemají ani jeden společný předmět, na základě počtu zapsaných předmětů.

```
FUNCTION compStudentR(in_OIdent1 NUMBER, in_OIdent2 NUMBER)
RETURN NUMBER

FUNCTION compStudentOR(in_OIdent1 NUMBER, in_OIdent2 NUMBER)
RETURN NUMBER
```

K provedení dotazu v OR verzi je vytvořen porovnávací operátor (viz kap. 4.8 bod 1)), jehož úkolem je provést porovnání známek totožných předmětů porovnávaných studentů. V relační verzi je porovnání implementováno klasickou funkcí.

15. **existExam** - má zadaný student absolvovanou požadovanou povinnost?

```
FUNCTION existExamR(in_oident NUMBER, in_pov VARCHAR2) RETURN
NUMBER

FUNCTION existExamOR(in_OIdent NUMBER, in_pov VARCHAR2)
RETURN NUMBER
```

Vyhledání požadovaného záznamu reprezentovaného pomocí odkazu v hnízděné tabulce s využitím operátoru EXISTS.

16. **matchPredmet** - porovná dva předměty na základě počtu všech (i vnořených) prerekvizit.

```
FUNCTION matchPredmetR(in_VID1 NUMBER, in_VID2 NUMBER) RETURN
NUMBER

FUNCTION matchPredmetOR(in_VID1 NUMBER, in_VID2 NUMBER)
RETURN NUMBER
```

Ukázka implementace operátoru porovnání dvou objektů uživatelsky definovaného typu v OR verzi (viz kap.4.8 bod 1)). V relační verzi realizován klasickou funkcí.

17. **moreExams** - vypíše studenty, kteří mají absolvováno alespoň požadovaný počet předmětů.

```
PROCEDURE moreExamsR(in_n NUMBER), přičemž vrací výstup
prostřednictvím tabulky studentsOID

FUNCTION moreExamsOR(in_n NUMBER) RETURN studentsOIDT
```

Použití agregační funkce nad hnízděnou tabulkou v OR verzi.

18. **numAbsPredmet** - vypíše pro každý předmět počet studentů, kteří ho mají absolvovaný.

```
PROCEDURE numAbsPredmetR(), přičemž vrací vystup
prostřednictvím tabulky nPR

FUNCTION numAbsPredmetOR(a number) RETURN nP1OR
```

Použití agregační funkce na záznamy obsahující požadovaný záznam v hnízděných tabulkách druhé úrovně.

19. **prumerZnamkyMesta** - seřadí města podle průměru všech studentů z těchto měst.

```
PROCEDURE prumerZnamkyMestaR(), přičemž vrací výstup
prostřednictvím tabulky tab_mestoPrumer
```

```
FUNCTION prumerZnamkyMestaOR(a number) RETURN NT_mestoPrumer
```

Použití agregační funkce na vybrané hnízděné tabulky, jejichž hodnoty jsou seskupeny podle hodnoty sloupce v rodičovském objektu.

20. **returnDeepPreq** - zobrazí hierarchii prerekvizit požadovaného předmětu.

```
PROCEDURE returnDeepPreqR(in_vid NUMBER, in_level NUMBER,
out_level OUT NUMBER), přičemž vrací výstup prostřednictvím
tabulky deepQR

FUNCTION returnDeepPreqOR(in_vid NUMBER, in_level NUMBER,
out_level OUT NUMBER) RETURN NT_Predmet
```

Ukázka zanořování realizovaném prostřednictvím hnízděných tabulek.

21. **returnMaxAbsolvovane** - vrátí seznam studentů, kteří mají absolvováno nejvíce předmětů.

```
PROCEDURE returnMaxAbsolvovaneR(), přičemž vrací výstup
prostřednictvím tabulky t_OIdent

FUNCTION returnMaxAbsolvovaneOR() RETURN NT_StudentRef
```

Použití agregační funkce na hodnoty všech hnízděných tabulek druhé úrovně s absolvovanými studenty plus setřídění výsledku.

22. **returnPocetStudium** - vrátí počty studií jednotlivých studentů.

```
PROCEDURE returnPocetStudiumR(), přičemž vrací výstup
prostřednictvím tabulky tab_pocetStudium

FUNCTION returnPocetStudiumOR() RETURN NT_pocetStudium
```

Použití agregační funkce na hodnoty všech hnízděných tabulek první úrovně.

23. **returnPrumerSS** - vrátí průměr známek studentů ze střední školy.

```
PROCEDURE returnPrumerSSR(), přičemž vrací výstup
prostřednictvím tabulky tab_studentPrumerSS

FUNCTION returnPrumerSSOR() RETURN sZSS
```

Ukázka implementace pole VARRAY.

24. **returnSamePreq** - ke každému předmětu vrátí seznam předmětů, které ho mají jako prerekvizitu, jedná-li se alespoň o skupinku dvou předmětů.

```
PROCEDURE returnSamePreqR(in_vid1 NUMBER, in_vid2 NUMBER),
přičemž vrací výstup prostřednictvím tabulky t_samePreqR

FUNCTION returnSamePreqOR(in_vid1 NUMBER, in_vid2 NUMBER)
RETURN NT_nextP1OR
```

Rekurzivní práce s hnízděnými tabulkami obsahujícími reference na skutečné předměty.

25. **returnStudent** - vrátí studenty, kteří mají absolvovaný daný předmět.

```
PROCEDURE returnStudentR(in_pov VARCHAR2), přičemž vrací
výstup prostřednictvím tabulky studentsOID

FUNCTION returnStudentOR(in_VID NUMBER) RETURN NT_Osoba
```

Vyhledání požadovaného záznamu ve všech hnízděných tabulkách druhé úrovně.

26. **returnSylabys** - vrátí seznam předmětů, u kterých se v sylabech nachází daný term.

```
PROCEDURE returnsylabysR(in_term VARCHAR2), přičemž vrací
výstup prostřednictvím tabulky sylabyTR

FUNCTION returnSylabysOR(in_term VARCHAR2) RETURN NT_REFP
```

Ukázka implementace vyhledávání v textu, kdy je u relační verze použito operátoru LIKE a u OR verze operátoru CONTAINS.

27. **sumAllBody** - vrací součet bodů všech absolvovaných předmětů všech studentů.

```
PROCEDURE sumAllBodyR(), přičemž vrací výstup prostřednictvím
tabulky tab_sumAllBody

FUNCTION sumAllBodyOR(a NUMBER) RETURN NT_sumAllBody
```

Hromadné použití agregační funkce na hodnoty hnízděných tabulek druhé úrovně.

28. **sumBody** - vrací součet bodů všech absolvovaných předmětů studenta.

```
FUNCTION sumBodyR(in_oident NUMBER) RETURN NUMBER

FUNCTION sumBodyOR(in_oident NUMBER) RETURN NUMBER
```

Použití agregační funkce na hodnoty hnízděné tabulky druhé úrovně.

V následujících podkapitolách jsou uvedeny hlavní rozdíly mezi relační a OR verzí, které vyplynuly z výsledků, získaných prostřednictvím nástrojů uvedených v předchozí kapitole (aby měli naměřené hodnoty větší výpovědnou hodnotu, byly dotazy prováděné opakovaně – to udává číslo v tabulce časů). Z plánů vyhodnocení uvedených u jednotlivých dotazech plyne, že OR systém používá při zpracování dotazů stejných technik jako relační systém. Tzn. selekce přístupových cest u OR dotazů funguje stejně jako u relačních. Navíc umí OR optimalizátor umět ohodnotit operace nad novými datovými typy a zahrnout je do celkového plánu.

7.3.1 Z časového hlediska efektivnější OR verze.

Asi v půlce sestavených dotazů je na tom z hlediska časové náročnosti lépe OR verze. Jedná se o dotazy, ve kterých jsou přímo využívána data z hnízděných tabulek. Časová úspora plyne z toho, že není nutné prohlížet tabulky obsahující hodnoty týkající se všech studentů resp. předmětů, ale stačí přistupovat pouze k záznamům, kterých se dotaz přímo týká.
Tento časový neprospěch pro relační verzi jde snížit vytvořením vhodných indexů nad relačními tabulkami (v našem případě např. vytvořením indexů nad tabulkou STUDIUM (SOIdent) a tabulkou ZKOUS (ZIdent), čímž se zefektivní přístup k hodnotám v těchto tabulkách., nicméně lepších časů než u OR verze se tím nedosáhne.
Do této kategorie patří dotazy číslo:2, 3, 5, 6, 8, 9, 12, 18, 20, 22, 24, 25, 26.
Jako nejzajímavější se jeví dotazy číslo 3 a 12, jejichž plány provádění, jak pro relační tak pro OR verzi, včetně statistik ukazují následující výpisy, kde je v plánu dotazu vidět hašované spojení tabulek (viz kap.4.7.1):
Do této kategorie patří i dotazy pracující s textem - např. vyhledání řetězce v textu.

7.3.1.1 returnSylabysNear

Textové vyhledávání je v OR verzi naimplementováno efektivněji než je tomu u relační verze, kdy bylo nutné, z důvodu neexistence odpovídající funkcionality, daný OR dotaz nasimulovat. Tak tomu bylo v případě simulace operátoru NEAR, prostřednictvím kterého je možné vyhledat texty, obsahující požadované termy, mezi kterými je určen maximální počet jiných termů.
Vlastní simulace pro relační verzi probíhala způsobem, kdy byly nejdřív vybrány předměty obsahující ve svých sylabech všechny požadované termy, přičemž byly tyto předměty z výsledné skupiny postupně odstraňovány v případech, kdy se ukázalo, že vzdálenost mezi požadovanými termy je větší než maximální (počet slov mezi termy byla určována na základě počtu mezer mezi nimi). Všechny požadované termy jsou přitom uloženy v jiné pomocné tabulce.

verze	čas - 500x
R	00:57:36.94
OR	00:00:08.36

returnSylabysNearR(in_pocet)
vzdálenost
space RETURN NUMBER
slovaMezi(in_term1, in_term2) RETURN NUMBER
BEGIN
vrátí počet slov mezi vstupními termy v probíraném sylabu
END;
BEGIN
pro všechny kombinace termů určí jejich vzdálenost v probíraném sylabu
jeli vzdálenost větši než maximální povolená, vrať 0, jinak vrať 1
END;
BEGIN
projížděj všechny vybrané předměty v pomocné tabulce a odstraňuj ty, pro které
neplatí podmínka na maximální vzdálenost zadaných termů
(space=0)
END;
BEGIN
vložení identifikátoru předmětů, které obsahují ve svých sylabech
zadané termy, do pomocné tabulky
voláni podprocedury vzdálenost
END;

(returnSylabysNear - OR)

```
SELECT CAST(MULTISET(SELECT REF(p)
FROM predmet p
WHERE CONTAINS(p.sylaby, s)>0) AS NT_REFP)
INTO col FROM dual;
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=2022)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'PREDMET' (Cost=2 Card=1
          Bytes=2022)
   2    1     DOMAIN INDEX OF 'IDX_SYLABY' (Cost=0)

Statistics
----------------------------------------------------------
6241          recursive calls
0             db block gets
2916          consistent gets
168           physical reads
0             redo size
219           bytes sent via SQL*Net to client
368           bytes received via SQL*Net from client
1             SQL*Net roundtrips to/from client
146           sorts (memory)
0             sorts (disk)
0             rows processed
```

7.3.1.2 avgZnamky

verze	čas - 1000x
R	00:00:12.55
OR	00:00:01.16

(avgZnamky – relační)

```
SELECT avg(z.znamka)
FROM zkous z
INNER JOIN STUDIUM s ON z.ZIdent=s.SIdent
INNER JOIN OSOBA o ON s.soident=o.OIdent AND OIdent=10866 AND
z.Znamka>0;
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=9 Card=1 Bytes=16)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'ZKOUS' (Cost=2 Card=23970
Bytes=119850)
   3    2       NESTED LOOPS (Cost=9 Card=31 Bytes=496)
   4    3         NESTED LOOPS (Cost=3 Card=3 Bytes=33)
   5    4           INDEX (UNIQUE SCAN) OF 'SYS_C0056356' (UNIQUE) (Cost=1 Card=1
Bytes=4)
   6    4           TABLE ACCESS (BY INDEX ROWID) OF 'STUDIUM' (Cost=2 Card=3
Bytes=21)
   7    6             INDEX (RANGE SCAN) OF 'IDX_STUDIUM' (NON-UNIQUE) (Cost=1
Card=3)
   8    3         INDEX (RANGE SCAN) OF 'IDX_ZKOUS' (NON-UNIQUE) (Cost=1 Card=15)


Statistics
----------------------------------------------------------
0          recursive calls
0          db block gets
15         consistent gets
3          physical reads
0          redo size
384        bytes sent via SQL*Net to client
499        bytes received via SQL*Net from client
2          SQL*Net roundtrips to/from client
0          sorts (memory)
0          sorts (disk)
1          rows processed
```

(avgZnamky – OR)

```
SELECT avg(saz.znamka)
FROM student s, TABLE(s.Absolvovane) sa, TABLE(sa.Zkousky)
saz
WHERE s.oident=in_OIdent;
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 Card=1 Bytes=66)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'ABSOLVOVANE_ZKOUSKY_NT' (Cost=2
Card=35286 Bytes=811578)
   3    2       NESTED LOOPS (Cost=6 Card=10 Bytes=660)
   4    3         NESTED LOOPS (Cost=4 Card=1 Bytes=43)
   5    4           TABLE ACCESS (BY INDEX ROWID) OF 'STUDENT' (Cost=2
             Card=1 Bytes=23)
   6    5             INDEX (UNIQUE SCAN) OF 'SYS_C00200039' (UNIQUE) (Cost=1
Card=101)
   7    4           TABLE ACCESS (BY INDEX ROWID) OF 'ABSOLVOVANE_NT' (Cost=2 Card=1
Bytes=20)
   8    7             INDEX (RANGE SCAN) OF 'IDX_ABSOLVOVANE' (NON-UNIQUE) (Cost=1
Card=1)
   9    3         INDEX (RANGE SCAN) OF 'IDX_ABSOLVOVANE_ZKOUSKY' (NON-UNIQUE)
(Cost=1 Card=10)
```

```
Statistics
----------------------------------------------------------
14          recursive calls
0           db block gets
20          consistent gets
2           physical reads
0           redo size
386         bytes sent via SQL*Net to client
499         bytes received via SQL*Net from client
2           SQL*Net roundtrips to/from client
0           sorts (memory)
0           sorts (disk)
1           rows processed
```

7.3.2 Z časového hlediska efektivnější relační verze.

Ve druhé polovině jsou dotazy rychleji proveditelné v relační verzi. Zde jsou dotazy z časového hlediska nevhodné pro OR verzi, jelikož se v nich musíme opakovaně vnořovat k datům uloženým ve hnízděných tabulkách (řádky 4 a 5 v execution plan dotazu č. 4), přičemž u relační verze jsou tyto data přístupná z jedné tabulky 1. úrovně.
Do této kategorie patří tyto dotazy: 1, 4, 7, 10, 11, 13, 14, 15, 16, 17, 19, 21, 23, 27, 28.
Zajímavým reprezentantem patřícím do této kategorie je již zmiňovaný dotaz číslo 4, jehož výpis je uveden níže:

7.3.2.1 countStudentExam

verze	čas - 1x
R	00:00:00.37
OR	00:01:37.97

(countStudentExam - relační)

```
SELECT count(*)
INTO v
FROM ZKOUS Z
INNER JOIN STUDIUM S ON Z.ZIdent=S.SIdent AND
Z.ZPovinn=in_pov AND Z.Znamka>0
INNER JOIN OSOBA O ON S.SOIdent=O.OIdent;
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=49 Card=1 Bytes=22)
   1    0   SORT (AGGREGATE)
   2    1     NESTED LOOPS (Cost=49 Card=6 Bytes=132)
   3    2       NESTED LOOPS (Cost=49 Card=6 Bytes=108)
   4    3         TABLE ACCESS (FULL) OF 'ZKOUS' (Cost=43 Card=6 Bytes=66)
   5    3         TABLE ACCESS (BY INDEX ROWID) OF 'STUDIUM' (Cost=1 Card=3 Bytes=21)
   6    5           INDEX (UNIQUE SCAN) OF 'SYS_C00109700' (UNIQUE)
   7    2       INDEX (UNIQUE SCAN) OF 'SYS_C0056356' (UNIQUE)

Statistics
----------------------------------------------------------
0           recursive calls
0           db block gets
626         consistent gets
0           physical reads
0           redo size
378         bytes sent via SQL*Net to client
499         bytes received via SQL*Net from client
2           SQL*Net roundtrips to/from client
0           sorts (memory)
0           sorts (disk)
1           rows processed
```

(countStudentExam - OR)

```
SELECT count(*)
INTO v
FROM STUDENT s, TABLE(s.Absolvovane) sa, TABLE(sa.Zkousky)
saz WHERE saz.Predmet.Predmet.Povinn=in_pov;
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=43 Card=1 Bytes=57)
   1    0   SORT (AGGREGATE)
   2    1     NESTED LOOPS (Cost=43 Card=353 Bytes=20121)
   3    2       TABLE ACCESS (FULL) OF 'ABSOLVOVANE_ZKOUSKY_NT' (Cost=43 Card=353
Bytes=13061)
   4    2       INDEX (UNIQUE SCAN) OF 'SYS_C00200038' (UNIQUE)

Statistics
----------------------------------------------------------
47330          recursive calls
0              db block gets
47758          consistent gets
0              physical reads
0              redo size
378            bytes sent via SQL*Net to client
499            bytes received via SQL*Net from client
2              SQL*Net roundtrips to/from client
0              sorts (memory)
0              sorts (disk)
1              rows processed
```

7.3.3 Současně otevřené kurzory při rekurzivním volání podprocedur

Pro složitější dotazy využívající volání podprocedur, u kterých je výpočet založen na procházení všech možností do hloubky, je výhodnější použití OR verze, kdy u rekurzivního volání podprocedur nedochází na rozdíl od relační verze k hromadění většího počtu současně otevřených kurzorů a tím zaplnění vyrovnávací paměti.
Jako příklad může posloužit procedura pro stanovení studijního plánu studenta, ve kterém požaduje, aby byly obsaženy zadané předměty, tedy dotaz číslo 5, jehož výpis včetně použitých podprocedur je uveden níže.

7.3.3.1 plan

verze	čas - 10x
R	02:40:23.78
OR	01:04:22.26

planOR(in_OIdent NUMBER, in_vids IN V_VID, in_typ NUMBER) RETURN NT_Predmet
extrem() - nový extrém?
returnAbsCol() - vrátí indexovanou kolekci absolvovaných předmětů studenta
odstranAbsolvovane() - odstraní ze vstupní kolekce předmětů požadovaných k zápisu již absolvované předměty
makeNTVIDS() - vytvoř ze vstupní kolekce předmětů in_vids indexovanou kolekci
makeNTVIDS1() - vytvoř z hnízděné tabulky prerekvizit indexovanou kolekci
deepPlanRek() - projde všechny možnosti vytvoření studijního plánu, a to procházením do hloubky, kdy jsou nejdřív probírány všechny prerekvizity předmětu a následně jsou všechny prerekvizity nahrazovány k nim zaměnitelnými předměty
pridejNesluc(in_pred REF O_Predmet) - přidá ke kolekci neslučitelných předmětů předměty neslučitelné s předmětem in_pred
odeberNesluc(in_pred REF O_Predmet) - odebere z kolekce neslučitelných předmětů předměty neslučitelné s in_pred

planR(in_OIdent NUMBER, in_vids V_VID)
pridejNesluc(in_pov VARCHAR2) - přidá do pomocné tabulky neslučitelné předměty k předmětu in_pov
odeberNesluc(in_pov VARCHAR2) - odebere z pomocné tabulky neslučitelné předměty k předmětu in_pov
nesluc(in_pov VARCHAR2) RETURN NUMBER - je předmět in_pov neslučitelný s některým s předmětů v pomocné tabulce neslučitelných předmětů?
aplnAbsol(in_OIdent) - naplní pomocnou tabulku absolvovanými předměty daného studenta
bsol(in_pov VARCHAR2) - je daný předmět mezi absolvovanými?
odstranAbsolvovane() - odstraní z tabulky požadovaných předmětů předměty, které již má daný student absolvované
zamen1(in_n IN OUT NUMBER, in_pov VARCHAR2, povr VARCHAR2, shift VARCHAR2) - projde všechny zaměnitelné předměty k probíranému předmětu, přičemž mění prozatím vypočten studijní plán
proc1(in_n IN OUT NUMBER, in_pov VARCHAR2, povr VARCHAR2, shift varchar2) - projde všechny prerekvizity probíraného předmětu a prodlužuje studijní plán

7.3.4 Současné využití relačních i OR tabulek.

U některých typů dotazů se jeví jako nejvýhodnější využití jak OR tak relačních tabulek v jednom dotazu, jak tomu bylo u dotazu číslo 6.

7.3.4.1 nextPredmety

verze	čas - 1000x
R	00:28:28.72
R + OR	00:40:30.05

(nextPredmety)

```
FOR i IN (
    SELECT id+1 id, p.Povinn
    FROM PREQ p
    INNER JOIN TABLE(col) npor ON p.ReqPovinn=npor.Povinn
    WHERE p.ReqTyp='P'
    AND npor.id>=in_id)
LOOP
    col.EXTEND;
    col(col.COUNT):=o_nextP(i.id, i.Povinn);
END LOOP;
```

kdy je pro získání další skupiny prerekvizit použito již existující relační tabulky PREQ uchovávající vztahy mezi prerekvizitami všech předmětů a takto nově získané prerekvizity jsou ukládány do kolekce místo do tabulky, jak by tomu bylo u čistě relační verze, u které to znamená jisté časové zpoždění, jelikož by v takovém případě bylo nutné zapisovat do databázové tabulky místo lokální kolekce, jak je tomu u OR verze.

7.3.5 Předávání parametrů volaným funkcím.

OR verzi je výhodnější použít při potřebě předávat parametry volaným funkcím, kdy je možno využít konstruktu pole, zatímco u relační verze je nutné v takovém případě použít přídavnou pomocnou tabulku, se kterou je spojena další režie.
Do této kategorie patří následující dotazy: 3, 5, 15, z nichž nejzajímavější je již jednou zmiňovaný dotaz číslo 3.

returnSylabyNearR(viz kap.7.3.1)

7.3.6 Předávání získaných výsledků.

Zatímco výsledek získaný v OR verzích funkcí je možné předávat jak prostřednictvím tabulek tak i prostřednictvím kolekcí, při relačních procedurách lze využívat pouze předem vytvořených pomocných tabulek.
Do této kategorie náleží dotazy: 2, 3, 6, 7, 8, 9, 10, 11, 13, 14, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27.
Z nich je zajímavý dotaz číslo 7:

7.3.6.1 returnCountExams

verze	čas - 1000x
R	00:06:54.38
OR	00:21:09.21

(returnCountExams – relační)

```
INSERT INTO tab_countExams
SELECT s.SOIdent, count(z.ZPovinn)
FROM STUDIUM s, ZKOUS z
WHERE z.ZIdent=s.SIdent AND z.Znamka>0
GROUP BY s.SOIdent
ORDER BY s.soident;
```

```
Execution Plan
----------------------------------------------------------
   0      INSERT STATEMENT Optimizer=ALL_ROWS (Cost=96 Card=1076 Bytes=12912)
   1    0   SORT (GROUP BY) (Cost=96 Card=1076 Bytes=12912)
   2    1     HASH JOIN (Cost=53 Card=23970 Bytes=287640)
   3    2       TABLE ACCESS (FULL) OF 'STUDIUM' (Cost=9 Card=3000 Bytes=21000)
   4    2       TABLE ACCESS (FULL) OF 'ZKOUS' (Cost=43 Card=23970 Bytes=119850)

Statistics
----------------------------------------------------------
0          recursive calls
1090       db block gets
712        consistent gets
0          physical reads
136204     redo size
628        bytes sent via SQL*Net to client
669        bytes received via SQL*Net from client
3          SQL*Net roundtrips to/from client
2          sorts (memory)
0          sorts (disk)
1060       rows processed
```

(returnCountExams – OR)

```
SELECT o_countExams(REF(s), count(saz.OID))
BULK COLLECT INTO col
FROM STUDENT s, TABLE(s.Absolvovane) sa, TABLE(sa.Zkousky)
saz
GROUP BY REF(s);
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=352 Card=47590 Bytes=3331300)
   1    0   SORT (GROUP BY) (Cost=352 Card=47590 Bytes=3331300)
   2    1     HASH JOIN (Cost=66 Card=47590 Bytes=3331300)
   3    2       HASH JOIN (Cost=15 Card=4738 Bytes=222686)
   4    3         TABLE ACCESS (FULL) OF 'ABSOLVOVANE_NT' (Cost=6 Card=3513
Bytes=70260)
   5    3         TABLE ACCESS (FULL) OF 'STUDENT' (Cost=7 Card=4738 Bytes=127926)
   6    2       TABLE ACCESS (FULL) OF 'ABSOLVOVANE_ZKOUSKY_NT' (Cost=43 Card=35286
Bytes=811578)

Statistics
----------------------------------------------------------
36          recursive calls
0           db block gets
575         consistent gets
0           physical reads
0           redo size
129398      bytes sent via SQL*Net to client
1676        bytes received via SQL*Net from client
76          SQL*Net roundtrips to/from client
1           sorts (memory)
0           sorts (disk)
1060        rows processed
```

7.3.7 Omezený počet iterací.

V dotazech, u kterých bylo možné omezit počet iterací při výpočtu, se ukázalo, že zatímco při menším počtu opakování na tom byla z časového hlediska lépe relační verze, s přibývajícími iteracemi se to postupně otočilo ve prospěch OR verze.
Příkladem dotazu z této kategorie je dotaz číslo 8.

7.3.7.1 sameExams

Výše uvedenou skutečnost lze vysvětlit rychlejší prací s lokálními poli v OR verzi oproti práci s pomocnými tabulkami v relační verzi – např. podprocedura naplnSC() opakovaně plnící pomocné pole dvojic (student, zkouška) prostřednictvím hodnot z jiného pomocného pole stejného typu se všemi požadovanými hodnotami (viz tab. 7.1, která ukazuje celkový, maximální a minimální čas potřebný pro provedení daných dotazů). V relační verzi ekvivalentní podprocedury se přitom pracuje s relačními tabulkami, kdy je čas potřebný na její provedení o dva řády menší. To, že byla v prvních iteracích OR verze pomalejší lze přičíst na vrub nutnosti úvodního naplnění pomocného pole z OR tabulek – podprocedura naplnSCMem() – kdy je naopak rozdíl v času oproti relační verzi dva řády (viz tab. 7.1).

sameExamsOR(in_n NUMBER, out_maxTempPov OUT NT_Povinn, out_maxTempSt OUT NT_OIdents) - vypíše největší skupinu studentů, kteří mají alespoň in_n stejných zkoušek
    naplnSCMem() - naplní stálou kolekci dvojic (student, absolvovaný předmět) pro všechny studenty a jejich absolvované předměty
    naplnSC() - prostřednictvím stálé kolekce SCMem naplní ekvivalentní pracovnou kolekci
    Exams() RETURN VARCHAR2 - jejím výsledkem je jedna ze skupin studentů, kteří mají zapsán požadovaný počet předmětů
    BEGIN
        v iteracích volej funkci Exams(), vracející kód povinnosti, kterou má v dané iteraci absolvováno nejvíce studentů
        je-li vrácená skupinka studentů početnější než doposud největší, zapamatuj si nový extrém
    END;

UNIT_NAME - sameExams	TOTAL_TIME	MIN_TIME	MAX_TIME
R - napInTempZkousMem() INSERT INTO tempZkousMem SELECT S.SOIdent, Z.ZIdent, Z.ZPovinn, Z.Znamka FROM ZKOUS Z INNER JOIN STUDIUM S ON Z.ZIdent=S.SIdent AND Z.znamka>0;	2960798000	20000	2960778000
OR - napInSCMem() SELECT pomo42(s.OIdent, saz.Predmet.VID) BULK COLLECT INTO scMem FROM STUDENT s, TABLE(s.Absolvovane) sa, TABLE(sa.Zkousky) saz;	1.1785E+11	3000	1.1785E+11
R - napInTempZkous() INSERT INTO tempZkous SELECT tzm.OIdent, tzm.ZIdent, tzm.ZPovinn, tzm.Znamka FROM tempZkousMem tzm; PROFILER_SAMEEXAMSOR	2682410000	26000	2682339000
OR - napInSC() sc:=scMem;	60933000	101000	60832000

Tab. 7.1 Část výstupu profileru pro dotaz sameExams

7.3.8 Rekurzivní volání lokálních funkcí.

Velkou nevýhodou relační databáze oproti OR databázi je nemožnost využívání lokálních polí při rekurzivních volání funkcí. Ta je nutné simulovat prostřednictvím pomocných tabulek, kterých efektivnost je o mnoho nižší.
Patří sem již zmiňovaný dotaz číslo 8.

(sameExams – viz kap 7.3.7.1))

7.3.9 Víceuživatelské aplikace.

Na základě statistik získaných pomocí nástroje RUNSTAT plyne, že se OR verze dále víc hodí pro víceuživatelské aplikace (2/3 případů – tento poměr lze pro OR verzi ještě zlepšit zavedením vhodných indexů nad hnízděnými tabulkami, to jde samozřejmě provést i pro relační verzi, ale zlepšení není tak výrazné jako u OR verze). Neboli při provádění OR dotazů dochází k menšímu počtu blokování (zamykání) než u relačních ekvivalentních dotazů.
Při provádění DML operací nad hnízděnou tabulkou dochází k uzamčení rodičovského řádku. Proto je možné současně provést pouze jednu modifikaci dat v dané hnízděné tabulce a to i v případě, že by další modifikace probíhala nad jiným řádkem hnízděné tabulky. Na druhou stranu, když stačí, aby byl vícenásobný přístup umožněn jenom pro část dat hnízděné tabulky, je možné použít reference na tato data [9].
Tedy při nutnosti zajistit vícenásobný přístup je výhodné uložit takováto data do samostatné tabulky a odkazovat se na ni prostřednictvím reference z hnízděné tabulky.
Do této kategorie patří dotazy číslo: 1, 2, 6, 9, 14, 15, 17, 18, 19, 21, 22, 23, 24
Nejzajímavějším ze všech se jeví dotaz číslo 9.

7.3.9.1 returnAllAbsolvovane

Verze	čas - 1000x
R	00:19:25.93
OR	00:04:16.82

(returnAllAbsolvovane – relační)

```
INSERT INTO tab_Povinn
SELECT z.ZPovinn
FROM ZKOUS z
INNER JOIN STUDIUM s ON z.ZIdent=s.SIdent AND z.znamka>0;
```

```
Execution Plan
----------------------------------------------------------
   0      INSERT STATEMENT Optimizer=ALL_ROWS (Cost=43 Card=23970 Bytes=335580)
   1    0   NESTED LOOPS (Cost=43 Card=23970 Bytes=335580)
   2    1     TABLE ACCESS (FULL) OF 'ZKOUS' (Cost=43 Card=23970 Bytes=263670)
   3    1     INDEX (UNIQUE SCAN) OF 'SYS_C00109700' (UNIQUE)

Statistics
----------------------------------------------------------
          8          recursive calls
      17320  db block gets
      17562  consistent gets
          0          physical reads
    2172924  redo size
        623  bytes sent via SQL*Net to client
        612  bytes received via SQL*Net from client
          3          SQL*Net roundtrips to/from client
          1          sorts (memory)
          0          sorts (disk)
      16906  rows processed
```

(returnAllAbsolvovane – OR)

```
SELECT saz.Predmet
BULK COLLECT INTO col
FROM STUDENT s, TABLE(s.Absolvovane) sa, TABLE(sa.Zkousky)
saz;
```

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=43 Card=22283 Bytes=1270131)
   1    0   NESTED LOOPS (Cost=43 Card=22283 Bytes=1270131)
   2    1     TABLE ACCESS (FULL) OF 'ABSOLVOVANE_ZKOUSKY_NT' (Cost=43 Card=22283
Bytes=824471)
   3    1     INDEX (UNIQUE SCAN) OF 'SYS_C00200038' (UNIQUE)

Statistics
----------------------------------------------------------
         20  recursive calls
          0          db block gets
      19695  consistent gets
          0          physical reads
          0          redo size
     787134  bytes sent via SQL*Net to client
      13607  bytes received via SQL*Net from client
       1139  SQL*Net roundtrips to/from client
          2          sorts (memory)
          0          sorts (disk)
      16906  rows processed
```

7.3.10 Volitelný počet parametrů při volání procedur.

Časová náročnost se pro OR verzi výrazně zlepší u dotazů, u kterých je možné zadat různý počet vstupních parametrů (např. zapisované předměty). U takových dotazů je relační přístup zdlouhavější, jelikož není možné použít konstruktu pole, ale je nezbytné využít pomocné tabulky na předání parametrů, kterou je nutno před spuštěním vlastního dotazu nejdřív těmito parametry v samostatném skriptu naplnit.
V této kategorii jsou dotazy číslo: 3 a 5.

(returnSylabyNear – viz bod 7.3.1.1))

7.4 Naměřené hodnoty

7.4.1 Autotrace

Na základě měřených údajů byla sestavena následující tabulka: Naměřené hodnoty jsou uvedeny ve formě procentuálního vyjádření, přičemž jako základ pro 100% je zvolen údaj pro relační měření. Tabulka obsahuje průměrné naměřené hodnoty jednotlivých statistik nástroje autotrace (viz kap. 7.2) při provádění jednotlivých dotazů. Z tabulky plyne, že:

- největším přínosem pro OR verzi je menší počet celkového počtu bloků načtených v aktuálním režimu z vyrovnávací mezipaměti (db block gets).
- V případě dotazů v OR verzi je dále značně nižší počet konzistentních načítání ve vyrovnávací paměti vyžadovaných pro blok (consistent gets).
- Nezanedbatelný je také menší počet potřebných fyzických načtení z datových souborů do vyrovnávací mezipaměti (physical reads).
- Počet nutných setřídění v paměti relace uživatele (oblasti třídění) je pro OR verzi 43x větší než pro relační dotazy (sorts memory).
- Pokud jde o množství dat poslaných klientovi přes SQL*Net, vychází naměřené hodnoty jednoznačně v neprospěch relační verze.
- Naproti tomu u počtu provedených SQL příkazů nutných k provedení uživatelského příkazu SQL stejně jako pro celkový počet zpráv programu SQL*Net odeslaných klientovi a přijatých od klienta, je OR přístup náročnější než relační verze

Nejhůře pro OR přístup z naměřených statistik vychází množství přijatých dat přes SQL*Net od klienta.

Statistic name	OR	R	Vztah
recursive calls	1176,8%	100%	OR = 11,76767*R
db block gets	0	1326,095	0
consistent gets	3,7%	100%	OR = 0,037208*R
physical reads	50,9%	100%	OR = 0,508753*R
bytes sent via SQL*Net to client	0	169029	0
bytes received via SQL*Net from client	29604%	100%	OR = 296,0399*R
SQL*Net roundtrips to/from client	562,3%	100%	OR = 5,623378*R
Sorts (memory)	4384,5%	100%	OR = 43,84483*R
Redo size	84,5%	100%	OR = 0,844828*R
Sorts (disk)	0	0	0

Tab. 7.2 Průměrné statistiky dotazů, získané prostřednictvím nástroje autotrace

7.4.2 Trace

Hodnoty pro jednotlivé dotazy, naměřené prostřednictvím nástroje TRACE, jsou uvedeny v následující tabulce. Obsahuje hodnoty získané na základě výstupu podrobného trasování, jejichž význam je popsán v kap. 7.2, přičemž hodnoty v tabulce reprezentují rozdíl mezi dotazy provedenými v OR (a to s použitím operátoru BULK – BOR – a bez něj – OR) a relační verzi. Pokud to bylo možné, byly dotazy prováděné opakovaně 1000x pro větší výpovědní hodnotu.

diff1	BOR - R
diff2	OR - R

Z uvedené tabulky také plyne, že při zpracování dotazu využívajícího konstruktu BULK je větší spotřeba času procesoru při zpracování, což plyne z nutnosti vytvořit z hodnot výsledku získaného provedením dotazu ucelený balík, který je pak jako celek poslán ke klientovi. V OR verzi bez BULK tato nutnost odpadá.

Name/ver	cpu	elapsed	disk	query	current
avgZnamkyStudentsR	**681,75**	**1222,3**	**630646**	**18128085**	**2180410**
diff1	232,91	-195,40	-630 623,00	49 936 128,00	-2 180 410,00
diff2	216,66	-198,85	-630642	49939069	-2180410
moreExam	**287,63**	**482,86**	**115045**	**1119208**	**2091176**
diff1	734,65	687,63	-114 944,00	66 944 870,00	-2 091 176,00
diff2	754,24	678,41	-92293	66947816	-2091176
numAbsPredmet	**1106,82**	**1595,14**	**5257409**	**8346098**	**4310105**
diff1	209,01	131,19	-4 885 698,00	59 718 138,00	-4 310 105,00
diff2	160,72	50,36	-4963679	59721144	-4310105
prumerZnamkyMesta	**329,46**	**420,4**	**766**	**1014145**	**777842**
diff1	640,52	748,86	0,00	67 050 103,00	-777 842,00
diff2	637,48	771,01	-749	67053112	-777842
returnAllAbsolvovane	**2616,9**	**5355,03**	**6430594**	**25858913**	**34779376**
diff1	-2 118,03	-4 782,69	-6 430 147,00	-8 514 728,00	-34 779 376,00
diff2	-2231,11	-4902,6	-6430568	-8511717	-34779376
returnCountExams	**308,67**	**402,53**	**307894**	**1219822**	**2180756**
diff1	749,05	831,53	-307 444,00	66 844 413,00	-2 180 756,00
diff2	735,27	827,83	-307876	66847420	-2180756
returnPocetStudium	**148,49**	**210,93**	**14636**	**521796**	**2213400**
diff1	-9,65	-36,32	-14 602,00	3 646 424,00	-2 213 400,00
diff2	-19,75	-67,48	-14623	3649432	-2213400
returnSamePreq	1544,71	1876,75	655	75772779	49842
diff1	-992,02	-1 197,55	-393,00	-75 468 930,00	-49 842,00
diff2	-994,29	-1234,21	-644	-75468946	-49842
avgZnamkyObor	**457,5**	**556,06**	**714**	**17649937**	**184270**
diff1	2 909,54	3 690,12	-152,00	22 942 799,00	-184 270,00
diff2	2872,56	3701,23	-702	22933251	-184270
avgZnamkyOborRok	**547,96**	**672,82**	**186254**	**17927281**	**902252**
diff1	3 822,45	4 824,86	-186 238,00	24 993 648,00	-902 252,00
diff2	3896	5500,14	-186150	24996499	-902252
bestPrumerObor	**544,39**	**651,99**	**611082**	**17649297**	**184219**
diff1	3 255,27	4 159,78	-611 082,00	56 750 805,00	-184 219,00
diff2	3357,49	4146,59	-610629	56754273	-184219
returnDeepPreq	**1609,96**	**1988,2**	**1**	**11452830**	**253872**
diff1	1 550,10	1 673,35	38,00	28 765 793,00	2 041,00
diff2	-185,6	-206,64	17	6843477	-253872
returnMaxAbsolvovane	**366,89**	**412,31**	**0**	**1682804**	**66**
diff1	2 077,15	2 297,73	432,00	163 708 051,00	-66,00
diff2	2074,58	2261,4	0	163708150	-66
returnPrumerSS	**178,71**	**242,07**	**42127**	**723919**	**2218433**
diff1	1 801,33	2 015,51	-42 127,00	40 150,00	-2 218 433,00
diff2	1862,89	2084,45	-42127	43105	-2218433

Name/ver	cpu	elapsed	disk	query	current
returnStudent	**50,98**	**57,14**	**1**	**663545**	**28825**
diff1	-7,77	-7,18	-1,00	-164 532,00	-28 825,00
diff2	-7,17	-9,15	-1	-161532	-28825
sumAllBody	**4**	**5,1**	**11015**	**12241**	**20748**
diff1	1 128,75	1 305,47	-10 979,00	1 172 795,00	-20 748,00
diff2	1193,87	1369,88	-10990	1628312	-20748

Tab. 7.3 Výstup nástroje trace s BULK a bez něj

7.4.3 Runstat

Následující tabulka porovnává hodnoty blokování (zamykání) (získané prostřednictvím nástroje runstat – viz kap. 7.2) při provádění dotazů jak v relační tak v OR verzi včetně jejich rozdílu.
LATCH je forma lehkého zámku používaného k ochraně datových struktur. V případěuzamčení datové struktury k ní nemá nikdo další přístup a musí čekat na uvolnění zámku.
Tzn. zámek se nevztahuje na celou databázi ale jenom na konkrétní datovou strukturu. [7]
Pársování vyžaduje mnoho zámků tohoto typu – na všechny datové struktury obsažené v shared pool.

Name	R	OR	diff	Pct
avgZnamkyStudents	42,540,280	74,360,797	31,820,517	57.21%
moreExams	16,498,531	74,304,044	57,805,513	22.20%
nextPredmety	12,093,696	567,428,386	555,334,690	2.13%
numAbsPredmet	92,595,100	86,797,982	-5,797,118	106.68%
prumerZnamkyMesta	8,372,084	75,375,448	67,003,364	11.11%
returnAbsolvovane	539,798	705,51	165,712	76.51%
returnAllAbsolvovane	274,698,172	20,137,183	-254,560,989	1,364.13%
returnPocetStudium	14,797,807	4,676,212	-10,121,595	316.45%
returnSamePreq	257,342,977	178,898,509	-78,444,468	143.85%
compStudent	3,061,266	1,435,576	-1,625,690	213.24%
existExam	35,369	55,367	19,998	63.88%
returnMaxAbsolvovane	5,027,200	180,590,758	175,563,558	2.78%
returnPrumerSS	15,490,126	1,977,856	-13,512,270	783.18%
returnCountExams	74,366,707	17,244,480	-57,122,227	431.25%
returnStudent	2,166,230	1,502,638	-663,592	144.16%
avgZnamkyObor	53,617,891	202,035,770	148,417,879	26.54%
avgZnamkyOborRok	34,816,992	99,304,045	64,487,053	35.06%
bestPrumerObor	29,305,602	121,518,871	92,213,269	24.12%
returnDeepPreq	27,545,158	111,443,252	83,898,094	24.72%
plan	8,657,009	7,822,646	-834,363	110.67%
avgZnamky	2,115,250	30,562	-2,084,688	6,921.18%
returnSylabys	1,027,279	241,193,908	240,166,629	0.43%
returnSylabysNear	70,005,015	232,17	-69,772,845	30,152.48%
countStudentExam	24,089,804	2,568,830	-21,520,974	937.77%
matchPredmet	34,591,453	270,497,889	235,906,436	12.79%
sumAllBody	1,703,396	3,606,922,051	3,605,218,655	0.05%

Tab. 7.4 Blokování pro relační a OR verzi

Následující tabulky ukazují nejlepší a nejhorší získané hodnoty pro blokování:

Nejlepší hodnoty blokování pro OR vzhledem k R
LATCH.cache buffers LRU Chain Tato hodnota je užitečná v případech, kdy se uživatel snaží prohlédnout buffer cache paměť organizovanou prostřednictvím LRU.
LATCH.redo allocation Tato blokovací hodnota udává množství alokovaného místa pro znovupoužitelné entity v redo-log bufferu. Pro každou instanci existuje jeden redo-blok.
STAT...table scan rows gotten Statistika je sbírána během procházení tabulkou, ale namísto počítání použitých databázových bloků počítá počet zpracovaných řádků.

Tab. 7.5 Nejlepší hodnoty blokování pro OR

Nejhorší hodnoty blokování pro OR vzhledem k R
LATCH.row cache objects Na tuto hodnotu se je vhodné soustředit ve chvíli, kdy chce uživatel přistupovat ke cachovaným datům v datovém slovníku.
LATCH.library cache Popisuje správu cache paměti pro knihovny. Přistoupení k objektu způsobí jeho nahrání do paměti. Chce - li následně uživatel modifikovat tento objekt, musí získat přístup k zámku.
LATCH.shared pool Tato hodnota měří cache informace, které mohou být sdíleny více uživateli, jako jsou např: SQL výrazy jsou ukládány do paměti, ze které pak mohou být volány znovu. Informace z datového slovníku jako např. data související s uživatelským účtem, popisky tabulek a indexů a taky práva jsou zapamatované pro rychlejší přístup a znovupoužitelnost. Uložené procedury mohou být také cachovány pro rychlejší použití.
STAT...session uga memory Statistika ukazuje aktuální velikost UGA paměti uživatelské session. UGA je část pameti PGA, která kontroluje prostor uživatelské session, ve kterém probíhá třídění a hašování.
STAT...session uga memory max Maximální velikost UGA paměti v session.
STAT...session pga memory

Tab. 7.6 Nejhorší hodnoty blokování pro OR

LATCH.cache buffers lru chain	R	OR	diff
avgZnamkyStudents	622 056	9	-622 047
moreExams	926 795	433 370	-493 425
nextPredmety	1 111 125	366 020	-745 105
numAbsPredmet	7 350	177 739 120	177 731 770
prumerZnamkyMesta	22 213 542	1 156 777	-21 056 765
returnAbsolvovane	2 494 338	271	-2 494 067
returnAllAbsolvovane	1 013 196	867 042	-146 154
returnPocetStudium	3 116	63	-3 053
returnSamePreq	15 674 248	864 121	-14 810 127
compStudent	432 297	1 100 054	667 757
existExam	226 344	772	-225 572
returnMaxAbsolvovane	84 482 345	4 202	-84 478 143
returnPrumerSS	12	116 041	116 029
returnCountExams	1 213 253	40	-1 213 213
returnStudent	8 552 302	38 150 498	29 598 196
avgZnamkyObor	1 059 944	439 034	-620 910
avgZnamkyOborRok	1 110 080	84	-1 109 996
bestPrumerObor	782 073	525 921	-256 152
returnDeepPreq	782 073	437 699	-344 374
plan	1 274 946	33 048	-1 241 898
avgZnamky	1 274 946	-55 174	-1 330 120
returnSylabys	8 016	4 277	-3 739
returnSylabysNear	1 474 467	1 051 770	-422 697
countStudentExam	408 814	6 067	-402 747
matchPredmet	622 571	431 079	-191 492
sumAllBody	116 297	60 254	-56 043

Tab. 7.7 LRU cache buffer

Následující tabulka ukazuje blokovací hodnoty pro cache informaci, která může být sdílená více uživateli.

LATCH.shared pool	R	OR	diff
avgZnamkyStudents	7 657	11 525	3 868
moreExams	6 099	12 079	5 980
nextPredmety	67 970	312 813	244 843
numAbsPredmet	23 909	13 679	-10 230
prumerZnamkyMesta	30 987	885 117	854 130
returnAbsolvovane	5 871	12 519	6 648
returnAllAbsolvovane	2 361	9 273	6 912
returnPocetStudium	55 354	8 354	-47 000
returnSamePreq	12 334	5 663	-6 671
compStudent	4 764	5 446	682
existExam	322 912	27 165 315	26 842 403
returnMaxAbsolvovane	71 236	22 555 629	22 484 393
returnPrumerSS	5 931 243	32 034	-5 899 209
returnCountExams	16 731	2 196 935	2 180 204
returnStudent	7 452	2 928 292	2 920 840
avgZnamkyObor	6 235	2 236 349	2 230 114
avgZnamkyOborRok	2 863	38 760	35 897
bestPrumerObor	2 873 733	164 352 412	161 478 679
returnDeepPreq	6 235	167 219 910	167 213 675
plan	1 017	8 396	7 379
avgZnamky	177 953	32 201 526	32 023 573
returnSylabys	143 699	13 144 453	13 000 754
returnSylabysNear	2 870	18 978	16 108
countStudentExam	6 334	18 830	12 496
matchPredmet	2 471	11 514	9 043
sumAllBody	764	547 535 707	547 534 943

Tab. 7.8 Blokování při přístupu ke cache informacím sdílenými více uživateli

V následující tabulce je statistika, která ukazuje počet zpracovaných řádků při procházení tabulkou.

STAT...table scan rows gotten	R	OR	diff
avgZnamkyStudents	29 329 000	0	-29 329 000
moreExams	30 387 940	16 907 000	-13 480 940
nextPredmety	30 345 982	16 907 000	-13 438 982
numAbsPredmet	504 102 000	23 067 999	-481 034 001
prumerZnamkyMesta	33 215 016	16 907 000	-16 308 016
returnAbsolvovane	75 647 013	3 170	-75 643 843
returnAllAbsolvovane	30 777 626	16 907 000	-13 870 626
returnPocetStudium	36 963	0	-36 963
returnSamePreq	43 218 105	16 907 000	-26 311 105
compStudent	16 907 000	30 387 940	13 480 940
existExam	1 074 924	2 061 000	986 076
returnMaxAbsolvovane	596 534 436	76 130	-596 458 306
returnPrumerSS	23 000	4 098 836	4 075 836
returnCountExams	5 898 565	11 500	-5 887 065
returnStudent	29 763 565	16 907 000	-12 856 565
avgZnamkyObor	29 415 913	16 907 000	-12 508 913
avgZnamkyOborRok	15 132 000	0	-15 132 000
bestPrumerObor	29 415 913	19 371 889	-10 044 024
returnDeepPreq	29 415 913	16 907 000	-12 508 913
plan	48 280 622	507 180	-47 773 442
avgZnamky	48 280 622	-1 957 709	-50 238 331
returnSylabys	516 319 997	98 000	-516 221 997
returnSylabysNear	415 125 000	40 000	-415 085 000
countStudentExam	69 509 759	41 081 610	-28 428 149
matchPredmet	2 148 925	1 000	-2 147 925
sumAllBody	26 342 986	16 907 000	-9 435 986

Tab. 7.9 Počet zpracovaných řádků

Poslední tabulka ukazuje aktuální velikost UGA paměti uživatelské session. UGA je část paměti PGA, která kontroluje prostor uživatelské session, ve kterém probíhá třídění a hašování.

STAT...session uga memory	R	OR	diff
avgZnamky	65 464	0	-65 464
avgZnamkyStudent	0	654 640	654 640
moreExams	65 464	130 928	65 464
numAbsPredmet	0	3 993 304	3 993 304
plan	1 178 352	261 856	-916 496
prumerZnamkyMesta	0	523 712	523 712
returnAbsolvovane	65 464	261 856	196 392
returnAllAbsolvovane	65 464	130 928	65 464
returnCountExam	196 392	0	-196 392
returnSamePreq	0	896 536	896 536
returnSylabys	65 464	0	-65 464
returnSylabysNear	65 464	196 392	130 928
avgZnamkyObor	65 464	785 568	720 104
avgZnamkyOborRok	65 464	785 568	720 104
bestPrumerObor	554 048	0	-554 048
countStudentExam	0	196 392	196 392
existExam	493 760	0	-493 760
matchPredmet	458 248	0	-458 248
sumAllBody	0	1 832 992	1 832 992

Tab. 7.10 Velikost UGA paměti

Při pohledu na hodnoty blokování získané při změně datového typu hnízděné tabulky z VARRAY na TABLE (měřeno na dotazu číslo. 1) je vidět, že v případě použití konstruktu VARRAY bylo provedeno víc blokování než při použití konstruktu TABLE, jak ukazuje následující tabulka:

TABLE			VARRAY	
R ran in 925 hsecs			R ran in 1147 hsecs	
OR ran in 4926 hsecs			OR ran in 11020 hsecs	
R ran in 38.32% of the time			R ran in 10.41% of the time	
R latches total versus runs -- difference and pct				
verze	**R**	**OR**	**Diff**	**Pct**
TABLE	539,798	705,510	165,712	76.51%
VARRAY	746,345	1,477,383	731,038	50.52%

Tab. 7.11 Blokování TABLE vs. VARRAY

Použitím typu VARRAY je možné vyhnout se drahým SQL spojením. Na druhou stranu, delší záznamy v poli VARRAY mají na svědomí déle trvající prohlížení celého pole (full scan).

Při dotazování se na položku tabulky typu VARRAY, se při měření nástrojem autotrace objevují hodnoty, jenž jsou mnohonásobně vyšší než je tomu u stejného dotazu, kde bylo pole VARRAY nahrazeno hnízděnou tabulkou (viz kap. 7.3.2.1), tak, jak ukazuje následujících výpis získán pro provedení dotazu č. 4:

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=477151298 Card=1 Bytes=1928)
   1    0   SORT (AGGREGATE)
   2    1     NESTED LOOPS (Cost=477151298 Card=3542631494 Bytes=6830193520432)
   3    2       NESTED LOOPS (Cost=58418 Card=43372080 Bytes=83621370240)
   4    3         TABLE ACCESS (FULL) OF 'STUDENT' (Cost=8 Card=5310 Bytes=10237680)
   5    3         COLLECTION ITERATOR (PICKLER FETCH)
   6    2       COLLECTION ITERATOR (PICKLER FETCH)
```

7.4.4 Velikost tabulek

Následující tabulka ukazuje velikost relačních a OR tabulek s ekvivalentními daty. Z uvedených hodnot vyplývá výhoda OR tabulek, u kterých je možno využívat referencí na objekty v tabulkách.

počet studentů	počet předmětů	počet úrovní prerekvizit	max.zápis	min_abs	max_abs
1075	3935	25	30	5	20
verze	**TABLE_NAME**	**BYTES**	**BLOCKS**		
OR	STUDENT	6291456	768		
OR	OSTUDIUM	851968	104		
R	OSOBA	393216	48		
R	STUDIUM	720896	88		
R	ZKOUS	4194304	512		
OR	PŘEDMĚT	2097152	256		
R	POVINN	4194304	512		
R	PREQ	917504	112		
počet studentů	**počet předmětů**	**počet úrovní prerekvizit**	**max.zápis**	**min_abs**	**max_abs**
100	500	5	25	5	20
verze	**TABLE_NAME**	**BYTES**	**BLOCKS**		
OR	STUDENT	524288	64		
R	OSOBA	393216	48		
R	ZKOUS	4194304	512		
OR	PŘEDMĚT	262144	32		
R	POVINN	4194304	512		
R	PREQ	917504	112		

Tab. 7.12 Tabulka velikostí relačních a OR tabulek obsahujících ekvivalentní data

7.4.5 Dávkové soubory

Další měření probíhalo prostřednictvím tří dávkových souborů, ve kterých se spouštěly všechny vytvořené dotazy. V prvním souboru byly postupně v cyklech volány dotazy pro relační verzi procedur, které byly součástí balíku. Druhý soubor obsahoval znovu relační procedury, tentokrát ale ve formě uložených procedur. A nakonec třetí dávkový soubor obsahoval ekvivalentní dotazy pro OR verzi uložených procedur.
Z měření vyplynulo, že obě relační verze pro uložené a neuložené procedury jsou ve všech parametrech stejné kromě celkového počtu datových bloků fyzicky přečtených z disku, které byly pro uloženou verzi relačních dotazů větší o řád.
Zajímavější výsledky byly získány porovnáním výsledků získaných z relačního a OR dávkového souboru. Z měření vyplynulo, že při provádění ekvivalentních dotazů obou verzí bylo u relační verze provedeno přes 50x více pársování, které trvalo přibližně 10x delší dobu. Na druhé straně proběhlo u OR verze, na rozdíl od relační, několik fyzických čtení z disku a použití bufferů v konzistentním módu.

Celkový čas potřebný pro provedení dotazů, jakož i počet fyzických čtení z disku, byl u relační verze přibližně 3 až 4 krát delší. Pokud jde o počet použitých bufferů v konzistentním módu, těch bylo pro relační verzi použito více přibližně o 2 řády.
Konzistence ve čtení je schopnost Oraclu znovupoužít undo informace, čímž je umožněno provádět neblokované dotazy a konzistentní čtení [6].
Systém Oracle přitom využívá dvou způsobů čtení bloků při provádění modifikačního výrazu:

- konzistentní čtení (consistent reads) - při hledání řádku k modifikaci
    - potenciálně historická verze bloku
    - nemůže být měněn
    - nemůže být ve stavu dirty
    - může být použit k sestavení konzistentní verze
    - ve vyrovnávací paměti může existovat více verzí stejného bloku
    - může být ve formě:
        - jednoduchého bloku (sekvenční čtení)
        - multi bloku (izolované čtení)
- aktuální čtení (current reads) - při získávání bloku pro aktuální změnu požadovaného řádku
    - aktuální verze bloku
    - může být měněn
    - může být ve stavu dirty
    - současně se může ve vyrovnávací paměti vyskytovat pouze jedna aktuální verze bloku a to i pro všechny instance
    - může být využit pro sestavení konzistentní verze

Buffery jsou uloženy ve vyrovnávací paměti, tedy většina V/V operací může být provedena právě prostřednictvím vyrovnávací paměti. Zbytek V/V operací se provede jako fyzické V/V operace.

Na druhou stranu, jde-li o celkový čas potřebný při získávání výsledných řádků, je na tom lépe relační verze a to přibližně o řád.
Uvedené skutečnosti dokumentují následující tabulky s hodnotami nástroje TRACE[5]:

call	count	cpu	Elapsed	Disk	Query	Current	rows
Parse	57074	9.15	8.77	0	0	0	0
Execute	1201894	3139.79	3073.17	878495	19032987	5180220	5143777
Fetch	1355714	2396.96	2337.44	699744	92343763	0	1004396
total	2614682	5545.91	5419.39	1578239	1,11E+08	5180220	6148173

Tab. 7.13 Výstup nástroje trace po hromadném provedení relačních dotazů implementovaných prostřednictvím neuložených procedur

[5] OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

call	count	cpu	elapsed	disk	query	Current	rows
Parse	57388	9.22	9.11	0	0	0	0
Execute	1202213	3208.17	3137.72	4223207	19032938	5209371	5143777
Fetch	1356441	2632.50	2569.68	5229302	92345795	0	1004886
total	2616042	5849.90	5716.52	9452509	1,11E+08	5209371	6148663

Tab. 7.14 Výstup nástroje trace po hromadném provedení relačních dotazů implementovaných prostřednictvím uložených procedur

call	count	cpu	elapsed	disk	query	Current	rows
Parse	1090	1.15	1.54	22	6819	0	0
Execute	4651067	1031.45	988.62	247670	660319	515403	456966
Fetch	4651741	33877.61	33127.75	111448	79746706	0	4654648
total	9303898	34910.22	34117.92	359140	80413844	515403	5111614

Tab. 7.15 Výstup nástroje trace po hromadném provedení OR dotazů implementovaných prostřednictvím uložených procedur

V dalším měření byly porovnávány statistiky získané po hromadném spuštění stejných dotazů a to pro relační verzi, OR verzi a OR verzi bez hnízděných tabulek.
Z naměřených hodnot vyplývá, že ve všech ukazovatelích je na tom nejlépe OR verze s hnízděnými tabulkami. Nejvýraznější rozdíl je v počtu diskových operací, kdy je na tom OR verze s hnízděnými tabulkami mnohonásobně lépe než relační a o řád lépe než OR verze nevyužívající hnízděných tabulek.
Tyto skutečnosti jsou zachyceny v následujících tabulkách:

call	Count	Cpu	elapsed	disk	query	current	rows
Parse	282	0.07	0.06	0	0	0	0
Execute	15789	35.21	42.03	37810	149455	49854	51698
Fetch	15877	485.95	492.90	6121462	7593142	0	15636
total	31948	521.24	535.00	6159272	7742597	49854	67334

Tab. 7.16 Výstup nástroje trace po hromadném provedení relačních dotazů

call	Count	Cpu	elapsed	disk	query	current	rows
Parse	50	0.10	0.09	0	309	0	0
Execute	51478	29.13	29.46	0	26562	0	0
Fetch	51532	374.98	369.43	10	6254370	0	51558
total	103060	404.22	398.99	10	6281241	0	51558

Tab. 7.17 Výstup nástroje trace po hromadném provedení OR dotazů s hnízděnými tabulkami

call	count	Cpu	elapsed	disk	query	current	rows
Parse	32	0.02	0.02	0	3	0	0
Execute	52209	34.10	34.65	0	698	0	0
Fetch	52209	700.38	693.33	432	19225411	0	52208
total	104450	734.50	728.01	432	19226112	0	52208

Tab. 7.18 Výstup nástroje trace po hromadném provedení OR dotazů bez hnízděných tabulek

Vytvoření indexů nad relačními tabulkami, které urychlí zpracování záznamů o zkouškách studentů v jednotlivých studiích, nemá na hodnoty Parse, Exekute a Fetch žádný zásadní vliv, jak ukazují následující tabulky obsahující výsledky získané po dávkovém spuštění relačních dotazů, znatelnější rozdíl je po zavedení indexů nad hnízděnými tabulkami u OR verze, kdy se zkrátil čas využívání procesoru a počet diskových operací:

call	Count	Cpu	elapsed	disk	query	current	rows
Parse	28079	5.12	5.91	0	0	0	0
Execute	1028274	1907.14	2277.31	452175	12513997	3990436	4047893
Fetch	1224165	2063.06	2424.40	1192256	84467047	0	931429
total	2280518	3975.34	4707.64	1644431	96981044	3990436	4979322

Tab. 7.19 Výstup nástroje trace po hromadném provedení relačních dotazů bez indexů

call	Count	Cpu	elapsed	disk	query	current	rows
Parse	28401	5.85	8.71	4	7	0	0
Execute	1028786	1891.82	2304.82	397829	12407944	3990382	4047893
Fetch	1225007	2074.02	2414.05	1105774	84428675	0	931914
total	2282194	3971.70	4727.59	1503607	96836626	3990382	4979807

Tab. 7.20 Výstup nástroje trace po hromadném provedení relačních dotazů s indexy

call	count	cpu	elapsed	disk	query	current	rows
Parse	611	0.56	4.44	0	1945	0	0
Execute	4396743	1992.39	2380.19	1639320	2557461	510322	457083
Fetch	4396239	32491.27	37183.55	478191	3,92E+08	0	4398235
total	8793593	34484.22	39568.19	2117511	3,94E+08	510322	4855318

Tab. 7.21 Výstup nástroje trace po hromadném provedení OR dotazů bez indexů

call	Count	cpu	elapsed	disk	query	current	rows
Parse	1729	1.67	4.67	57	6020	0	0
Execute	3386371	928.82	1083.99	444880	1964840	515295	457086
Fetch	3388653	25665.62	30078.94	963	3,86E+08	0	3389713
total	6776753	26596.12	31167.62	445900	3,87E+08	515295	3846799

Tab. 7.22 Výstup nástroje trace po hromadném provedení OR dotazů s indexy

Úplně jiná situace po přidání indexů je, podíváme-li se na blokování a časové hledisko. Zde došlo úplně přirozeně ke zkrácení doby potřebné pro provádění jednotlivých dotazů pro obě verze, přičemž bylo vyžadováno použití méně zámků, což se výrazněji projevilo u OR verze, jak ukazuje následující tabulka s hodnotami pro provedení obou verzí dotazu **avgZnamkyOborRok**:

tečky			reference	
R ran in 57 hsecs			R ran in 5665 hsecs	
OR ran in 105749 hsecs			OR ran in 34763 hsecs	
R ran in .05% of the time			R ran in 16.3% of the time	
R latches total versus runs -- difference and pct				
verze	**R**	**OR**	**Diff**	**Pct**
tečky	16,539	385,587,389	385,570,850	.00%
reference	1,333,534	7,581,425	6,247,891	17.59%

Tab. 7.26 Odlišné použití referencí

Pokud jde o plán vyhodnocení dotazu, postupuje optimalizátor u obou verzí dost podobně a v některých případech téměř úplně stejně, tak jako to ukazuje následující výpis pro dotaz **existExam**:

R

```
SELECT 1
FROM dual
WHERE EXISTS(SELECT 1
             FROM ZKOUS Z
             INNER JOIN STUDIUM S ON
               Z.ZIdent=S.SIdent AND
               Z.ZPovinn=in_pov AND
               Z.Znamka>0
             INNER JOIN OSOBA O ON
             S.SOIdent=O.OIdent AND
             O.OIdent=in_OIdent);
```

Execution Plan

```
0    SELECT STATEMENT
     Optimizer=ALL_ROWS (Cost=11
     Card=8168)
1    0   FILTER
2    1     TABLE ACCESS (FULL) OF
     'DUAL' (Cost=11 Card=8168)
3    1     TABLE ACCESS (BY INDEX
     ROWID) OF 'ZKOUS' (Cost=2 Card=6
     Bytes=66)
4    3       NESTED LOOPS (Cost=9
     Card=2 Bytes=44)
5    4         NESTED LOOPS (Cost=3
     Card=3 Bytes=33)
6    5           INDEX (UNIQUE SCAN)
     OF 'SYS_C0056356' (UNIQUE) (Cost=1
     Card=1 Bytes=4)
7    5           TABLE ACCESS (BY
     INDEX ROWID) OF 'STUDIUM' (Cost=2
         Card=3 Bytes=21)
8    7             INDEX (RANGE SCAN)
     OF 'IDX_STUDIUM' (NON-UNIQUE)
     (Cost=1 Card=3)
9    4         INDEX (RANGE SCAN) OF
     'IDX_ZKOUS' (NON-UNIQUE) (Cost=1
     Card=15)
```

OR

```
SELECT 1
FROM dual
WHERE EXISTS(SELECT 1
             FROM STUDENT s,
             TABLE(s.Absolvovane) sa,
             TABLE(sa.Zkousky) saz
             WHERE s.OIdent=in_OIdent
AND saz.Predmet.Predmet.Povinn=in_pov);
```

Execution Plan

```
0    SELECT STATEMENT
     Optimizer=ALL_ROWS (Cost=11
     Card=8168)
1    0   FILTER
2    1     TABLE ACCESS (FULL) OF 'DUAL'
     (Cost=11 Card=8168)
3    1     TABLE ACCESS (BY INDEX ROWID)
     OF 'ABSOLVOVANE_ZKOUSKY_NT'
     (Cost=2 Card=1 Bytes=37)
4    3       NESTED LOOPS (Cost=6 Card=1
     Bytes=80)
5    4         NESTED LOOPS (Cost=4
     Card=1 Bytes=43)
6    5           TABLE ACCESS (BY INDEX
     ROWID) OF 'STUDENT' (Cost=2 Card=1
     Bytes=23)
7    6             INDEX (UNIQUE SCAN)
     OF 'SYS_C00200039' (UNIQUE)
     (Cost=1 Card=101)
8    5           TABLE ACCESS (BY INDEX
     ROWID) OF 'ABSOLVOVANE_NT' (Cost=2
     Card=1 Bytes=20)
9    8             INDEX (RANGE SCAN) OF
     'IDX_ABSOLVOVANE' (NON-UNI
     QUE) (Cost=1 Card=1)
10   4         INDEX (RANGE SCAN) OF
     'IDX_ABSOLVOVANE_ZKOUSKY' (NON
     -UNIQUE) (Cost=1 Card=10)
```

Následující výpis plánů vyhodnocení dotazu ukazuje vyšší režii při vykonávání plánu dotazu pro OR verzi s hnízděnými tabulkami, hodnoty získané při provádění dotazu **returnPocetStudium**

R

```
INSERT INTO tab_pocetStudium
        SELECT s.SOIdent, count(s.SIdent)
        FROM STUDIUM s
        GROUP BY S.SOIdent
        ORDER BY count(s.SOIdent) desc;
```

Execution Plan

```
0      INSERT STATEMENT
       Optimizer=ALL_ROWS (Cost=14
       Card=1076 Bytes=4304)
1    0   SORT (ORDER BY) (Cost=14
       Card=1076 Bytes=4304)
2    1     SORT (GROUP BY) (Cost=14
       Card=1076 Bytes=4304)
3    2       INDEX (FAST FULL SCAN) OF
       'IDX_STUDIUM' (NON-UNIQUE) (Cost=4
       Card=3000 Bytes=12000)
```

OR

```
SELECT o_pocetStudium(REF(s),
count(sa.oid))
        FROM STUDENT s,
TABLE(s.Absolvovane) sa
        GROUP BY REF(s)
        ORDER BY count(sa.oid) desc;
```

Execution Plan

```
0      SELECT STATEMENT
       Optimizer=ALL_ROWS (Cost=60
       Card=4738 Bytes=236900)
1    0   SORT (ORDER BY) (Cost=60
       Card=4738 Bytes=236900)
2    1     SORT (GROUP BY) (Cost=60
       Card=4738 Bytes=236900)
3    2       HASH JOIN (Cost=15
       Card=4738 Bytes=236900)
4    3         TABLE ACCESS (FULL) OF
       'ABSOLVOVANE_NT' (Cost=6 Card=3513
       Bytes=80799)

5    3         TABLE ACCESS (FULL) OF
       'STUDENT' (Cost=7 Card=4738
       Bytes=127926)
```

Z uživatelského hlediska jsou z uvedených hodnot nejdůležitější ty, které můžeme bezprostředně pocítit při provádění jednotlivých dotazů. Za nejviditelnější se dá pokládat časové hledisko dotazů obou přístupů. S ním taky úzce souvisí škálovatelnost, neboli je-li systém schopen zpracovávat dané dotazy i v případě vícenásobného současného přístupu více uživatelů. Dále by šlo do této kategorie zařadit i počet fyzických operací, tedy čtení a zápis z disku. Ostatní charakteristiky jako logické čtení, třídění, rekurzivní zpracování jsou pro uživatele málo viditelné.

ZÁVĚR

I když bylo původním předpokladem, že OR přístup je efektivnější oproti klasickému relačnímu, na základě provedených praktických měření se ukázalo, že takto jednoznačný přínos OR databáze nepřinášejí. Na základě hodnot, získaných při provádění relačních, resp. OR dotazů, nelze označit ani jednu z verzí jako jednoznačně lepší. Každá má své výhody a nevýhody v závislosti na konkrétním použití v konkrétní situaci.
Relační model je jednoduchý a elegantní. Relační databáze však nejsou navrhovány pro ukládání objektů a naprogramování rozhraní pro ukládání objektů v databázi je velmi složité. Jsou dobré pro řízení velkého množství dat a vyhledávání v nich, ale poskytují nízkou podporu pro manipulaci s nimi. Jsou založeny na dvourozměrných tabulkách a vztahy mezi daty jsou vyjadřovány porovnáváním hodnot v nich uložených. Z toho plyne jejich nízká kampatibilita s OO návrhem aplikací a nutností jeho převodu do relačního datového modelu. U OR přístupu se s tímto problémem nesetkáváme. Jazyky jako SQL umožňují tabulky propojit za běhu, aby vyjádřily vztah mezi daty.
Jako výhodné se ukázalo používání hnízděných tabulek, které ve většině případů vedlo k vyšší rychlosti vykonávání dotazů oproti verzi relační (viz kap. 7.3.3). Za jeden z hlavních přínosů jejich používání v OR verzi lze označit lepší škálovatelnost modelu (viz tab. 7.4), kdy je možné OR aplikaci s menší režijí na zamykání nasadit do víceuživatelského prostředí. Dále pak již vzpomenutá výhodnost použití hnízděných tabulek, sdružujících data stejného charakteru (viz kap.4.8 bod 12)). S takto sestavenými daty lze navíc pracovat jako s celkem na rozdíl od relačního přístupu.
Dalším velkým přínosem je jednodušší a přímočařejší práce s předáváním vícenásobných hodnot mezi podprocedurami, kde lze využít konstruktu pole, zatímco u relačního přístupu lze téhož dosáhnout pouze za využití pomocných tabulek. Efektivní využití nachází OR konstrukt kolekce při předávání dat mezi OR SŘBD a klientským programem, kdy je možné s jeho pomocí snížit objem předávaných dat (viz kap. 4.4). Výhodou OR je také efektivnější modelování struktur využívaných procedurami, možnosti využívání objektových rysů jako dědění, znovupoužitelnost, uživatelsky definované datové typy, reference na řádky v OR tabulkách, vytváření uživatelsky definovaných agregovaných datových typů, díky kterým již není nutné vytvářet pohledy spojující několik relačních tabulek pro náhled na složené objekty a z nich pak čerpat potřebná data.
Další výhodou OR databází je, že umožňují při tvorbě nových aplikací jednoduchý přechod z relačních aplikací a to bez velkých změn.
Mezi nevýhody OR přístupu patří:

- malá až nulová přenositelnost aplikace s ORDB vrstvou v porovnání se stejnou používající RDB vrstvu
- problémy s implementací VARRAY v Oracle9, komplikovaný přístup k datům v něm, praktická nepoužitelnost v případě, že jsou ve složkách pole další vnořené tabulky a podobně
- v mnoha případech vyžaduje OR přístup více času na provedení dotazu, což přímo souvisí s práci s objekty, tedy větší časová náročnost při manipulaci s nimi – načítání, ukládání (viz kap. 7.3.2.1)
- jelikož OR tabulky nejsou normalizované, může u nich docházet k redundanci v nich uložených dat (viz kap. 5.3)

Při provádění dotazů pro relační verzi, OR verzi s hnízděnými a OR verzi bez hnízděných tabulek vyplynulo, že ve všech ukazatelích je na tom nejlépe OR verze s hnízděnými tabulkami, což se nejvýrazněji projevilo u počtu diskových operací (viz tab. 7.16, 7.17, 7.18).

OR verze se v porovnání s relační verzí jeví jako více závislá na správném použití indexů. Jejich případné vynechání se výrazněji projevilo právě u OR dotazů, kdy se prodloužil čas využívání procesoru a počet diskových operací. U relační verze byt rozdíl méně patrný (viz tab. 7.19, 7.20, 7.21, 7.22). Vynechání indexů se pro OR verzi výrazněji projevilo rovněž v počtu blokování (viz tab. 7.23) a v nárustu celkového času stráveného prováděním dotazu (viz tab. 7.24).
Pokud jde o plán vyhodnocení dotazu, postupuje optimalizátor u obou verzí obdobným způsobem (viz výpisy v závěru kap. 7.4.5). Při provádění obou verzí dotazů byl pužit tentýž plánovač z OR verze databáze, tedy relační verze modelu do jisté míry čerpala z možností, vyvinutých pro OR model. Z porovnání blokování dotazů relační verze realizované prostřednictvím uložených a neuložených procedur s OR verzi s hnízděnými tabulkami plyne, že pro většinu dotazů není mezi oběma přístupy rozdíl (viz tab. 7.25).
Z hlediska náročnosti zápisu zdrojového kódu jsou dotazy pro obě verze v podstatě stejné. Délka zápisu ekvivalentních dotazů je však pro OR verzi o něco větší, především díky nutnosti využívání operátoru pro TABLE odhnízďování hnízděných tabulek.

Z výše uvedeného plyne, že existuje hodně případů, ve kterých představuje OR přístup jednoznačný přínos (což závisí hlavně na vhodném použití nových datových struktur a jejich implementaci systémem), na druhé straně je možno najít řadu příkladů, kdy je lepší použít při řešení konkrétního problému klasický relační přístup (řízení velkého množství dat a jejich vyhledávání).

LITERATURA

[1] Stonebraker M., Brown P., Mooreová D. (1999): Objektově-relační SŘBD Analýza příští velké vlny. BEN, Praha

[2] Dokumentace systému Informix *IDS 2000*, Dokumentační CD Informix Answers OnLine ver. 3.2, http://www-306.ibm.com/software/data/informix/pubs/library/ids_92.html

[3] Dokumentace systému Oracle 9i, http://www.oracle.com/technology/documentation/oracle9i.html

[4] Oracle interMedia User's Guide and Reference. http://www.cise.ufl.edu/help/database/oracle- docs/appdev.920/ a88786/mm_intr.htm#610846

[5] Kyte T. (2005): ORACLE Návrh a tvorba aplikací. CP Books, Brno

[6] Dyke J. (2005): prezentace Logica I/O

[7] Danchenkov A., Burleson D.: ORACLE TUNING - The definitive reference

[8] Moore D. (2003): Rampant Tech Press

[9] Oracle9i Application Developer's Guide - Object-Relational Features Release 2 (9.2) http://www.lc.leidenuniv.nl/awcourse/oracle/appdev.920/a96594/toc.htm

[10] Martin M. (2005): Models of Human Reasoning - Inheritance Tudory. California State University http://www.cs.nmsu.edu/~mmartin/inheritance_theory.pdf

[11] Melton J.: Oracle's SQL:1999 Presentation. www.wiscorp.com/SQLStandards.html

[12] Ehrlich J., Marek P., Marvan T., Paľo M., vedoucí projektu RNDr. Kuthan V. (2003): Dynamické HTML , Softwarový projekt MFF UK

SEZNAM TABULEK

SEZNAM OBRÁZKŮ