Jihočeská univerzita v Českých Budějovicích
Pedagogická fakulta, katedra informatiky
Obor: Výpočetní technika a informatika

Objektové a objektově-relační databáze

Bakalářské práce

Autor: Jakub Steinbauer
Vedoucí práce: Mgr. Miloš Prokýšek

České Budějovice 2007

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně, a že jsem veškeré použité zdroje uvedl v seznamu použitých zdrojů.

…….....

Jakub Steinbauer

Anotace

Cílem mé absolventské práce bylo podat základní přehled o současném stavu a vývoji objektových a objektově relačních databází. Dále se práce věnuje detailnímu popisu těchto databázových technologií z pohledu doporučených standardů a z pohledu jejich použitelnosti v dnešní době objektově orientovaného programovacího přístupu.

Abstract

The main goal of my dissertation work was to make basic overview of current situation and development in objected and objected-relational databases. Further, work showing these databases technologies in more details. From the view of recommended standards and from the view of their usability in today's word, where object oriented development dominating.

1 ÚVOD

Objektové databáze jsou dnes jedním ze směrů vývoje databázových systémů. V praxi jsou však stále málo využívané oproti zavedeným relačním databázovým systémům. Hlavním důvodem je jistá dostatečnost a osvědčenost těchto systémů. Chuť experimentovat a investovat nemalé prostředky do změny technologie má jen velmi malá část tohoto tržního sektoru.

Tato bakalářská práce si klade za cíl přinést pohled na současné trendy v oblasti databázových technologií a detailně popisuje dva hlavní směry. Objektově-relační - vylepšení relačního modelu o objektové vlastnosti a již zmíněné plně objektové databáze - jako přirozenou reakcí na vývoj programovacích postupů v moderních technologiích. Práce přináší ukázky možností a podpory objektově orientovaného programování v těchto systémech a popis standardů kterými by se měli řídit.

2. Přehled databázových technologií

2.1 Historie a pohled na současný stav

V dnešní době můžeme sledovat dva důležité trendy v používání databází, databáze se stále zvětšují a stávají se více a více komplexními. Větší jsou nejen v množství uložených dat (datové sklady se zvětšují ze stovek gigabytů na stovky terabytů), ale také v počtu uživatelů, kteří vyžadují současný přístup do databáze. Některé takové databáze, umožňující současný přístup k uživatelům se rozšířily z několika tuctů uživatelů až na několik stovek nebo tisíců. Databáze v dnešní době musí být také komplexnější, protože se zvětšila různorodost aplikací, které mají podporovat. Od klasických aplikací, jako jsou datové sklady nebo podpora rozhodování, až po webové servery, internet a intranet. V souvislosti s tímto trendem se také zvětšuje rozmanitost typů dat, které má databáze zpracovávat, organizovat a skladovat. Dnešní databáze již nejsou zaplněny pouze čísly a znaky, ale mají co do činění s komplexními datovými typy, jako jsou obrázky, nestrukturovaný text, HTML stránky, applety, audio a video apod.

2.2 Trendy v databázových systémech

Potřeba práce s nestrukturovanými daty ukazuje, že relační databázové systémy mají jistá koncepční omezení, která vycházejí právě z přísně strukturovaného pohledu na data. Relační systémy již nejsou schopny efektivně optimalizovat zpracování dotazů nad těmito nestrukturovanými daty a poskytnout podporu pro manipulační jazyky s těmito objekty. Řešení nabízí jiný směr ve způsobu chápání dat a jejich vztahů. Ten se uplatňuje v objektově orientovaných databázových systémech. Rozdíl mezi oběma modely spočívá v tom, že zatímco relační model je založen na množinách záznamů, kde jsou statické informace o datech. Objektově orientovaný model v jednotlivých objektech

uchovává nejen tyto statické informace, ale také konkrétní procedurální informace, vztahující se k chování a operacím s těmito objekty.

Za posledních 25 let vývoje počítačů a programování aplikací, můžeme vysledovat mohutný trend přechodu od strukturovaného k objektově orientovanému programování. Toto platí i v oblasti zpracování dat a databází. V osmdesátých létech způsobily revoluci relační databáze, v létech devadesátých s mohutným nástupem objektově orientovaného programování začaly vznikat i objektově orientované databáze, které si kladou za cíl ulehčit a urychlit práci s daty. V rámci ‚objektové orientace' v databázích lze v současné době rozlišit dva základní vývojové trendy. Jedná se o trend evoluční a revoluční.

Evoluční trend – spočívá v postupných změnách a vylepšeních klasického relačního datového modelu směrem k objektům. Jedná se o využívání triggerů, možnost tvorby nových datových typů, rozšíření základních datových typů (BLOB,CLOB,BFILE,apod.) a objektově orientovaných aplikačních programovacích jazyků. To postupně vede k rozšiřování datového modelu po teoretické stránce (např. odklon od požadavku na první normální formu). Tento přístup také umožňuje zpětnou kompatibilitu s relačními systémy, což znamená, že starší relační aplikace bude pracovat beze změn i na nových databázových strojích[4].

2.3 Současné databázové technologie

V současné době existuje na trhu obrovské množství systémů řízení báze dat. Vývojové týmy na celém světě musí denně řešit otázku jaký produkt zvolit. Důležité je samozřejmě jaký typ aplikací vyvíjí, jaké jsou jejich požadavky na databázový systém. Může jít o jednotlivce tvořícího dynamické webové stránky, menší systémy na míru nebo o velké korporace pracující s terabyty

dat. Vyznat se ve všech z nich není v lidských možnostech, přesto se dají rozdělit do několika skupin:

- **objektově – relační databáze** – jde v podstatě o všechny novější relační systémy řízení báze dat, které jsou obohaceny o objektové vlastnosti. Vzhledem k tomu , že jsou tyto databázové systémy založeny na relačním modelu a „objektovost“ je navíc (volitelná, je kladen důraz na zachování stoprocentní kompatibility) jsou velmi rozšířené a populární.
- **objektové-databáze –** vznikly jako reakce na volání po plně objektovém systému řízení báze dat. Kdysi se vedly nesmyslné spory o tom jestli objektové databáze nejsou slepou větví vývoje vzhledem k jejich výkonu v porovnání s relačními databázemi. Dnes již víme, že tomu tak není a některé objektové databáze zdatně konkurují i velkým a zaběhlým relačním systémům.
- **post-relační databáze –** jedná se o systém, který se snaží kombinovat relační a plně objektové databáze. Jde v podstatě o objektovou databázi, která zároveň implementuje relační systém dotazování. Na trhu jí reprezentuje především databázový produkt Caché.
- **multimediální databáze –** jedná se o proprietální řešení správy multimediálních dat. Multimediální databáze poskytují mnoho různých prostředků jak s těmito daty efektivně nakládat.

O objektových a objektově-relačních databázových systémech se vzhledem k jejich zásadnosti a také tématu této bakalářské práce bude hovořit v následujících kapitolách.

2.4 Objektově – relační mapování

Vzhledem k tomu, že jsem byl v naší firmě svědkem vývoje několika informačních systémů pro nejrůznější zákazníky, musím se zmínit o technice

objektově relačního mapování. V dnešní době kdy ve většině případů je čas zásadní proměnnou ve vývoji software, jsme svědky nebývalého využití této technologie. Drtivá většina dnešních programů je psaná v některém z objektově orientovaných jazyků. To přináší nejrůznější výhody, jako například existující sofistikované postupy vývoje, opakovaná použitelnost atd. Při programování vyvstává otázka jaký databázový systém použít k vývoji. Objektové databáze stále nejsou v očích výrobců software shledávány jako ideální (často většinou z neznalosti). Objektově-relační databáze se zdají vyhovující, ovšem do té doby než si uvědomíme, že o perzistenci se musíme starat sami a zanášíme objektově orientovaný kód SQL příkazy. Jak jsme si již řekli, jelikož čas hraje hlavní roli při získávání zakázky,jsou objektově-relační mapovací nástroje velmi oblíbené.

Objektově relační mapování je výhodné proto, že umožňuje snadné propojení vyzkoušených relačních systémů s moderními objektovými programovacími jazyky. Mapovací nástroje slouží k načítání dat z relačních databází , jimi naplnit datové položky jednotlivých objektů a poté je opět uložit. Snaha je v co nejvyšší míře využít možnosti obou technologií, tzn. objekty by měly reprezentovat objekty reálného světa, databáze zase využívat indexy, pohledy, primární klíče.

Základní idea je taková, že jedna databázová tabulka představuje jednu třídu, jednotlivé řádky tabulky pak představují instance třídy. Pro vazby mezi objekty (zapouzdření) se pak používají primární a cizí klíče v databázi. Dále je potřeba řešit dědičnost tříd (buď ukládáním dceřiných tříd do jedné tabulky, která má příslušné sloupce pro datové složky všech podtříd, nebo vytvářením dalších tabulek, které udržují údaje o doplňujících datových složkách); některé databáze (např. PostgreSQL) umožňují používat dědičnost přímo v návrhu databází (tabulka může být vytvořena jako potomek jiné tabulky - pak „dědí“

všechny její sloupce a může přidat další; lze se odkazovat jak na jednotlivé tabulky tak na celou „rodinu“ - vypíší se společné sloupce ze všech tabulek). V případě obnovování objektů z uložených dat je pak vhodné využít síly relačních databází ve vyhledávání údajů a filtrování. Framework by měl také řešit efektivní přenos dat – např. při vylistování objektů z databáze nám mohou stačit pouze některé jejich datové složky (např. název) a jejich plný „obsah“ můžeme potřebovat pouze u některých až na požádání (např. v případě výběru objektu ze seznamu uživatelem). ORM framework by měl v takovém případě umět vytvářet zástupné objekty, které z databáze načítají požadovaná data dle potřeby[7].

Výhody ORM

- rychlé a spolehlivé ukládání velkého množství objektových dat v relačních databázích
- čistě objektový přístup k programování, kód se nezanáší zlomky SQL kódu

Nevýhody ORM

- jde o mapování , což přináší určitou režii a může utrpět rychlost
- některé konstrukce mohou být obtížně převoditelné

2.4.1 Hibernate

Hibernate je nejrozšířenější Open Source objektově relační mapovací framework, který je vytvářen s ohledem na výkon aplikace a pokrývá nejrůznější možnosti využití a funkce vyzkoušené jsou tyto databáze: Oracle, DB2, Sybase, MS SQL Server, PostgreSQL, MySQL, HypersonicSQL, Mckoi SQL, SAP DB, Interbase, Pointbase, Progress, FrontBase, Ingres, Informix, Firebird . Hibernate má vlastní jazyk Hibernate Query Language (HQL), což je

jazyk určený k dotazování se nad databází, respektive vyhledávání a filtrování uložených dat (jazyk je podobný podmínkám klauzule WHERE jazyka SQL).

Mezi další vlastnosti Hibernate patří například různé způsoby zadávání vyhledávacích podmínek např. jazyk HQL, nativní dotazy SQL nebo zadávání omezujících podmínek v Javě.

Dále umožňuje ukládat různé druhy objektových asociací, různé stupně cache, zpožděná inicializace (nahrávání kompletních dat objektu až v okamžiku potřeby), práci s podtřídami či s výčtem hodnot (enum). Hibernate umožňuje používat uživatelem definované typy, různé druhy identifikátorů-primárních klíčů. Samozřejmostí je mapování one-to-one i many-to-one. V případě jednoduchých objektů a tabulek je možné definovat pouze mapování datové složky na název sloupce, není nutné uvádět datové typy další vlastnosti. Volitelně je možné nechat si na základě mapovacích souborů vygenerovat schéma databáze, resp. SQL příkaz pro vytvoření potřebných tabulek. V případě náročnějších požadavků je možné u jednotlivých vlastností definovat jejich typy, chování při různých operacích (insert, update, čtení dat – vypočítané hodnoty)[7].

3 Objektové databáze

3.1 Historie

Pojem objektově orientovaná databáze se poprvé objevil v roce 1985. Za zmínku stojí vědecké projekty Encore-Ob/Sever(Brownova univerzita), Exodus (Wisonzinská univerzita), Iris (Hewlett-Packard), ODE (Bellovi laboratoře), Orion, Vodak a Zetgeist.

Brzy poté se objevily první komerční produkty jako například GemStone, Base a VBase. V první polovině devadesátých let to byly ITASCA, Jasmíne, Matjese, Objektivity/DB, ObjectStore, ONTOS, O2, POET, Verstant Object Databáze. Některé tyto produkty existují ještě dnes.

Tyto první produkty, které poskytovaly koncept perzistence pro objektově orientované jazyky podporovali vazby na nerůznější programovací jazyky GemStone(SmalTalk) nebo Gbase(Lisp).

Kolem roku 2004 byla odstartována druhá vlna, když na scénu vlétly produkty s otevřeným zdrojovým kódem(open source), které jsou velmi jednoduché a jsou napsány kompletně v některém z objektově orientovaných jazyků, jako je například Java nebo C#. Můžeme zmínit například projekty Db4Objects nebo Perst.

3.2 Standarty OOSŘBD

Použití SŘBD v praxi není zdaleka tak jednoznačné , jak naznačují propagační či komerční periodika komentující šíření databázové technologie. Přes široké využívání SŘBD v hospodářské sféře nelze říct, že problémy reprezentace a zpracování dat jsou řešeny pro uživatele uspokojivě. Další vývoj SŘBD je

v současnosti ovlivněn více technologiemi z nichž nejznámější je objektově orientovaný přístup.

Tyto tendence mají původ především v potřebách praxe. Cílem je zlepšit možnosti stávajících SŘBD. Pro nové typy aplikací (jako jsou multimedia, hypermedia, kancelářské systémy požadující bohatě strukturovaný a odkazy provázaný text, obrázky , CAD aplikace apod.) a pro další aplikační domény (zahrnující např. CAD/CAM, telekomunikace, prostorové objekty, multimédia) se ukázaly být stávající SŘBD příliš těžkopádné, jelikož se v aplikační sféře objevily požadavky, na které tyto systémy nestačily. Nároky na databáze v těchto aplikacích se týkají jednak operací, které se s takovými strukturami mají provádět. Změny vyžadovala i vlastní databázová technologie. Objektová orientace posouvá databázovou úroveň mnohem blíže ke konceptuální[5]. Hlavní změnou v myšlení, kterou přináší objektově orientovaný přístup, je pohled na data v databázi. Data nejsou dána pouze hodnotami seskupenými do datových struktur, jako relace v RMD, nýbrž jsou vztažena k jistým (abstraktním) objektům, které mohou přímo odpovídat entitám reálného světa.

Nedostatky tradičních RDBMS v možnostech vypořádat se s novými aplikacemi, jež vyžadují souběžné modelování na datech a procesech s daty, vedly ke zrození 3. generace databázových systému – objektově orientovaných systémů řízení báze dat (dále jen OOSŘBD). Jako každá nová technologie v průmyslu i OOSŘBD neměly pevně určené standardy, když se objevily. Aby se rapidně prosadil jejich růst, uchází se několik standardů o definování objektově orientovaného datového modelu. To zahrnuje: Tři manifesty, ODMG Standard, JDO Standard a další.

3.3 Manifesty objektově orientovaných databázových systémů

Existuje několik standardů, jež definují objektově orientovaný datový model. V učebnicích jsou obvykle objektově orientované SŘBD identifikovány Manifestem skupiny ALTAIR. Základní požadavky na nově vytvářené objektově orientované databázové systémy jsou shrnuty ve třech dokumentech, které byly vytvořeny na začátku devadesátých let.

Tři manifesty pro OODBS

- rok 1898 – OODBS Manifest
- rok 1990 – Manifest třetí generace databázových systémů(Stonebraker)
- rok 1995 – Třetí manifest (Darwin a Date)

V prvním manifestu autoři stanovili 13 objektově orientovaných rysů, které by měly OOSŘBD obsahovat. Byl to spíše jen náčrt, než formální standard, který by směřoval určitým přímým směrem. V manifestu se praví, že objektové systémy nemusí být nutně následníci a dědicové systému relačních, a proto by výzkum prováděný v této oblasti měl být prováděn bez ohledu na současný technologický stav. V době uveřejnění manifestu bylo konstatováno, že na rozdíl od relačního modelu, který má formální základ, OOSŘBD jsou primárně založeny na experimentální aktivitě, nemají formální základ a existuje zde nebezpečí, že kterýkoliv systém jako první obchodně úspěšný se může stát de facto normou bez ohledu na dlouhodobé aktivity ostatních. Cílem Manifestu je spíše vylepšení terminologie, nežli standardizování jazyků. Tedy na rozdíl od autorů ODMG, kteří se koncentrovali jak na terminologii, tak na jazyky, chtěli autoři Manifestu vnést řád do roztříštěných experimentálních aktivit v této oblasti[6].

Autoři druhého manifestu rozdělili charakteristiky OOSŘBD do 3 skupin, povinné, výběrové a otevřené. Mimoto do detailů prezentovali 13 návrhů pro třetí generaci databázových systémů.

- **Povinné** – tyto charakteristiky musí být splněny, aby si systém zasloužil etiketu objektově orientovaného databázového systému. Dostupné komerční OOSŘBD odrážejí více či méně povinné rysy dané tímto Manifestem. Povinné rysy jsou díle sdruženy do dvou skupin.
  Systém musí být databázový, což znamená, že musí podporovat následující rysy :

  Persistenci, správu sekundárních pamětí, paralelismus, prostředky zotavení a prostředky pro kladení ad hoc dotazů.

  Systém musí být objektově orientovaný, což znamená, že musí mít následujících osm rysů: komplexní objekty, identifikaci objektů, zapouzdření, typy a/nebo třídy, dědičnost, upřesňování spolu s pozdní vazbou, výpočtovou úplnost a rozšiřitelnost.

- **Volitelné** – tyto charakteristiky vylepšují základní funkcionalitu sytému, ale nejsou povinné. Patří sem vlastnosti jako vícenásobná dědičnost, typová kontrola, distribuovanost, verzování, dlouhotrvající transakce.
- **Otevřené** – Návrhář systému si může vybrat z několika ekvivalentních akceptovatelných alternativ. Otevřené typy jsou uvedeny pouze v jediné skupině a jsou následující:

  programovací paradigma – uživatel by měl mít možnost použít libovolné programovací paradigma (procedurální, deklarativní, logické apod.)
  rozšíření základních datových typů – základní datové typy a konstrukce mají být rozšiřitelné různými způsoby pro podporu objektové orientace.

typový systém – vedle zapouzdření by měly být dostupné generické typy, omezení, sjednocení a funkce.

Třetí manifest bere ohled na budoucí databázové systémy, avšak svou jistou naivitou nepřinesl nic nového.

Manifesty objektově orientovaných systémů jsou často kritizovány mimo jiné za to, že pouze specifikují rysy objektově orientovaného přístupu v členění na povinné, nepovinné a volitelné a nenabízí tak ucelenou koncepci tohoto přístupu k databázím.

3.4 ODMG Standard

Neexistence normy objektově orientovaných systémů byla jedním z hlavních omezení jejich širokého využití. Proto dochází k různým aktivitám směřujícím k normalizaci OOSŘBD. Průlomem v poněkud nepřehledném světě modelů a standardů objektově orientovaných databází se stalo vydání standardu ODMG konsorciem vůdčích výrobců, prodejců a zainteresovaných stran pracujících na specifikaci objektově orientovaných databázových systémů, které návrh vyvinulo. Standard ODMG se stal nadějným východiskem z bezbřehosti objektově orientovaných přístupů minulosti. Více o tomto standardu si povíme v další části této bakalářské práce.

3.5 Vybrané implementace objektových databází

Při sepisování tohoto přehledu jsem si všiml jedné zajímavé věci týkající se podpory standardů. Některé produkty jsou velmi úzce svázány se standardy definujícími objektovou technologii v databázích (např. ODMG nebo JDO), zatímco jiné produkty z těchto standardů podporují jen vybrané základní rysy. Záleží samozřejmě na velikosti výsledného produktu a na podpoře ze strany výrobce. Je jasné, že malá volně šiřitelná objektová databáze nemůže se svou

základnou podporovat všechny rysy standardů a konkurovat tak velkým společnostem vyvíjejícím produkty vysoké kvality i funkčnosti. Zatímco malé produkty se snaží držet základních rysů definovaných ve standardech, velké produkty obsahují navíc velké množství dalších nadstandardních rysů. Při popisu velkých produktů jsem se proto zaměřil pouze na výčet těch nejdůležitějších rysů, která pokládám za prioritní.

3.5.1 Versant Developer Suite

Základní charakteristika

Versant Developer Suite(VDS) je sedmou generací objektové databáze, která byla vyvinuta ke zjednodušení vývoje a zvýšení výkonu aplikací v komplexních a heterogenních prostředích. Společnosti, jejichž aplikace jsou napsány v Javě, C++ , těží právě z implementování tohoto řešení. VDS poskytuje pokročilé databázové funkce v čele s vysoce stupňovatelným transakčním správním systémem pro distribuované víceuživatelské systémy. Tento OOSŘBD je předurčen pro vysoký výkon především díky podpoře multithreadingu, schopností vytvářet více sezení (multisession) najednou a také díky architektuře zamykání na objektové úrovni. Další výhodou je dotazovací mechanismus (engine) na straně serveru, neboť zpracování dotazů probíhá přímo na serveru, takže ke klientovi se dostávají už zpracované výsledky, což jednak značně ulehčuje přenosu a komunikaci a dále to odlehčuje zátěži na klientovi.

Java a C++ rozhraní

Java rozhraní poskytuje kompletní transparentní persistenci pro objekty bez nutnosti měnit model javovských tříd. Dále umožňuje javovským objektům ukládání a sdílení mezi aplikacemi bez nutnosti mapování. Podobě jako Java rozhraní, C++ rozhraní dovoluje transparentní persistenci C++ objektů.

Rozšiřitelnost a 64-bitová podpora

Hlavní přednosti VDS spočívají ve výkonu a rozšiřitelnosti. To zahrnuje podporu pro dnešní generaci 64-bitových operačních systémů. Díky této podpoře zmizelo omezení týkající se maximální velikosti souboru.

Sdílení objektů

Objekty mohou být sdíleny distribuovanými aplikacemi používající přímý navigační přístup. Stejné objekty uložené ve VDS databázi mohou být přístupné z Java aplikací, ale mohou být také přístupné a modifikované C++ aplikací a naopak.

3.5.2 GemStone/S

Základní charakteristika

GemStone/S je aktivní databáze, což znamená, že v databázi jsou kromě objektů uloženy i jejich třídy a jejich metody. Objekty mohou dostávat zprávy a ty provádět svůj kód přímo v databázi, aniž by byl příjemce zprávy přemístěn na nějakého klienta. GemStone/S Server je programován v dialektu Smaltalku nazvaném GemStone Smaltalk. Vychází z normy ANSI X3J20 a je optimalizovaný pro práci s velkým množstvím objektů uložených na disku, proto normu rozšiřuje. Vše je řešeno Smaltalkem – Smaltalk je *data definition language* (DDL), *data manipulation language* (DML) i *query language* (QL). Jiný jazyk se nepoužívá. Objekty jsou organizovány do tříd, které specifikují strukturu objektů a jejich protokol. Třídy jsou vázány vztahem jednoduché dědičnosti.

Architektura serveru

GemStone/S je víceuživatelský server. Ovládá proto:

- Autentifikace
- Optimistické i pesimistické řízení transakcí
- Zabezpečení přístupu až na úrovni jednotlivých objektů.
- Umožňuje snadnou změnu databázového schématu automatickou tvorbou a číslováním verzí tříd za plného chodu databáze.

Výkonnostní parametry

- Umožňuje uložit až miliardu objektů do celkové velikosti řádově 30GB
- Podporuje současnou práci řádově tisíců uživatelů
- dovoluje provádět řádově stovky transakcí za vteřinu
- je schopen dodat téměř lineární zrychlení výpočtu při přidávání procesorů
- databázi je možno rozdělit na více souborů umístněných na různých discích a počítačích, čímž umožníme paralelní čtení a zápis části databáze

3.5.3 Objectivity/DB

Základní charakteristika

Objektivity/DB je multithreadingový databázový systém, nabízející výjimečnou rozšiřitelnost a výkon. Vedle velké škály vývojových a administračních nástrojů poskytuje Objektivity/DB zjednodušený přístup ke správě komplexních dat, směřující k redukci operačních nákladů a ke zrychlení vývoje produktů. Objektivity/DB prezentuje sofistikovaný mechanismus shlukování objektů, logicky dovolující spojeným objektům seskupování na fyzické úrovni pro minimalizaci I/O operací a pro snížení režií týkajících se správy zámků v databázi.

Rozhraní programovacích jazyků

Objektivity/DB poskytuje programovací rozhraní pro Javu, C++, Smaltalk a SQL (přes ODBC ovladač).

Objektivity/DB nabízí plnou podporu pro programovací jazyk Java, včetně jazykové vazby podle standardu ODMG 2.0. Díky své unikátní "single process model" architektuře poskytuje vysoký výkon pro aplikace založené na Javě. Pro tuto architekturu je charakteristické, že rozhraní databáze běží ve stejném procesu jako JVM (Java Virtual Machine). Tím je tato databáze rychlejší, protože se neplýtvá cennými procesními cykly nutnými ke konstantnímu střídání procesů (přepínání mezi procesem, ve kterém běží JVM a procesem, ve kterém běží databáze).

Objektivity/DB rozhraní podporuje ANSI standard jazyka C++ a přidává persistentní verzi STL (Standard Templáře Library), která dovoluje vývojářům vytvářet, ukládat a získávat části databáze s využitím STL metod.

Objektivity/SQL++ poskytuje kompletní SQL rozhraní podle standardu ANSI. Dává tím programátorům do rukou nástroj pro přístup k objektové databázi. Toto rozhraní zahrnuje SQL3 standard, který oproti standardu SQL2, který oproti standardu SQL2 podporuje rozšířené databázové modelovací a objektové vlastnosti.

3.5.4 db4o

Základní charakteristika

Db4o je objektový databázový stroj. Je dodáván jako programovací knihovna (.jar nebo .dll), která se využívá při vývoji aplikací. Výjimečná jednoduchost

tohoto produktu vede ke snížení nákladů na vývoj, k urychlení vývoje, zlepšení údržby a v neposlední řadě také k využití principů znuvupoužitelnosti.

Správa databázového schématu

Objekty jsou analyzovány v době běhu programu. Schéma tříd všech uložených objektů je vytvářeno a udržováno automaticky. Přidávání a odstranění polí tříd je možné v reálném čase, bez nutnosti reorganizovat databázový soubor.

Ukládání objektů

Abychom umožnili ukládání objektů, není potřeba měnit definice tříd. Nepotřebujeme odvozovat z proprietárních bázových tříd ani implementovat specifické rozhraní, které by umožnilo uložení objektů v databázi. Nemusíme implementovat rozhraní pro serializaci ani modifikovat zdrojový kód. Objekty jsou ukládány tak jak jsou.

Jazykové vazby

Db4o rozumí všem obvyklým jazykovým konstruktům programovacích jazyků Java a C# (objekty, rozhraní, primitivní typy, pole, vícerozměrné pole, třídy kolekcí, neveřejné konstruktory, .NET struktury). Výsledky dotazu v db4o mohou být poslány do standardních kolekcí, které nabízejí tyto jazyky.

Snadná instalace

Databázový stroj se skládá pouze z jednoho souboru (db4o.jar / db4o.dll). Tento soubor ke knihovnou všech tříd, potřebných k obsluze a běhu této databáze. Instalace se provádí pouhým přidáním tohoto souboru k vašemu programovacímu prostředí.

Databázový soubor

Db4o ukládá data do jednoho databázového souboru. Záloha, obnova a správa rozdílných verzí databáze spočívá v nutnosti starat se pouze o tento soubor.

4 Objektově-relační databáze

4.1 Historie

Objektově-relační databázové systémy se vyvinuly z výzkumných projektů, které se započaly na začátku devadesátých let. Cílem výzkumů bylo rozšíření stávajících relačních konceptů o koncept objektový. Idea byla taková, aby zůstal zachován deklarativní dotazovací jazyk založený na predikátové logice. Za největší zmínku pravděpodobně stojí projekt Postgres kalifornské univerzity v Berkeley. Dva produkty, které se stále pokračují v těchto stopách jsou Illustra a PostgreSQL.

V polovině devadesátých let, se začaly objevovat první komerční produkty. Illustra, která byla později po koupi společnosti IBM přejmenována na Informix, Omniscience koupena společností Oracle a přejmenována na Oracle Lite a UniSQL. Tyto produkty můžeme označit jako ORSŘBD.

Mnoho z původních myšlenek bylo přidáno do standardu SQL:1999, proto se nyní všechny produkty, které implementují SQL:1999 nebo nejnovější SQL:2003 můžou chlubit tím, že jsou to objektově-relační systémy.

4.2 Stručná charakteristika

Objektově relační databázové systémy jsou jakýmsi kompromisem mezi staršími a osvědčenými relačními a novými objektovými databázovými systémy. Objektové systémy, známé již od 80-tých let se v praxi zatím moc nerozšířily, neboť ohromné množství uživatelů vlastní starší aplikace založené

na relačních systémech a přechod na objektové řešení by si vyžádalo i změnu tohoto aplikačního vybavení.

Zároveň však starší relační model databázového systému v dnešní době multimediálních informací již nedostačuje, protože relační systémy jsou uzpůsobené pro práci s jednoduchými datovými typy. Proto se v praxi začínají rozvíjet právě objektově relační systémy, podporující jak relační funkcionalitu, tak prvky objektových systémů.

Objektově-relační databáze nebo také objektově-relační systém řízení báze dat (ORSŘBD) vychází z klasických relační databázových systémů, navíc umožňují vývojářům obohatit databázi o své vlastní datové prvky a metody. Pojem objektově-relační databáze se někdy milně používá v souvislosti s objektově-relačními mapovacími nástroji. Tyto nástroje však pouze poskytují podobné funkce jako objektově-relační databáze, ve skutečnosti však používají klasické databázové systémy.

4.3 Hlavní rysy objektově-orientovaných databází

Základní rysy objektově-orientovaných databází oproti klasickým relačním jsou :

- podpora rozšíření základních datových typů
- podpora složitých objektů
- podpora dědičnosti
- podpora systému pravidel

Tyto základní rysy jsou pro objektově relační systémy určující, nutno ovšem poznamenat, že ne všichni dodavatelé ORSŘBD tyto rysy implementují.

Zároveň je však třeba zachovat konzistenci se stávajícími databázemi. Jedná se o :

- zajištění kompatibility s existujícími relačními databázemi
- ponechání jazyka SQL
- volitelně primární klíče
- existenci pohledů
- mechanizmus tvorby relací

4.3.1 Podpora rozšíření základních datových typů

Nejjednodušší formou uživatelem definovaných typů je tzv. oddělený typ (distinct type). Tento typ umožňuje, že dvě jinak ekvivalentní deklarace typů jsou zpracovávány jako samostatné typy. Klíčové slovo DISTINCT, použité v definici typu určuje, že výsledný typ má být zpracován odděleně od jakékoliv jiné deklarace toho samého typu. Například pokud jsou dva nové typy deklarovány jako :

CREATE DISTINCT TYPE americky_dolar AS DECIMAL (9,2)
CREATE DISTINCT TYPE kanadský_dolar AS DECIMAL (9,12)

Zde vytváříme dva nové uživatelem definované jednoduché typy americký dolar a kanadský dolar s rozdílným způsobem zaokrouhlování. Jakýkoliv pokus zpracovat jeden typ jako instanci druhého způsobí chybu, ačkoli oba typy mají stejnou reprezentaci. Tímto způsobem lze vytvářet jakýkoliv nový datový typ odvozený od nějakého základního datové typu.

Příklad odvození datové typu datum_narozeni :

CREATE DISTINCT TYPE datum_narozeni AS DATE

4.3.2 Podpora složitých typů

Kromě definování nových základních datových typů lze v dobrých objektově relačních databázových systémech definovat složité datové objekty využívající základní i rozšířené datové typy. Mohou však ještě využívat další speciální datové typy:

- pole , jsou podtypem collection
- pojmenované řádkové typy
- abstraktní datové typy

4.3.2.1 Pole

SQL:1999 přidává podporu polí. Představme si, že máme tabulku knihy, každá kniha může mít vícero autorů. Podle konvencí bychom museli například vytvořit pomocnou tabulku autoři knih, toto je ale již minulostí. Znamená to tedy, že SQL:1999 porušuje první normální formu? Tato otázka vyvstává, když si uvědomíme, že se nám v databázi můžou objevovat opakující se skupiny dat, které první normální forma zakazuje. Obecný názor je že ne, protože je víceméně implementačně závislé jakým způsobem se databázové produkty s poli vyrovnají.

```
CREATE TABLE knihy (
cena INTEGER
autori VARCHAR (30) ARRAY[8]
nazev VARCHAR (50));
```

V tomto případě jsme si vytvořili tabulku *knihy*, se třemi položkami (*cena, autoři a název knihy*) . Pole autoři může obsahovat až osm hodnot typu VARCHAR o třiceti znacích.

Příklad vložení knihy se dvěma autory do tabulky *knihy :*

INSERT INTO knihy VALUES
(102, '{"Tomáš Pokorný","Vít Janeček}', 'O vzniku světa');

Pokud bychom nyní chtěli získat všechny autory knihy „O vzniku světa“ posloužil by nám následující jednoduchý dotaz:

SELECT autori
FROM knihy
WHERE nazev = 'O vzniku světa'

Můžeme specifikovat i konkrétní index :

SELECT autori[1]
FROM knihy
WHERE nazev = 'O vzniku světa'

Tímto výčet možností nekončí, lze použít vícerozměrná pole, lze se dotazovat na velikost pole apod.

Problematika polí je samozřejmě daleko rozsáhlejší a přesahuje rámec této bakalářské práce. Důležité je samozřejmě jakou konkrétní databázovou platformu použijeme.

4.3.2.2 Pojmenované řádkové typy

Datový typ řádek lze požít jako definici k vytvoření nové tabulky. Data toho typu mohou být opět použity jako buňka řádku v jiné tabulce, tzv. nested tables (hnízděné tabulky).

Příklad vytvoření datového typu řádek, zde se jedná o vytvoření typu *auto* s vnořeným typem *TovarniUdajeAuta.*

```
CREATE ROW TYPE typTovarniUdajeAuta (
        znacka          VARCHAR (30 ) ,
        vyrobce         VARCHAR (30 ) ,
        datumVyroby  Date );

CREATE ROW TYPE typAuto (
        tovarniUdaje   typTovarniUdajeAuta,
        spz             VARCHAR (10),
        barva           VARCHAR (10));
```

Nyní již můžeme vytvořit tabulku:

```
CREATE TABLE AUTA OF typAuto ;
```

Lze ovšem vytvářet i tabulku pomocí nepojmenovaných typů

```
CREATE TABLE AUTA (
spz             VARCHAR (10),
barva           VARCHAR (10),
tovarniUdaje ROW (
```

znacka VARCHAR (30) ,
vyrobce VARCHAR (30) ,
datumVyroby Date));

Ke zpřístupnění komponent řádkového typu se používá tečková notace.

Příklad který vypíše všechny značky aut, která jsou modrá :

SELECT AUTA.tovarniUdaje.znacka
FROM AUTA
WHERE Auta.barva = 'modrá'

Místo relačního spojení lze do jedné tabulky uvést odkaz např. na konkrétní řádek v druhé tabulce. Při výběru dat z první tabulky před definovaný odkaz na řádek druhé tabulky lze vybrat i data z konkrétního řádku druhé tabulky, bez nutnosti definovat a poté provést spojení mezi těmito dvěma tabulkami. Výběr dat přes odkazy na jiné tabulky je rychlejší než přes provádění spojení tabulek.

V případě odkazu na řádek v druhé tabulce se do pole odkazu první tabulky uloží OID, tedy unikátní číslo objektu (řádku tabulky). Výběr dat přes definovaný odkaz se provádí kaskádní tečkovou notací.

4.3.2.3 Abstraktní datové typy

Abstraktní datové typy umožňují definování nových typů plně v kompetenci vývojáře, navíc poskytují zapouzdření atributů a operací (na rozdíl od řádkových typů)**.**

Abstraktní datový typ je uživatelem definovaný složený typ, který zapouzdřuje datové struktury spolu s funkcemi a procedurami, které jsou potřebné pro manipulaci s daty. Proměnné, které vytvářejí datové struktury nazýváme atributy. Funkce a procedury, které charakterizují chování objektu nazýváme metody.

Po nadefinování objektového typu příkazem CREATE TYPE je vytvořena abstraktní podoba nějakého objektu reálného světa. Tato předloha specifikuje pouze ty vlastnosti a chování objektu, které budou používány uvnitř aplikace. Například zaměstnanec může mít mnoho vlastností, ale pro potřeby aplikace je jich nutná pouze malá část.

Definice abstraktního datového typu je následující :

CREATE TYPE jméno typu AS
seznam atributů a jejich typů
nepovinná deklarace metod
údaje o děditelnosti

Další funkce lze deklarovat vně příkazu CREATE TYPE (nejsou svázány s ADT).

Příklad :

CREATE TYPE typZamestnanec AS (
cislo_zam INTEGER,
jméno CHAR (20),
adresa typAdresa,

datum_nastupu DATE,
zakladni_plat DECIMAL(6,2))
INSTANTIABLE,
METHOD odpracovana_leta() RETURNS INTEGER;
/pouze hlavičky/*
METHOD mzda () RETURNS DECIMAL;

CREATE METHOD odpr_leta
FOR typzam_stnanec
BEGIN … END ;

CREATE METHOD mzda
FOR typzam_stnanec
BEGIN … END;

Metody jsou vždy svázány s konkrétním abstraktním datovým prvkem, narozdíl od funkcí a procedur a jsou vždy uloženy ve schématu daného typu definovaného uživatelem. Metody se dědí a mohou být přetěžovány. To znamená, že můžeme mít několik metod stejného názvu, které se budou lišit pouze počtem či typem parametrů.

4.3.2.3.1 Reference na objekty

V některých případech nechceme v atributu uchovávat celý objekt, ale pouze referenci (ukazatel, odkaz) na něj. Proto byl v SQL:1999 zaveden typ reference. Uvažme případ, že máme danou databázi pracovišť a předmětů uložených na jednotlivých pracovištích. Vzhledem k centrální správě předmětů by bylo poněkud nevýhodné mít u každého pracoviště vnořenou tabulku předmětů. Navíc předměty mohou být rozděleny do několika kategorií a tedy i tabulek. Následující příklad ukazuje nastíněnou situaci, kdy předměty jsou

ukládány v jediné tabulce a u každého předmětu je reference na pracoviště, na němž se nachází :

```
CREATE TYPE typPracoviste AS  (
                zkratka  VARCHAR2(3),
                nazev    VARCHAR2(50)
);

CREATE TABLE kancelare OF typPracoviste

CREATE TYPE typPredmet AS  (
                nazev       VARCHAR2(30),
                ev_cislo    NUMBER(5),
                pracoviste  REF typPracoviste
);

CREATE TABLE predmety OF t_predmet
```

Každý objekt v objektově-relačních databázích je jednoznačně identifikovatelný pomocí OID – Object Identifier. OID je jednoznačný v rámci celého systému. Reference na objekty nelze považovat za referenční integritní omezení

4.3.3 Podpora dědičnosti

Podpora dědičnosti se kterou se můžeme setkat u objektově-orientovaných databází je jednou z dalších vítaných novinek se kterou přišel standard SQL:1999. Nyní nemusíme pro každý nový typ či tabulku, která je podtypem již existující zavádět zcela novou, můžeme využít dědičnosti.

4.3.3.1 Podtypy

Pokud chceme například zavést typ zaměstnance *řidič*, který je logickým podtypem *zaměstnance* použijeme následující zápis:

```
CREATE TYPE ridic UNDER typZamestnanec AS (
Zde můžeme nadefinovat další atributy a metody
Specifické pro povolání řidiče
) ;
```

Moderní objektově-relační databáze umožňují vícenásobnou dědičnost. Nový typ řidič dědí všechny atributy a metody svého rodiče. Navíc získává nově definované atributy a metody

4.3.3.2 Dědění tabulek

Dědění tabulek je další užitečnou novinkou. Dovoluje nám odvozovat od již existujících tabulek tabulky nové s tím, že si můžeme přidat nové sloupce. Navíc tabulky dědí od svých rodičů též triggery. Následující příklad nám ukazuje jakým způsobem dědění provést.

Vytvoříme tabulku *města* a podtabulku *hlavní města*.

```
CREATE TABLE MESTA (
        nazev     VARCHAR(20),
        pocet_obyvatel      float,
        nadmorska_vyska   int     -- (v metrech));

CREATE TABLE HLAVNI_MESTA UNDER MESTA(
        stat          char(2)):
```

Tabulka hlavní města dědí všechny prvky, které obsahuje tabulka města (název, počet_obyvatel, nadmořská_výška), navíc přidává atribut stát, který určuje ve kterém státě dané hlavní město leží.

Následující dotaz vrací názvy a nadmořskou výšku všech měst, tedy včetně hlavních, která leží v nadmořské výšce větší než 500 m.

```
SELECT nazev, nadmorska_vyska
FROM MESTA
WHERE nadmorska_vyska > 500;
```

Pokud bychom chtěli ten samý výsledek ovšem bez hlavních měst použili bychom následující zápis:

```
SELECT nazev, nadmorska_vyska
FROM ONLY MESTA
WHERE nadmorska_vyska > 500;
```

Parametr ONLY zde značí, že do výsledku dotazu mají být zahrnuty pouze data z tabulky města.

4.3.4 Podpora systému pravidel

Pravidla chrání integritu dat a usnadňují údržbu. Relační systémy nabízejí podporu pravidel v podobě triggerů, což je ovšem pouze podmnožina podporovaných pravidel v dobrém objektově relačním databázovém systému.

Pravidla mohou být čtyř typů:

pravidla aktualizace – aktualizace
pravidla dotaz – aktualizace
pravidla aktualizace – dotaz
pravidla dotaz – dotaz

Pravidla aktualizace – aktualizace jsou pravidla, které při aktualizace nějaké tabulky aktualizují zároveň i tabulku jinou. Tyto pravidla se nazývají v relačních systémech právě triggery. Syntaxe definování těchto pravidel je následovná:

CREATE RULE název_pravidla
ON {UDPATE | INSERT | DELETE} TO název_tabulky[.název_pole]
[podmínka WHERE]
DO aktualizační_dotaz;

Pravidla dotaz – aktualizace jsou velice vhodná ke kontrole výběru dat z definované tabulky. Lze tak totiž vytvářet logovací záznamy o přístupu k tabulkám, jež uživatele zajímají.

CREATE RULE název_pravidla
ON SELECT TO název_tabulky[.název_pole]
[podmínka WHERE]
DO atualizační_dotaz;

Pravidla aktualizace – dotaz představují tzv. výstrahy. Databázový systém jimi může upozornit uživatele na události při aktualizaci dat, které jej zajímají.

Uživatelova aplikace však musí umět tyto výstrahy přijímat, musí tedy těmto výstrahám naslouchat.

CREATE ALERT název_výstrahy (MECHANISM='CALLBACK');
CREATE RULE název_pravidla
ON {UDPATE | INSERT | DELETE} TO název_tabulky
[podmínka WHERE]
DO
RAISE ALERT název_výstrahy;

Pravidly dotaz – dotaz se dají definovat například pohledy (views). Principem těchto pravidel je totiž zaměnit dotaz SELECT, který definuje uživatel, za nějaký jiný SELECT[8].

CREATE RULE název_pravidla
ON SELECT TO název_tabulky[.název_pole]
[podmínka WHERE]
DO INSTEAD

4.4 Vybrané objektově-relační databáze

V současné době existuje nepřeberné množství databázových systému, které sebe označují za objektově-orientované většinou však implementují jen část z objektových vymožeností SQL:1999 a SQL:2003. Jde jak o drahé robustní komerční systémy, tak o open source projekty. Následující databázové systémy patří k těm lepším.

- DB 2 – IBM
- Informix – IBM
- OpenLink Virtuoso – OpenLink Software
- PostgreSQL - PostgreSQL Global Development Group
- Oracle – Oracle Corporation

Tabulka srovnání implementace základních objektových rozšíření tří databázových produktů.

	Typy	Metody	Typová dědičnost	Dědičnost tabulek
Oracle	ANO	ANO	ANO	NE
OpenLink Virtuoso	ANO	NE	ANO	ANO
PostgreSQL	ANO	NE	NE	ANO

Tabulka srovnání podpory nových datových typů

	Pole	Seznamy	Množiny	Multimnožiny	Objektové reference
Oracle	ANO	NE	ANO	ANO	ANO
OpenLink Virtuoso	ANO	NE	NE	NE	ANO
PostgreSQL	ANO	NE	NE	NE	NE

Zdroj : WikiPedia

5 Standard ODMG 3.0

V dnešní době hrají informace podstatnou roli v mnoha oborech lidské činnost. Tyto informace je třeba nějakým způsobem ukládat. Velmi oblíbenou formou je ukládání do databáze, respektive jejich využívání pomocí systému řízení báze dat.

5.1 Vývoj ODMG

Objektově orientovaný přístup v oblasti databází se v komerční sféře vyskytuje již od konce 80-tých let. V té době se každý výrobce snažil vytvořit svoji nejlepší implementaci. V důsledku toho vznikaly různé datové modely. Různé jazyky definice dat a různé jazyky pro manipulaci s daty. Tato rozdílnost však samozřejmě přinášela své problémy, největší, že aplikace nemohla být přenositelná na jiný systém. Proto se největší výrobci databázových produktů té doby sešli v roce 1991 ke společnému jednání, aby dojednali jeden společný standart. Vznikla skupina Object Databáze Management Group jako podskupina OMG – Object Management Group.

5.2 Základní cíle

Základním cílem ODMG je kromě přenositelnosti jednotlivých aplikací také překlenutí “object paradism mismatch“ – jde o neshodu relačních databázových systémů s objektově orientovaným paradigmatem programování.

Databázové jazyky obvykle nejsou výpočetně úplné, tj. nelze s nimi vypočítat libovolnou vyčíslitelnou funkci, jako tomu je u programovacích jazyků. Problém síly databázového jazyka se obvykle řeší začleněním do programovacího jazyka, což ovšem vede k tomu, že se získá strukturálně i

sémanticky nehomogenní prostředek. Cílem je vytvořit výpočetně úplný Objektově Orientovaný Systém Řízení Báze Dat pro překlenutí tohoto problému, a to tak, aby jeho databázový jazyk byl v konstruktech dostatečně homogenní. V podstatě jde o zjednodušení přístupu k datům pomocí objektově orientovanému paradigmatu[5].

5.3 Čeho se standard týká

Standard ODMG 3.0 se skládá z těchto komponent [1]:

- **objektového modelu, včetně databázových konceptů** – Obecný datový model podporovaný ODMG implementací je založen na OMG objektovém modelu. Objektový model ODMG je nadmnožinou již existujícího obecnějšího modelu Common Object Model, vytvořeného skupinou OMG. Cílem OMG modelu bylo zkonstruovat společného jmenovatele pro objektově orientované databázové systémy, objektově orientované programovací jazyky a další uplatnění. Objektový model ODMG je základem pro jazyk ODL, kterým lze vyjádřit schéma v objektové databázi a pro objektový dotazovací jazyk OQL. Databázové koncepty, které objektový model zahrnuje jsou perzistentnt (stálost údajů v databázi), správa transakcí a databází.
- **určuje jazyk specifikace objektů** – Existují dvě normy jazyků : ODL(Object Definition Language) – jazyk pro definici objektů (jazyk specifikující objekty) a OIF (Object Interchange Format) – formát výměny objektů.

ODL je specifikační jazyk používaný k definici datových objektů, které se přizpůsobují ODMG objektovému modelu, jinak řečeno ODL je standardizovaný jazyk pro specifikaci struktury databáze v objektově orientovaných termínech. ODL má mnohé vlastnosti společné s jazykem

definice rozhraní (IDL – Interface Definition Language) od OMG. ODL je striktně nadřazen OMG IDL pro definici databázového schématu. Objekty smí být definovány ve smyslu typů, atributů, vztahů a operací. ODL vytváří abstraktní vrstvu tak, že schéma generované v ODL je nezávislé na programovacím jazyku a databázi, což dovoluje aplikaci, aby byla přesouvána mezi vhodnými databázemi různými jazykovými implementacemi nebo jen přeložena do jiného definičního jazyku.

Jazyk OIF slouží pro zjednodušení přesunu dat mezi různými systémy.

- **určuje objektový dotazovací jazyk** – OQL(Object Query Language) jako aplikaci SQL na objektový model.. OQL je deklarativní (neprocedurální) jazyk pro dotazování objektů. Dotazovací jazyk OQL úzce souvisí s koncepcí dotazovací části standardu SQL – 92.
- **stanovuje jazykové (implementační) vazby** – ODMG pro různé objektově orientované jazyky, jakými jsou Java, C++ nebo Smaltalk. Tyto jazyky jsou nazývány OML (Object Manipulation Language), v překladu to znamená jazyky pro manipulaci s objekty.

5.4 Objektový model ODMG 3.0

ODMG definuje objektově orientovaný databázový systém, který integruje databázové přednosti se schopnostmi objektově orientovaných programovacích jazyků.

Manifest také přichází s podobnou definicí: mělo by se jednat o SŘBD a měl by to být objektově orientovaný systém. To jest, v celém možném rozsahu by měl být konsistentní se současnou nabídkou objektově orientovaných programovacích jazyků. Takže objektový model databází musí integrovat model jak objektově orientovaný, tak databázový. Objektový model podle standardu ODMG není založen na existujícím relačním modelu, ale je vystavěn z principů v objektového programován, není tedy obvyklým rozšířením relačního modelu .

Objektový model můžeme shrnout takto :

- **měl by být SŘBD** – tj. měl by zajišťovat perzistenci dat na vnějších pamětech, paralelní přístup více uživatelů, spolehlivost dat a možnost dotazů.
- **měl by být objektově orientovaný** – tj. měl by připouštět existenci identity objektů, umožnit práci s objekty, typy a třídami, využívat zapouzdření (objektů a operací, které s nimi manipulují), polymorfismus a dědění, strukturu složitých objektů, podporovat rozšiřitelnost a zahrnovat výpočetně úplný jazyk.

Tato definice se dále ještě rozšiřuje o rysy související s novými databázovými aplikacemi. Jedná se o operační charakteristiky, jako je řízení verzí, dlouhé a hnízděné (nested) transakce, možnost formulace a použití integritních omezení (constraints), možnost zapojení více serverů do clusteru a konstrukce speciálních metod.

Nyní připomeneme stručně význam některých databázových charakteristik.

Perzistence znamená přetrvání dat po ukončení procesu, který s nimi pracuje, a možnost jejich znovupoužití v dalších procesech.
Řízení vnější paměti se týká návrhu a manipulace s vyrovnávacími paměťmi indexů a rozmístění dat v paměti.
Souběžnost zpracování více uživateli vede k využití pojmů transakce, rozvrhu, zámků.
Spolehlivost je podporována prostředky pro zotavení z chyb, jako jsou deníky transakcí, protokoly zaznamenávající historií aktualizace struktury dat.
V dotazovacím jazyku je obsažena možnost dotazů, tak jak je to běžné pro relační model v SQL.

5.4.1 Objekty a jejich identita

Základními stavebními bloky objektového modelu jsou objekty a literály (hodnoty). Hlavní rozdíl mezi objektem a literálem spočívá v tom, že literál představuje pouze hodnotu bez jakéhokoliv objektového určení.

Rozdíl mezi objekty a literály je zřejmý: objekt je instancí abstraktního údajového typu (třídy), literál je instancí jednoduchého údajového typu, jako jsou například celá čísla a řetězce. Objekt by byl v relační databázi řádek tabulky (kterou můžeme považovat za třídu) a hodnota by byla obsahem jednotlivých sloupců v tabulce. Objekt je popsán čtyřmi základními vlastnostmi, kterými jsou identifikátor, jméno perzistence a struktura.

Co dělá objekt objektem je jeho identita. Je implementována pomocí identifikátoru objektu, což je v celém systému unikátní označení objektu, většinou nazývané OID (Object Identifier). OID se používá jako reference na instanci, Každý objekt musí mít nějaký OID:

Charakteristika OID je následující :

1. OID je nezávislý na datech, které objekty obsahují (vnitřní hodnota dat není použita ke generaci identifikace), stručně řečeno je nezávislý na hodnotě objektu.
2. OID je generován objektovým systémem (uživatel nebo program nemají kontrolu nad identifikací). OID se generuje při vytváření objektu.
3. OID existuje po celou dobu existence objektu
4. OID se v průběhu života objektu nemění, OID je hodnota stabilní a jednoznačná v celé databázi.

Standart nespecifikuje, zda jsou jednotlivé OID opětovně použitelné při zrušení objektu, hovoří však o tom, že uživatel (klient) OID nevidí. Domény referenčních údajových typů (směrníky) obsahují právě tyto OID hodnoty. Když chceme přistupovat k jednotlivým objektům, nemůžeme použít OID, neboť ty vidí jen databáze, použijeme tedy názvy objektů. Tyto názvy databáze konvertuje na OID.

OID nelze zaměňovat s pojmem klíče v relačním modelu dat. OID zůstává stále stejný, mění se pouze hodnota objektu, která de facto reprezentuje stav objektu. Rozlišujeme tedy identitu objektů – jde o tentýž objekt a rovnost objektů – mající stejnou hodnotu.

Pomocí identity objektů lze snadno řešit problém referenční integrity známého z relačních databází. To co se například v SQL musí zapisovat explicitně(cizí klíče), není v objektově orientovaném systému řízení báze dat nutné, protože pomocí OID jsou objekty automaticky fyzicky “provázané” v souladu s vazbami specifikovanými v definicích objektů. V relační databázi je však vazba daná referenční integritou pouze logická.

Každý objekt může (ale nemusí) mít navíc své jméno, které je určeno k odkazování na tento objekt v programu. Systém je schopen určit objekt podle jeho jména.

Perzistence objektů určuje, zdali se jedná o objekt perzistentní nebo o objekt transientní, tedy existující pouze po dobu běhu aplikace. Perzistence lze dosáhnout pojmenováním objektu, nebo je dosažitelnosti na vnější paměti. O perzistenci budeme hovořit i v další části kapitoly.

Struktura objektu určuje způsob vytvoření objektu, zdali je atomický nebo složitý. Složitými objekty jsou například kolekce set, list, array, bag apod.

5.4.2 Typy a třídy

Typ v OOŘBD shrnuje společnou strukturu množiny objektů se stejnými charakteristikami. Kromě struktury jsou zadány operace, které lze s objekty daného typu provádět. Typ koresponduje s pojmem abstraktní datový typ používaným v informatice ještě před formulací OOSŘBD. Pojem třídy je něco více něž pojmem typu a obvykle je v různých OOSŘBD pojímán různě. Zahrnuje v sobě možnost produkovat nové objekty, to znamená některou formu operace “new”, dále pak možnost sdružovat všechny potřebné objekty, také zvané instance dané třídy pohromadě v tvz. kontejneru objektů. Ke kontejneru může uživatel přistupovat pomocí daných operací. Množina všech (právě daných) objektů dané třídy se také nazývá extenzí třídy. Typ je tedy obecně vztažen spíše ke konceptuálnímu pohledu, třída je použita při implementaci typu[1].

Pojem třídy lze využít k vytváření objektů „vyšší“ úrovně. Na třídu se lze totiž dívat opět jako na objekt, tj. lze vytvořit třídu tříd – metatřídu. K popisu metatříd slouží atributy tříd. Např. počet prvků třídy by mohl být takovým atributem. Uvedené atributy a operace instance tříd dané metatřídy nedědí. Většina OOŘSBD však metatřídy nepoužívá.

5.4.3 Perzistenstnost

Vytváření rozsáhlejších aplikací je téměř vždy spojeno s nutností spravovat údaje, přetrvávající i po ukončení činnosti programu, který je vytvořil – tyto údaje budeme označovat jako perzistentní. Může se jednat o jednoduché údaje jako jsou například konfigurační parametry, informace o předchozí činnosti aplikace, nebo může jít o objemná data uložená v souborech nebo databázích. Například úkoly z oblasti plánování, návrhu nebo řízení obvykle vyžadují vyhodnocování a shromažďování informací po delší dobu, přepínání mezi různými činnostmi a externími procesy a z toho vyplývající nutnost správy perzistentních dat.

Při použití klasických programovacích jazyků typu Pascal, C/C++ nebo Java musí obvykle aktualizaci perzistentních dat proběhnout následující kroky[2] :

1. návrh formátu vnější reprezentace dat
2. návrh formátu vnitřní reprezentace dat v programu
3. provedení vstupních operací pro načtení dat a jejich transformaci do formátu vnitřní reprezentace
4. aktualizace údajů
5. transformace do formátu vnější reprezentace a provedení výstupních operací pro uložení dat.

Je zřejmé, že kromě návrhu vlastního algoritmu pro aktualizaci údajů se musí tvůrce aplikace zabývat také explicitními převody mezi vnějším a vnitřním formátem uložení dat a zabezpečením operací vstupu/výstupu včetně ošetření případných chyb.

S nástupem databázové technologie došlo k oddělení správy vnější reprezentace perzistentních údajů a metod umožňujících k těmto údajům efektivní přístup. Vznikla standardizovaná programátorská rozhraní typu OBDC, JDBC nebo OLE DB, která zajišťují přístup k perzistentním údajům do značné míry nezávisle na konkrétním použitém systému správy databáze. Na posloupnosti kroků , které musí programátor při práci s takto uloženými údaji provést, se však principiálně nic nezměnilo. Místo návrhu vnější reprezentace dat musí programátor definovat schéma databáze a k tomuto schématu a k tomuto schématu pak najít zobrazení do vnitřních datových struktur a zpět. Nutnost konverze mezi vnější a vnitřní reprezentací vyplývá nejen z toho, že databázové systémy obvykle podporují jen velmi úzkou množinu datových typů, ale také z principiální odlišnosti přístupových metod. Například výsledkem dotazu v jazyce SQL je obecně posloupnost záznamů, která je ovšem přístupná pouze sekvenčním čtením po jednotlivých záznamech.

Cílem perzistentních programovacích jazyků je zejména potřeba odstranění typové impendance a v konečném důsledku zavedení zcela transparentního přístupu k perzistentním údajům. Tohoto cíle lze dosáhnout při návrhu programovacího jazyka dodržováním souboru určitých zásad, označovaných jako princip ortogonální perzistence. Tento princip bude vysvětlen v následující části.

5.4.3.1 Vlastnosti perzistence

Z hlediska doby života můžeme údaje, s nimiž pracuje konkrétní aplikace, rozdělit na transientní a perzistentní. Transientní údaje se vytvářejí i zanikají v průběhu jednoho spuštění aplikace. Perzistentní údaje se uchovávají i po ukončení aplikace, která je vytvořila a jsou dostupné při dalším spuštění aplikace nebo i pro jiné aplikace. Transientní údaj může být povýšen na perzistentní. Toto povýšení může nastat na základě explicitního požadavku nebo implicitně jako důsledek pravidel, k nimž se dostaneme později. Vlastnost perzistence spočívá v možnosti přidělit komponentám modelu libovolnou dobu života vyžadovanou aplikací. Pokud mohou všechny komponenty modelu bez ohledu na typ potenciálně nabývat všech možných rozsahů doby života, hovoříme o ortogonální perzistenci. Tento pojem je obvykle spojován s následujícími vlastnostmi [2]:

- ortogonalita datových typů – podpora perzistence pro všechna data bez ohledu na typ
- identifikace perzistence – mechanismus zjištění, které objekty jsou perzistentní, není závislý na datových typech.
- nezávislost na perzistenci – programy nejsou závislé na tom, zda pracují s perzistentními nebo transientními údaji. Uživatel neřídí přesun mezi krátkodobou a dlouhodobou pamětí.

Zajištění těchto vlastností je závislé na konkrétní metodě, která je pro dosažení perzistence použita. Nejvýznamějším důsledkem ortogonální perzistence pro programátora je dokonale transparentní prostředí, ve kterém lze s tranzitními I perzistentními objekty zacházet zcela shodně, a úplné odstranění typové impendace. Není-li některý z uvedených principů dodržen, musí programátor psát něco navíc – různé varianty algoritmů pro perzistentní a transientní data, konverze dat mezi typy nepodporujícími perzistenci na typy, které perzistenci umožňují, explicitní označování perzistencích objektů a pod.

5.4.3.2 Zajištění perzistence údajů

Perzistenci údajů lze zajistit různými metodami. Nejjednodušší z nich je klasický přístup, při kterém využíváme ad hoc formátů pro uložení dat na konci činnosti programu a jejich obnovení při novém spuštění. Tento přístup byl představen již v úvodu. Jeho hlavní nedostatky jsou obtížná spolupráce různých aplikací (nutná znalost formátu uložení), nutnost realizovat přístup k datům explicitně a nutnost provádění konverzí formátů.

Další představenou možností bylo ukládání dat v relačních databázích a využití speciálních knihoven zajišťující transformaci mezi datovými modely. Spolupráce aplikací je méně obtížná vzhledem ke standardizaci rozhraní, ovšem explicitní přístup k datům a typová impendance zůstávají zachovány. Vážným problémem je zde nutnost normalizace údajů, která vede ke značně komplikovaným transformacím při ukládání i zpětném načítání složitých objektů (např. takových, které obsahují zanořené objekty či jejich kolekce). Ukládání dat v objektových databázích práci s perzistentními daty značně zjednodušuje, neboť nevyžaduje transformaci mezi různými datovými modely jako v předchozím případě. K dispozici jsou standardy pro vazbu objektových databází a programovacích jazyků jako C++ nebo Java. Přístup k perzistentním datům však stále není transparentní, zejména kvůli nutnosti explicitního zamykání, zpracování transakcí, případně explicitního nastavování příznaků modifikace objektů.

5.4.3.3 Perzistentní programovací jazyky

Zajištění ortogonální perzistence obvykle vyžaduje více, než může přímo poskytnout prostředí hostitelského programovacího jazyka. Princip ortogonality perzistence vyžaduje, aby hodnota libovolného datového typu mohla být perzistentní. V kontextu objektově orientovaných programovacích

jazyků, které rozlišují primitivní datové typy jako je char, int , long, double a objektové typu definované na základě tříd, se ortogonalita obvykle vztahuje pouze na objektové typy.

Princip identifikace perzistence se vztahuje na způsob, jakým jsou určeny perzistentní objekty. V perzistentních jazycích se setkáváme s následujícími variantami[2]:

- perzistence dědičností – perzistentní jsou pouze ty objekty, které jsou instancí následníka určité speciální bázové třídy. Například standard ODMG 93‘ vyžadoval,l aby třídy perzistentních objektů byly odvozeny od třídy PersitenceCapable. Tato varianta porušuje jak princip ortogonality, tak i princip nezávislosti na perzistenci.
- perzistence instanciací – perzistentní jsou pouze ty objekty, které byly označeny jako perzistentní při vytváření. Princip ortogonality zde porušen není. Ale vzhledem k tomu, že musíme dobu života volit v okamžiku vytvoření objektu a vytvořený transientní objekt pak nemůže být povýšen na perzistentní, je tento přístup v rozporu s principem nezávislosti. Navíc v případě, že perzistentní objekt odkazuje na jiný transientní objekt, může být po uložení a zpětném načtení porušena referenční integrita modelu.
- perzistence dosažitelností – perzistentní jsou pouze ty objekty, které jsou dosažitelné z určitých pevně definovaných objektů – tzv. kořenů perzistence. Tento přístup je zcela v souladu s principy ortogonální perzistence.

5.4.4 Zapouzdření

Jde o princip přístupu k objektům pouze přes dané rozhraní a ne jiným způsobem (tj. hodnoty atributů objektů nejsou obecně přístupné z

programovacího jazyka). Klient nemá jinou možnost než používat nabídnuté operace, což vede kromě jiného, I k podpoře ochrany dat.

Objekt má rozhraní a implementaci. Rozhraní je dáno specifikací operací, které mohou být prováděny na objektu, tj. jménem a argumenty. Operace tvoří jedinou viditelnou část objektu. Připomeňme, že operace se v OOSŘBD obvykle nazývá metoda. Implementace má část datovou a procedurální, která slouží k reprezentaci stavu objektu a implementaci každé operace. Zapouzdření dovoluje měnit programátorovi datovou část, aniž by se změnily programy pracující s odpovídajícími objekty (podpora datové nezávislosti).

5.4.5 Rozšiřitelnost

Rozšiřitelnost znamená možnost definovat nové základní typy a posléze nové typy pomocí konstruktorů pro vytváření typů (externě definované typy). To vede k požadavku mít možnost vytvořit I efektivní implementaci takovýchto typů. Rozšiřitelnost je spíše požadavek na architekturu než na objektově orientovaný přístup (vyskytuje se I v rozšířených relačních systémech). Pojem rozšiřitelnost se používá zejména ve funkčním smyslu, tj. jde o zahrnutí uživatelem specifikovaných operací. Problém ovšem spočívá v tom, na které úrovni databázové architektury je vhodné je zahrnout.

5.4.6 Transakce

Přístup k transakcím (databázovým operacím, které splňují určité vlastnosti - ACID) je samozřejmě také řešen. Je však třeba zmínit, že sémantika transakcí je určená jen pro perzistentní objekty. Model transakcí, jako více databázových operací seskupených do jedné atomické jednotky, která se buď vykoná celá anebo se nevykoná vůbec je založený na protokolu uzamykání s konvenční maticí kompatibility (tedy ostatní transakce nemají k danému objektu přístup, dokud daná transakce neproběhne). Jak je zvykem I v relačních databázích

podporujících transakce, v případě, že narazíme na nějaký uzamknutý objekt, můžeme volit mezi čekáním na odemknutí nebo zrušením celé transakce.

Znalosti z oblasti relačních databází hovoří o tom, že může nastat stav, kdy se nějaký objekt zablokuje a nastane tzv. deadlock. Standard nám s tímto problémem nepomůže a řešení je třeba nechat na tvůrcích jednotlivých databázových systémů. Musím podotknout, že zde neexistují vnořené transakce a nepoužívají se hierarchické zámky ani jiné sémantické informace. I když systém implicitně uzamyká objekty, aplikace je může uzamknout I sama, tj.j explicitně pro případ kdy by to bylo potřebné[1].

5.5 ODMG 3.0 ODL

ODMG 3.0, jako standard pro objektově orientované databáze, hovoří kromě jiného také o věcech, jako je vztah objektů a hodnot (literálů) v objektových databázích, o zapouzdření, o podmínkách, na jejichž základě považujeme objekty za identické, o tom jak určujeme třídy a vztahy mezi nimi (asociace), jak lze používat specializace tříd (dědičnost), co je v objektových databázích perzistentnost, jak souvisí s transakcemi a jaké databázové operace můžeme použít. Jazyk ODL je navržen tak, aby plně vyhovoval objektovému modelu ODMG a byl nezávislý na konkrétní implementaci programovacího jazyka. Jeho hlavní využití spočívá v tvorbě specifikací objektů, tedy tříd a rozhraní.

5.5.1 Datové typy

ODMG definuje následující datové typy pro použití v objektových databázích. Rozlišují se tři typy literálů : automatické, strukturované a kolekce.

- Atomické literály (atomické datové typy) korespondují s hodnotami základních typů a jsou předdefinovány. Základními datovými typy objektového modelu jsou celá čísla (klíčová slova long, short unsigned long, unsigned short), dekadická čísla s pohyblivou čárkou (klíčová

slova float, double), logické hodnoty (boolean), znaky (char), řetězce znaků (string) a výčtový typ (enum).

- Strukturované literály jsou hodnoty vytvořené pomocí speciálního n-ticového konstruktoru (klíčové slovo struct). Patří sem již vytvořené struktury date, interval, time, timestamp a struktury, které si vytvoří uživatel v programu v závislosti na svých potřebách. Kolekce literálů specifikuje hodnotu, která je sadou objektů nebo hodnot stejného typu.
- Kolekce v objektovém modelu může být množina (set<t>), multimnožina (bag<t>), seznam (list<t>) nebo pole (array<t>), kde t je typem objektů nebo hodnot obsažených v kolekci. Dalším typem kolekce je slovník (vícerozměrné pole) (dictionary <k,v>), kde k je klíčem a v hodnotou. Slovník představuje uspořádanou posloupnost vztahů mezi klíčem a hodnotou. Tato struktura může být použita k tvorbě indexů nad skupinou hodnot. Kolekce (kolektivní typy) jsou skládány z více elementů, a proto potřebují (a mají) iterátory pro přechod mezi jednotlivými elementy.

Objekty se také dělí na atomické, kolekce a strukturované. Literál kolekce a objekt kolekce se liší pouze tím, že první z nich nemá OID. Co se týče identity objektů tak v textu jsem již dříve popsal OID jako prostředek určování identity objektů.

Typy objektů jsou specifikovány dvojím způsobem. Může jít o rozhraní (interface), tedy o popis abstraktních objektů, jež nemohou být instanciovány. Tyto definici slouží k popisu operací, které mohou dědit uživatelem definované třídy. Pokud má být objekt instanciován, tzn. mají být vytvářeny objekty tohoto typu, je namísto klíčového slova interface použito slovo class, tj. je definována třída. Připomeňme, že pro objekty kolekce a strukturované objekty jsou definována standardní rozhraní. Abychom objekt dokázali vytvořit a použít,

potřebujeme mít třídy. Vztah tříd a rozhraní definujeme v jazyku ODL. ODMG je implementační model a proto potřebujeme definovat vztahy mezi jednotlivými třídami [1]. Vztahy mezi objekty jsou určené prostřednictvím referenčních atributů (které používají OID).

5.6 OQL – objektově orientovaný dotazovací jazyk

Podobně, jako se SQL používá pro přístup k datům v relačních databázových strojích, v objektově orientovaných databázových strojích používáme dotazovací jazyk OQL (Object Query Language). Ten vznikl na základě jazyka OOSQL. Když se podíváme na OQL, na první pohled je jasné, že jde o jazyk který byl postavený na základech standartu SQL-92. Jako jazyk, kterým máme přistupovat k jednotlivým objektům, musí umožňovat přístup ke všem konstrukcím objektově orientovaných databázových systémů. To znamená přístup k datům prostřednictvím rozšíření. V případě kolektivních datových typů musí umožňovat navigaci mezi jednotlivými prvky. OQL obsahuje objektové rozšíření pro objektovou identitu, komplexní objekty, definici cest, volání operací a dědičnost. Podle definice má být OQL na rozdíl od SQL funkcionálním jazykem a musí podporovat celý typový systém (a umožňovat volání metod). Jazyk sám nemusí být výpočetně úplným (ani SQL není výpočetně úplný jazyk), stačí nám, když dokážeme volat metody. Nesmíme opomenout, že jazyk OQL nepodporuje rekurzi. Dotazy OQL mohou volat metody v programovacích jazycích vázaných standardem ODMG tím, že bude zapuštěn v těchto ODMG jazykových vazbách [3].

Jazyk OQL udržuje objektovou integritu (stav objektu, neporušenost, úplnost objektu) používáním metod objektu spíše, než pomocí svých vlastních aktualizačních operátorů. Přesněji řečeno, OQL je nekompletní jazyk, jelikož

nejsou explicitně poskytovány aktualizační operátory jako insert nebo delete. K aktualizování se využívá rysů programovacích jazyků. Naštěstí OQL může volat metody měnící stav objektu jako create(), add() a delete(), aby získal podobnou funkčnost. Výsledkem dotazů v OQL jsou objekty nebo literály, které korespondují s definicí ODL. Syntaxe OQL je podobná syntaxi dotazovací části jazyka SQL, do které jsou přidány další rysy přímo související s objektovými rysy databáze.

5.6.1 Vlastnosti jazyka OQL

Existuje několik vlastností jazyka OQL[1] :

- ortogonálnost – možnost kombinovat operace podle typového systému
- ohraničenost, účinnost, bezpečnost a optimalizovatelnost – dotazy v OQL mohou využívat volání metod, které poté zabezpečí výpočtovou úplnost. Tento přístup má důsledky v tom, že o účinnost, bezpečnost a optimalizovatelnost je třeba se postarat v rámci operací objektů.
- uzavřenost – objektový model podporuje výsledky relačních dotazů, dotazy obsahující a generující objekty.

5.6.2 Jádro OQL

Ve standardu ODMG je k dispozici gramatika a popis OQL. Následující gramatika popisuje jádro tohoto jazyka.

Nejprve uvedeme všechny unární a binární operace.

binop = {**and,or,intersect,union,except, =, <, >, <=, >=, +, -, /, ||, mod**}
unop = {**first, last, max, min, avg, sum, count, distinct, listtoset, element, flatten, and, not, -**}

Netypované dotazy jsou pak definovány jako

```
q ::= b | f |  i | c | s
    | x
    | bag(q, …, q) | set(q, …, q) | list(q, …, q) | array(q, …, q)
    | dictionary((q,q), …, (q,q)) | struct (l:q, …, l:q)
    | C(l:q, …, l:q) | q.l | (C)q
  | q[q] | q in q | q() | q(q, …, q)
   | forall x in q:q | exists x in q:q
   | q binop q | unop(q)
  | select [distinct] q
    from (q as x, …, q as x)
    where q
    [group by (l:q, …, l:q) ]
    [having q]
    [order by (q asc | dest, …, q asc | desc) ]
```

kde b, f, c, s jsou po řadě typu boolean, čísla s pohyblivou řádovou čárkou, celá čísla, znaky a řetězce. Označení x a l (labels) lze brát ze spočetné množiny identifikátorů. C je ze spočetné množiny jmen tříd.

5.6.3 Blok SFW

Dotazy jazyka OQL mohou být celkem triviální, jako např. Názvy objektů, termy, konstanty, rozšíření, volání konstruktorů, atd., přičemž komplikovanější dotazy používají tzv. blok SFW – Select From Where (podobně, jako je to v jazyku SQL).

SELECT – v této části bloku můžeme vykonávat volání metod (operací) a subdotazů. Výsledek dotazu může být komplexní typ, pokud ho tak nadefinujeme. Dotaz select, jak je to i v jazyku SQL vrací multimnožinu

hodnot, které najde v databázi podle zadaných kritérií. Pro omezení výsledku na množinu (ne multimnožinu) použijeme klíčové slovo distinct :

```
select distinct struct (m.nazev, projekty :
                                (
                                      select p.projektID
                                       from m.zucastnuje_se as p
                                 )
                                )
from Pracovnici as m;
```

Výsledek : Set < Struct <nazev: String, projekty: Bag < Long >>

FROM – V této části zadáme kolekci objektů nebo hodnot, jako rozšíření tříd, referenčního atributu, komplexního atributu, výsledku volání metody anebo subdotazu . V této části se vykonává automatický převod List(seznam) a Array (pole) na Bag (multimnožina). Příklad :

```
select m.nazev
from ( select p vede_projekt
from Projekt as p
where p.status = “uzavreny”) as m
where m.vydelek > 9000;
```

WHERE – Tato část obsahuje booleovský výraz, který může obsahovat tyto prvky: predikáty, subdotazy, trojhodnotová logika (nil => undefined, predikáty is_undefined, is_defined), logické spojky(and, or, not), podmíněné spojky (and then, or else).

Vstupními body do databáze jsou pojmenované perzistentní objekty. Pro většinu dotazů je představují jména extenzí, tedy jména uvedená klíčovým slovem extent (nebo ve složce extent). Dále tato jména reprezentují pojmenované perzistentní objekty a kolekce (většinou množiny) objektů třídy, ve které jsou definovány [1].

Najdi jméno oboru, který má číslo 15.

SELECT o jméno FROM o in Obory WHERE o cislo=15

Například objekt Obory, který vznikl extenzí třídy Obor, je typu set<Obor>, Osoby typu set<Osoba>, atd. Pojmenovaný však může být i jednotlivý objekt. Tyto vstupní body používáme k vytvoření proměnné (v našem případě o), která prochází všechny objekty v kolekci zastoupené tímto objektem (jde o typ Iterator). Do výsledku jsou pak vybrány všechny objekty, pro které je splněna podmínka ve složce where. Z těchto výsledků jsou na výstup dotazu vybrány atributy, které jsou ve složce select. Výsledkem daného dotazu tedy je multimnožina bag<string> všech jmen oborů, které mají číslo 15, protože atribut o.jmeno je typu string. Pokud bychom použili za klíčovým příkazem select klíčové slovo distinct, výsledkem by byla množina.

V OQL jsou tři různé způsoby, jak specifikovat proměnnou o, která prochází kolekcí. Jsou to :

o in obory
obory o
obory as o

Použití pojmenovaných objektů není nijak limitováno. Každý pojmenovaný atomický objekt nebo kolekce objektů může být vstupním bodem do databáze.

5.6.4 Spojení (join)

Spojení jsou známá z relačních databází, kde se používají na spojení dat ze dvou tabulek na základě stejných hodnot ve sloupcích. Podobné požadavky jsou také u objektových databází, proto jsou samozřejmou záležitostí. Tak, jako v relačních databázích, I v objektových databázích můžeme zapsat explicitně pro spojení, které by za normálních okolností nebylo možné vyčíst ze schématu databáze, např.

Které osoby mají jméno totožné s názvem ulice jedné z adres.

```
SELECT  p
FROM  p in Osoby
g in (SELECT DISTINCT  b.v_dome FROM  b IN Byty)
WHERE  p.jmeno = q.nazevUlice
```

Operace spojení můžeme často obejít použitím výrazů cesty.

5.6.5 Výrazy cesty

ODMG 3.0 OQL je navigační jazyk, což představuje možnost sledovat referenci od získaného objektu k výběru jiného objektu. Výrazy cesty jsou navigační mechanismy. Výraz cesty začíná proměnnou instanciovanou s objektem, následovanou tečkou nebo šipkou, operátor přístupu k objektu je následován libovolně dlouhou sekvencí referenčních jmen a teček nebo šipek končící jménem atributu nebo jménem vztahu. Když pracujeme s objekty a strukturami, k jejich atributům se musíme dopracovat – používáme výrazy cesty. Tyto výrazy známe už z jazyků podporujících strukturované

programování, odkud se přenesly do objektově orientovaných programovacích jazyků [1].

Předpokládejme, že z je instance typu zaměstnanec. Tyto dva ekvivalentní zápisy výrazu vrací státní poznávací značku auto, které řídí ředitel.

z.reditel.vuz.spz
z -> reditel -> vuz -> spz

Platí však jedno omezení. Mezičlánky cesty musí být reference skládající se z jediného elementu. Výraz cesty nemůže obsahovat kolekci v cestě, tzn. cesta musí být jednoznačná.

Následující výraz není povolený, i když zucastnuje_se je plnohodnotná reference:

o.zucastnuje_se.jmeno

ale správně je:

SELECT m.jmeno FROM o.zucastnuje_se as m

Pokud chceme mít vícero plnohodnotných referencí v jediné cestě, použijeme dotaz podobný tomuto:

o.dite.adresa.mesto.jmeno

Poněvadž dítě je atributem kolekce, odkazujeme se na více objektů, což znamená, že je poskytnuto více cest k odkazování se na mnoho jmen měst. Ke

sledování více cest musíme použít jiný způsob. Jak ukázal minulý případ, řešením je položit výraz cesty v OQL poddotazu.

SELECT d.adresa.mesto.jmeno FROM o.dite AS d

Klauzule from obsahuje cestu z objektu Osoba o ke kolekci atributů dítě. V tomto případě se každé d stane instancí dítěte. Klauzule select sleduje u každé instance město a dotaz vrací bag<string> vyplněný městem.

Doposud jsme hovořili jen o jednoduchých dotazech. Nyní si uvedeme komplikovanější.

Dej adresu každé osoby i jména a adresy svých dětí.

```
SELECT struct
(
    os: o.jmeno,
    os_Adresa: o.adresa,
    os_Deti :
    (
        SELECT struct
        (
            di_jmeno: d.jmeno
            di_Adresa: d.adresa
        )
        FROM d in o.Deti
)
    )
FROM o in osoby
```

Kdo je šéfem zaměstnance Jakuba Steinbauera a jaké má služební auto :

```
SELECT struct
(
    sef_Jmeno: o.jmeno,
    sef_Prijmeni: o.prijmeni,
    sef_SluzebniAuto: o.SluzebniAuto.SPZ
)
FROM  o in
(
    SELECTt z.zamestnany.firma.sef
    FROM  z in Zamestnany
    WHERE  z.Jmeno = Jakub and z.prijmeni = Steinbauer
)
```

5.6.6 Výsledek dotazu

Dosud jsme mluvili jen jak zapsat dotaz. Musíme se však zamyslet I nad tím, co nám konkrétní dotaz vrací. V relačních databázích to byly relace. Výsledek OQL dotazu může být jakýkoliv typ, který lze vyjádřit v objektovém modelu ODMG. Dotaz nemusí být zadán *select* blokem. Například výsledkem dotazu :

Najdi všechny obory

Obory

je kolekce všech perzistentních objektů typu *Obor*, tj. typ výsledku je *set<Obor>*. Podobně by to bylo i s pojmenovaným atomickým objektem,

výsledkem by byl tento objekt. Na atributy objektu se můžeme odkazovat pomocí tečkové notace, například :

studenti.adresa.ulice
studenti.studuje.jmeno

Tyto výrazy samozřejmě označují dotazy. Výsledek prvního z nich je typu *bag<string>*, tj. multimnožina všech ulic. Je-li Katedra k, pak pro

k.lide

obdržíme výsledek typu *set<Ucitel>* (množina všech učitelů, kteří jsou členy dané katedry).

Pokud chceme získat výsledek v nějaké složitější struktuře, můžeme použít klíčové slovo *struct*.

```
SELECT  struct
(
        jmeno_ucitele: struct
        (
                jmeno: o.jmeno_osoby.jmeno,
                prijmeni: o.jmeno_osoby.prijmeni
        ),
        pl: o.plat,
        titul: o.titul,
        predmety:
        (
                SELECT  struct(cislo: p.cislo, jmeno : p.jmeno)
```

FROM p in o.vyucuje

)

)

FROM o in ucitele

WHERE o.vek() < 30 order by plat asc, titul desc

Proměnná o prochází přes všechny učitele, proměnná p přes všechny předměty, které daný učitel vyučuje. Na příkladě je také patrné, jak lze vnořovat dotazy do sebe. Konstrukt order by se chová podobně jako v SQL. Specifikuje, podle kterých atributů se má výsledek třídit (asc – vzestupně, desc – sestupně) a výsledkem je pak seznam. V tečkové notaci mohou být použity atributy, vztahy i operace, které se případně dále odkazují na další atributy, vztahy či operace. Při dotazování je nutné ještě přihlédnout k dvěma aspektům výsledku dotazu[1].

5.6.6.1 Výsledek v souvislosti se sémantikou obsahující objekty

Výsledek dotazu obsahující objekty je kolekce objektů anebo samostatný objekt. V souvislosti s tím vyvstává množství problémů, např. problém použití množinových operací. Nemáme informace o výpočtu typu výsledku při heterogenních typech dat jako například :

konstrukter union planovac_vyroby

Jednoduchý výběr jako dotaz obsahující objekty vrátí kolekci objektů.

SELECT p FROM Projekt as p WHERE status "ukonceny"

Výpočet jediného objektu jako kolekce objektů skládající se z jediného elementu pomocí element.

element (SELECT p FROM Projekt as p WHERE projektID = 4711)

Můžeme též uskutečnit změnu typu, jak jsme na to zvyklí z jiných jazyků. Problém je však to, že výsledek dotazu není možné považovat za třídu a nezařazuje se do hierarchie specializace (dědění)

SELECT (Konstrukter) p
FROM Pracovnici as p

5.6.6.2 Výsledek v souvislosti se sémantikou generující objekty

Dotaz generuje nové objekty, proto je samozřejmě nutné spustit konstruktor.Ten je určený v předdefinovaném typu. Když je určený v předefinovaným, nemůžeme použít dynamické typové konstrukce a nemůžeme mít ani dynamické včlenění ho hierarchie specializace. O generování nové hodnoty OID platí, že nemá žádnou souvislost s existujícími OID hodnotami. Pokud vytváříme nový objekt tak, ten zavádí do databáze redundance. O zabezpečení perzistentnosti výsledku dotazu se nemluví. Příklady pro sémantiku dotazů generujících objekty.

Dotaz jako čisté volání konstruktoru:

Produkt (produktCislo:0815, popis:’”lednicka”, status:”hotova”, hmotnost:30)

Dotaz jako Blok SWF:

SELECT Vyrobek

```
(
        produktCislo: p.produktCislo, popis: p.popis,
        status: p.status, hmotnost :p.hmotnost,
        naklady: 1359
)
FROM Produkt as p
WHERE status = “ve_vyrobe”
```

5.6.7 Ostatní koncepty dotazů, další vlastnosti OQL

Kvantifikátory (forall a exists) známé z SQL, jsou v OQL funkce. Třídění (order by) je podobné tomu z SQL a je možné podle více atributů, avšak výsledkem je vždy seznam.

```
SELECT  p
FROM Osoby p
ORDER BY  age desc, name asc
```

5.6.7.1 Seskupování hodnot, operátor seskupení

Seskupování (*group by*) je vždy vázané na blok SFW, přičemž opět, jako v případě třídění, I seskupování je možné podle více atributů. Lze také použít klauzuli *having* (pro zadání výběrových podmínek). Složka *group by* má v OQL podobnou sémantiku jako v SQL. Jedná se o explicitní odkaz na kolekci objektů spojených do skupin.

Najdi všechny dvojice <*v*,*p*>, kde *v* je vyučující a *p* počet doktorandů, které vede.

```
SELECT  struct(osoba, pocet_doktorandu: count(partition))
FROM  d IN  doktorandi
GROUP BY  osoba: d.vedouci
```

Klíčové slovo *partition* je použito na jednu skupinu, tj. zde kolekci doktorandů, které vede stejný školitel. Syntaxe operátoru *group by* je **group by f_1:e_1, …, f_k:e_k,** kde *fi* je atribut podle kterého se seskupuje a e_i výraz. Výsledkem aplikace operátoru seskupení je množina.

set<struct(f_1:e_1, f_2:e_2, …, f_k:e_k, partition:bag<B>)>

jinak

set <struct <status: enum (…), partition:bag <projekt>>>

kde f_i je návratový typ výrazu e_i, *partition* je jméno atributu a B je proměnná definovaná ve složce *where*, která prochází po objektech pojmenovaného objektu. Složka *having* může být použita jako filtr nalezených skupin, podobně jak tomu je v SQL.

Najdi všechny dvojice <k,p>, kde k je katedra, která má více než 10 učitelů a p je průměrný plat na té katedře.

```
SELECT katedra, prum_plat(SELECT  p.plat FROM  p IN partition)
FROM  u IN  ucitele
GROUP BY  katedra: u.z_katedry
HAVING COUNT(partition) > 10
```

Výsledkem vyhodnocení dotazu je (multi)množina kateder, které mají více než 10 učitelů. Druhým atributem ve výsledku je průměrný plat.

5.6.7.2 Agregační funkce, kvantifikátory

Agregovaná funkce operující nad kolekcemi jsou *min*, *max*, *sum* a *avg*. Jejich sémantika je podobná jako v SQL. Funkce *count* vrací hodnotu typu integer. Funkce *min, max, sum* a *avg* vracejí hodnotu stejného typu jako jsou prvky uvažované kolekce[1], např.

COUNT (s IN studenti_s_profilovanym_predmetem(35))
AVG(SELECT s.vekQ FROm s IN studenti WHERE s.adresa.mesto = "České Budějovice")

Poznamenejme, že agregované operace jsou aplikovány na skupiny prvků příslušného typu a jejich výsledky mohou být použity v další části dotazu.

Najdi všechny předměty, na které se zapsalo více než 50 studentů.

SELECT p FROM p IN predmety WHERE COUNT(p.studenti) > 50

Vidíme, že OQL je budován "funkcionálněji" než SQL. Konstrukty je možné skládat jako funkce. Příkladem toho je aplikace agregační funkce na množinu danou pomocí *select*. V SQL by bylo nutné psát *select count x* apod. Kromě toho ve výrazech dotazu je možné používat aplikace operací, všude tam, kde je to typově možné.

Příkladem je například použití operace vek(). Kdyby nebyly použity závorky, pak na uživatelské úrovni nemusí být rozlišitelné, zdali jde o atribut, tj. data z databáze, či o operaci, tj. vypočítaná data.

OQL má také predikát náležení do množiny a kvantifikátory. Nechť *v* je proměnná, *c* je kolekce nebo dotaz s výsledkem typu kolekce, *b* je výraz

(podmínka), jejíž výsledkem je *true* nebo *false* a *e* nechť je prvek stejného typu jako prvky v kolekci *c*. Pak:

(e in c) je *true* pokud je *e* prvkem *c*

(forall v in c : b) je *true* pokud všechny prvky skupiny *c* splňují podmínku *b*

(exists c in v :b) je *true* pokud existuje alespoň jeden prvek skupiny *c* splňující *b*

Výsledkem vyhodnocení dotazu

SELECT u FROM u IN ucitele
WHERE "Databáze" IN (SELECT p.jmeno FROM p IN u.vyucuje)

Je (jednoprvková) množina učitelů, kteří vyučují předmět Databáze.

5.6.7.3 Výrazy a operace pracující se setříděnou, indexovanou kolekcí

Jak již bylo výše uvedeno, seznamy a pole mají operace, které umožňují získání i-tého, prvního nebo posledního prvku. Navíc jsou k možné operace na získání podseznamu (nebo spíše podkolekce) nebo na spojení dvou seznamů. OQL umožňuje tyto operace nad seznamy a poli.

first (SELECT u FROM IN ucitele ORDER BY u.plat desc)
(SELECT u FROM u IN ucitele ORDER BY u.plat desc)[0:2]

První z dotazů vrátí učitele s nejvyšším platem, druhý první tři. V hranatých závorkách se uvádí index prvního a posledního požadovaného prvku.

5.6.7.4 Získání prvku z dotazu, který vrací jednoprvkovou kolekci

Typickým výsledkem dotazu je kolekce prvků. Většinou jde o multimnožinu, množinu, pokud je v dotazu slovo *distinct*, nebo jde o seznam, pokud je v dotazu *order by*. Pokud chceme, aby výsledek byl jen jeden element, musíme použít klíčové slovo *element* tak, jak to ukazuje následující příklad. Pokud výsledkem původního dotazu nebude kolekce s právě jedním prvkem, pak operátor element způsobí výjimku.

element (SELECT s FROM s IN studenti
WHERE s.jmeno_osoby.jmeno=”Jakub” AND
s.jmeno_osoby.prijmeni=”Bauer”)

Výsledkem dotazu je student, který se jmenuje *Jakub Bauer*. Typ výsledku je *student*. Připomeňme, že v SQL taková péče o typování není. Jednoprvková množina daná pomocí *select* se v jistých kontextech považuje za prvek.

5.6.8 Operace na kolektivních typech

Operace: listtoset: přemění seznam na množinu

flatten: zjednoduší dělené množiny {{1,2},{3,5}} se změní na {1,2,3,5}

union (sjednocení), intersect (průnik), except (rozdíl)

first na seznamy – vrátí první prvek seznamu

5.6.8.1 Pojmenované dotazy

Rozložení složitějších dotazů na poddotazy je OQL založeno na konceptu pojmenování dotazů. V relačních databázových systémech dokážeme napsat procedury v jazyku, které fungují jako pojmenované dotazy. V OOSŘBD jsou pojmenované dotazy dané standardem. Vytvoříme je pomocí *define as*. Ve složce *define* se zapisuje jméno dotazu, které musí být unikátní vůči všem

pojmenovaným objektům, jménům tříd, operacím a funkcím ve schématu. Pokud má nový identifikátor stejné jméno jako již existující pojmenovaný dotaz, je předchozí definice přepsána. Pojmenovaný dotaz v OQL může navíc mít ve své definici parametry (argumety)[1].

Čeho tím dosáhneme, je zjednodušení dotazů, kromě toho to můžeme považovat za prvotní koncept pohledů.

```
define De_Monti_Projekty AS
SELECT o
FROM pracovnici AS  m, m.vede AS  o WHERE  prijmeni = "De
Monti"
SELECT o FROM  De_Monti_Projekty AS  d, d.prirazeny AS  o
ORDER BY  status
```

další příklad definuje pojmenovaný dotaz, který má jako parametr ročník studia. Výsledkem je množina profilových předmětů, které mají studenti v daném ročníku.

```
define profilove_predmety_studentu(d) AS
SELECT  DISTINCT  s.profilovy_predmet FROM  s IN studenti
WHERE  s.rocnik = d;
```

5.6.9 Volání metod v OQL

V ODMG 3.0 OQL můžeme stejně dobře jako vlastnosti vyvolávat metody. Návratový typ metody volané v OQL se musí shodovat s návratovým typem definovaným v metodě. Volání metody s s návratovým typem void je neplatné. Metody, jež nemají žádné vstupní argumenty, nevyžadují při svém volání

v OQL kulaté závorky, které jsou jinak povinné při volání v objektově orientovaných programovacích jazycích.

Metoda vek() je metodou bez argumentů třídy Osoba. Následující dotaz vrátí počet osob ve městě České Budějovice s věkem přesahujícím 20 let.

```
SELECT COUNT(*)
FROM Osoba o
WHERE o.adresa.mesto = 'České Budějovice' AND o.vek > 25
```

Metody vracející komplexní objekt mohou být provázány jako by byly ve vztahu. Třída *Osoba* obsahuje metodu *auto()*, která vrací objekt *Auto* asociováný s osobou.

```
SELECT COUNT(*)
FROM Osoba o
WHERE  o.adresa.mesto = 'České Budějovice' AND  o.vek > 20
AND  o.auto.model = 'Škoda'
```

Metody s argumenty vyžadují pro vyjmenování argumentů použití syntaxe se závorkami. Následující dotaz vrací jméno a věk každé osoby a jejího předka. Metoda maPredka() má jako svůj argument objekt Osoba.

```
SELECT  o.osoba, o.vek, q.jmeno, q.vek
FROM Osoba o, Osoby q
WHERE  o.maPredka(q)
```

5.6.10 Shrnutí OQL

Dotazovací jazyk je jazyk vhodným jak pro aplikaci, tak i pro databázi. Poskytuje těsný vztah mezi objektem aplikace a uloženým objektem. Názorně je to vidět v definici a manipulaci s daty v dotazech. Cílem tvorby standardu je podpora přenositelnosti databázových aplikací a překlenutí problému "impendance mismatch". Standard určuje mnoho věcí, avšak ponechává prostor pro jednotlivé implementace OOSŘBD a pro mnoho jiných věcí, např. Model vnořených transakcí. Není také absolutně jasná úloha ODL a chybějí důležité koncepty databází, jako jsou : *kontrola přístupu, pohledy, role, migrace objektů, podmíněné prostředí, deklarativní podmínky integrity* apod.

5.7 ODMG jazykové vazby

OQL může být zabudovaný (vložený) do zdrojového kódu napsaného v některém z následujících objektově orientovaných programovacích jazyků – Java, C++, SmallTalk prostřednictvím jazykové vazby. Dotazy mohou také vyvolat metody napsané v jakémkoliv ze zmíněných implementačních jazyků. Standardní ODMG jazykové vazby definují OML (object manipulation language), který obohacuje programovací jazyky podporou persistentních objektů. Vazby též obsahují podporu pro OQL, navigaci a transakce. OML umožňuje vývojářům pracovat uvnitř jednotného jazykového prostředí bez nutnosti používat samostatný dotazovací jazyk[1].

5.7.1 C++ vazba

ODMG C++ vazba poskytuje přímé jazykové rozšíření, zajišťující vhodné prostředky pro vytváření, pojmenování a manipulaci s objekty, dále mazání objektů, transakce a jiné databázové operace. Tato jazyková vazba, připouští vytváření perzistentních tříd pomocí dědičnosti. OQL a OML mohou být

roztroušeny, takže OQL dotazy smí být přímo vestavěny do aplikačního kódu a naopak OQL může volat OML.

5.7.2 Java vazba

ODMG vazba na Javu používá zavedené jazykové zvyklosti a styly, jež jsou vlastní pro prostředí a programátory. Instance existujících tříd se mohou stát perzistentními bez nutnosti měnit zdrojový kód. Persistence je zajištěna dosažitelností – jakmile je transakce potvrzena, tak jakýkoliv objekt, jež může být dosažen z kořenového objektu v databázi, se automaticky stává persistentním. ODMG vazba na Javu přidává třídy a další konstrukty do javovského prostředí, aby obohatila toto prostředí o možnosti ODMG objektového modelu, včetně kolekcí, transakcí, a databází, bez provádění změn v sémantice jazyka. Vazba na Javu je dále rozšířena v odvozeném standardu JDO (Java Data Objects).

5.7.3 Smalltalk vazba

Tato vazba na Smaltalk zajišťuje schopnost *ukládat, získávat nazpět a modifikovat persistentní objekty* ve Smalltalku. Vazba zahrnuje mechanismy pro volání OQL, procedur pro transakce a operace v databázi. Tak jako je tomu ve vazbě na Javu, objekty ve Smalltalku se stávají persistentními pomocí dosažitelnosti. To znamená, že se objekt stane persistentním, jakmile je referencován jiným persistentním objektem v databázi. Dále platí, že pokud není objekt nadále dosažitelný, je odstraněn. Tato vazba přímo mapuje všechny třídy definované v ODMG objektovém modelu na třídy Smalltalku. Kolekce a operace v objektovém modelu jsou mapovány ke standardním kolekcím a operacím. Vztahy, transakce a databázové operace jsou také mapovány do konstruktů jazyka.

6 Závěr

Ve své bakalářské práci jsem nastínil aktuální problematiku objektových a objektově-relačních databází. Podrobně jsem rozebral z jakých myšlenek vycházejí a jakými standardy se řídí. Čtenář si může vytvořit základní obrázek pro rozhodnutí, který z těchto dvou systémů využít. Ve světě informačních technologií je velmi těžké odhadovat budoucnost, ovšem můžeme předpokládat, že problémy které brání masivnějšímu přechodu k objektovým databázím (konservatismus, zkušenosti) budou odstraněny a objektové databáze se systémům založeným na relačním řízení báze dat minimálně vyrovnají.

7 Seznam použitých zdrojů

[1] ODMG 3.0. Dostupné na internetu.
http://de.geocities.com/bartkozo/odmg30.pdf

[2] Beneš Miroslav – Perzistence v objektově orientovaných jazycích
http://objekty.pef.czu.cz/2000/benes_m.pdf

[3] Lin Caixue – Object-Oriented Database systems: A Survey.
http://www.soe.ucsc.edu/~lcx/courses/cmps277/cmps277-project.pdf

[4] Pokorný Jaroslav – Odkud a kam kráčíte databáze. 1. vydání, Masarykova Univerzita, Brno 2000. s. 85-104, ISBN 80-210-2428-3

[5] Lu Dingwei – An Introduction to Object-Oriented Database Implementation.
http://www.cas.mcmaster.ca/~emil/courses/ootopics00/reports/Nail.pdf

[6] Atkinson Malcolm – Object-Oriented Database System Manifesto.
http://www.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html

[7] Jirsák Filip - ORM Framework Hibernace..
http://nb.vse.cz/~zelenyj/it380/eseje/xjirf01/xjirf01.htm

[8] Colombův blog.
http://colombo.vilecka.net/colombo.index.php?akce=clanek&id=433

[9] Wikipedia, The Free Encyclopedia
http://www.wikipedia.org