Objektové databáze

Martin Švec

1. července 2003

1 Proč objektové databáze?

Univerzální databázové systémy (*database management systems – DBMS*) v současné době představují nejrozšířenější prostředek pro uchovávání a manipulaci s daty komerčních aplikací. Zajišťují relativně jednoduchou správu rozsáhlých databází, integritu, efektivní vyhledávání a zpracovávání informací, odolnost proti poruchám a výpadkům, souběžný víceuživatelský přístup a další výhody.

Tento úspěch databázových systémů byl však docílen za cenu maximálního zjednodušení základního datového modelu a operací. *Relační datový model* – nejběžnější model používaný v dnešních databázových systémech – omezuje strukturu a vztahy uchovávaných dat na pouhou množinu tabulek nad předdefinovanou množinou základních datových typů. Nespornou výhodou tohoto modelu je jednoduchost a v důsledku toho i snadná standardizovatelnost a přenositelnost. Nevýhodou však je rozdílnost schématu reálných dat od vnitřního tabulkového modelu databáze. Veškeré vztahy mezi daty lze reprezentovat pouze tabulkami, což u aplikace s komplikovanějším datovým modelem vede k množství tabulek vzájemně provázaných pomocnými odkazy, tzv. klíči. Tím dochází ke ztrátě přehlednosti, databáze se stává hůře spravovatelnou a budoucí změny v datovém modelu aplikace nutí programátory k citelným zásahům do jejich tabulkové reprezentace.

Relační databáze jsou a zřejmě i nadále zůstanou hlavním prostředkem pro správu dat komerčních aplikací, které jsou charakteristické velkým objemem údajů s jednoduchou strukturou. Řada aplikací, například z oblasti designu, multimédií, geografických systémů apod., však potřebuje takový datový model, který umožní lepší korespondenci mezi složitými reálnými daty a jejich reprezentací v databázovém systému. Tímto modelem je *objektový model dat*.

Objektový model dat v databázových systémech vychází ze známých principů objektově orientovaného modelování a programování. Je však dále obohacen o techniky perzistence, reprezentace vztahů, dotazování, transakčního přístupu apod.

V následující kapitole jsou stručně popsány základní principy objektového modelu. V dalších kapitolách je pak věnována pozornost výše uvedeným charakteristickým rysům objektově orientovaných databázových systémů (*object oriented database management systems – OODBMS*) a především standardu ODMG.

2 Základy objektové orientace

2.1 Objekty a třídy

Objektově orientovaný model je založen na dekompozici informací z reálného světa na tzv. *objekty*. Objektem se rozumí každá (i strukturovaná) entita, která je jednoznačně a nezávisle identifikovatelná v rámci určitého kontextu okolního světa. Objekt tak má jednoznačnou *identitu*, každé dva i jinak datově shodné objekty jsou vzájemně odlišitelné. Identita objektu je určena *identifikátorem* (*object identifier – oid*), který je generovaný systémem, unikátní, neměnný po dobu existence objektu, skrytý pro programátora i koncového uživatele.

Objekty jsou charakterizovány pomocí *tříd*. Třída je abstraktní popis objektu, určuje datové složky objektu a operace (nazývané *metody*), které lze nad objektem provádět. Každý objekt je instancí nějaké třídy, od jedné třídy je možné instanciovat obecně neomezený počet strukturálně shodných objektů. Dále rozlišujeme *rozhraní* třídy, tj. popis atributů a operací, od *implementace*, tj. kódu popisujícího činnost operací.

2.2 Literály

Kromě objektů se v rámci objektově orientovaného modelu zavádí i pojem *literálu*. Literál je datová entita určitého *datového typu*, která však na rozdíl od objektu nemá vlastní identitu. Literály se obvykle vyskytují jako datové atributy objektů. Množina operací nad datovým typem literálu je pevně stanovená, není možné ji měnit.

S objekty a literály souvisí pojem *proměnlivosti* (*mutability*). Proměnlivost je chápána jako schopnost měnit data při zachování identity – v tomto smyslu jsou objekty proměnlivé, protože je možné měnit hodnoty jejich datových složek a přitom si ponechávají původní identitu. Naproti tomu literály proměnlivé nejsou.

2.3 Operace

Existuje několik základních typů operací nad objekty. Každý objekt má jeden či více *konstruktorů*. Účelem konstruktoru je inicializovat objekt v okamžiku jeho vytvoření. Dále je součástí každého objektu tzv. *destruktor*. Destruktor je volán v okamžiku rušení objektu a jeho cílem je provést úklid objektu před jeho odstraněním.

Důležitou operací nad objekty je *kopírování*. Rozlišují se tzv. mělké (*shallow*) a hluboké (*deep*) kopie. V případě mělké kopie dojde ke zkopírování atributů objektu, avšak všechny případné odkazy na jiné objekty dále ukazují na stejné objekty jako v originálním objektu. Naproti tomu u hluboké kopie dojde nejen ke zkopírování atributů kopírovaného objektu, ale současně se vytvoří i kopie objektů, na které se původní objekt odkazoval.

Dalšími typickými operacemi jsou metody pro zjišťování a přiřazování hodnot atributů, metody provádějící výpočty a manipulace s atributy objektu, metody produkující uživatelský výstup atd.

2.4 Zapouzdření, skrývání informace

Jak už bylo řečeno, součástí třídy je i definice operací, které lze nad objektem provádět. Okolí objektů má však přístup pouze k rozhraním operací, vlastní implementace operací je vždy před okolím skryta. Jedná se o typickou vlastnost objektově orientovaného přístupu, která zvyšuje míru abstrakce a nezávislosti objektů.

Kromě skrývání implementace je možné skrývat i části rozhraní. Atributy a operace lze označit za veřejné (*public*, přístupné z okolních objektů) nebo za soukromé (*private*, přístupné pouze v kontextu daného objektu). Některé systémy zavádí ještě jednu úroveň, obvykle nazývanou *protected* (chráněné). Takto označené atributy a operace jsou přístupné v objektech dané třídy a v objektech tříd z této třídy zděděných.

2.5 Dědičnost

Dědičnost je další typickou vlastností objektového modelu. Vychází z myšlenky, že některé třídy můžou být specializovanou verzí jiné třídy, resp. určitá třída je zobecněním jedné či více jiných speciálnějších tříd. Je-li třída zděděná z jiné třídy, dědí všechny její atributy a operace. Dále může doplnit nové atributy a operace, případně předefinovat operace zděděné.

Existuje dědičnost *jednoduchá* a *vícenásobná*. Jednoduchá dědičnost umožňuje třídě dědit pouze z jedné nadřazené třídy, v případě vícenásobné dědičnosti může třída dědit vlastnosti z většího počtu tříd.

2.6 Polymorfismus, pozdní vazba

Polymorfními operacemi se označují operace, které lze provádět nad objekty různých tříd. Přitom činnost operace se může lišit podle třídy objektu, nad kterým je prováděna. V objektově orientovaných programovacích jazycích se polymorfismus metod obvykle omezuje na třídy, které jsou ve vztahu generalizace/specializace.

S polymorfními operacemi souvisí pojem *pozdní vazba*. Pozdní vazbou se označuje způsob volání polymorfních operací, kdy aplikace při volání operace dynamicky za chodu programu zvolí kód metody (vybere příslušnou implementaci) na základě třídy objektu, nad kterým je metoda volána.

3 Perzistence v objektovém modelu

3.1 Co je perzistence?

Vlastnost *perzistence* je zřejmě nejdůležitejší vlastností všech databázových systémů, ať už relačně nebo objektově orientovaných. Obecně ji lze definovat jako schopnost systému trvale uchovávat data nejen po dobu svého běhu, ale i mezi vypnutím a opětovným spuštěním.

Běžné programovací jazyky mají obvykle kvalitní podporu pro správu dat po dobu běhu programu (tato data jsou označována jako *tranzientní*, dočasná). Jejich nedostatkem je však nízká podpora pro práci s daty, které je třeba udržovat i po ukončení programu (tzv. *perzistentní*, trvalá data). Naproti tomu databázové systémy jsou navrženy právě pro

správu údajů perzistentního a nikoliv tranzientního charakteru. Jelikož však běžné aplikace potřebují pracovat jak s dočasnými tak i trvalými daty, vzniká zde nutnost aby kvalitní databázový systém pokrýval současně schopnosti programovacího jazyka i databázového přístupu.

Je-li systém označován jako perzistentní, kromě výše uvedené definice perzistence jsou od něj očekávány i další vlastnosti:

(a) data v paměti i v perzistentním úložišti mají stejnou strukturu, přinejmenším na logické úrovni;

(b) libovolný údaj může být, bez ohledu na typ, označen za tranzientní či perzistentní.

Tyto vlastnosti spolu s vlastností trvalého uchovávání dat jsou označovány termínem *ortogonální perzistence*.

Perzistenci na úrovni objektově orientovaných databázových systémů spočívá ve schopnosti perzistentně udržovat libovolné objekty různých tříd. Oproti relačnímu modelu však objektově orientovaný model vede k heterogenním, složitě strukturovaným datům, provázaným množstvím odkazů. Důsledkem je nutnost sofistikovaných mechanizmů perzistence, zachovávajících efektivitu databázového systému i pro objektově orientovaná data. V následujících odstavcích budou proto diskutovány základní otázky, které je třeba vzít v úvahu při návrhu modelu perzistence OODBMS.

3.2 Druhy perzistence

V zásadě lze mezi modely perzistence najít tři základní typy:

(a) Perzistence na *úrovni sezení*. Program používající tento typ perzistence na konci běhu uloží obraz veškeré své paměti na disk a při opětovném spuštění jej opět vcelku načte (typické pro prostředí Smalltalku).

Tento model zjevně není pro potřeby OODBMS vhodný, protože nemá žádnou kontrolu nad tím, co všechno se ukládá, proces ukládání je jednorázový, časově náročný a málo bezpečný.

(b) Perzistence na *úrovni souborů*, tj. založená na principu ukládání a otvírání souborů (používáno textovými editory apod.) Ani tento typ perzistence není pro OODBMS vhodný. Typickým problémem je zde např. absence transparentního přístupu k datům bez ohledu na aktuální úložiště (paměť, disk), jak je požadováno u databázových systémů a jiné problémy.

(c) *Ortogonální perzistence*. Model ortogonální perzistence zajišťuje ukládání dat ve stejném tvaru a struktuře jako v paměti, transparentní otvírání a ukládání, načítání dat po malých částech dle potřeby (*on demand*), i stejný systém perzistence pro data a pro kód. Představuje tak optimální model splňující většinu požadavků OODBMS.

3.3 Co vše ukládat?

Je zjevné, že ukládat všechny objekty vyskytující se v OODBMS do perzistentního úložiště by bylo neefektivní. Po dobu běhu systému vždy existuje řada tranzientních objektů, které jsou potřeba pouze pro dočasné výpočty a není nutné je ukládat.

Položená otázka tedy zní spíše takto: jak systém pozná co je potřeba ukládat a co ne? Zde existuje několik přístupů, které jsou obvykle používány v OODBMS:

- *Perzistentní třídy.* V okamžiku, kdy se definuje třída objektů, je současně specifikováno, zda má být perzistentní či nikoliv. Je-li třída označena za perzistentní, budou mít všechny objekty od ní instanciované charakter perzistentních dat.

- *Perzistentní stínové třídy* (*shadow classes*). Při vytváření třídy objektů se automaticky vytvoří jedna třída perzistentní a jedna třída tranzientní. Je-li pak objekt instanciován od perzistentní verze, je perzistentní, je-li instanciován od tranzientní verze, je považován za tranzientní.

- *Perzistentní kořenová třída.* V systému existuje jedna předdefinovaná perzistentní třída a vlastnost perzistence se přenáší na další třídy prostřednictvím dědičnosti. Tedy perzistentní třídy jsou všechny třídy, u kterých se někde v hierarchii dědičnosti jako nadřazená třída vyskytuje tato speciální kořenová třída.

- *Perzistence specifikovaná při vytváření objektů.* V tomto případě se nedeklaruje perzistence na úrovni tříd, ale na úrovni jednotlivých objektů. Při vytváření objektu je objekt prohlášen buďto za perzistentní nebo za tranzientní (např. v operátoru *new*).

- *Explicitní ukládání.* Tento mechanismus, blízký tradičním aplikacím pracujícím nad soubory, umožňuje explicitní uložení objektu jednou z jeho metod. Ukládání objektů je tedy řízeno programátorem, který databázovou aplikaci vytváří.

- *Perzistentní kořeny (perzistent roots) předdefinované systémem.* Toto populární schéma perzistence je založeno na myšlence *perzistence dle dosažitelnosti* (persistence by reachability). V systému existují předdefinované objekty (persistent roots), obvykle charakteru kontejneru. Pak všechny objekty, na které existuje odkaz (i nepřímý) z některého perzistentního kořenu, jsou považovány za perzistentní.

- *Pojmenované objekty jako perzistentní kořeny.* Vzniká rozšířením předcházejícího modelu o volitelnost perzistentních kořenů. Za perzistentní kořeny jsou nyní považovány všechny tzv. *pojmenované objekty*; pojmenováním se zde rozumí přiřazení jména konkrétnímu objektu na úrovni schématu databáze. Stejně jako v předchozím případě se pak za perzistentní považují všechny objekty dosažitelné z některého perzistentního kořene.

3.4 Ukládání kódu a dat

Další otázkou perzistence OODBMS je metoda ukládání aplikačního kódu v souvislosti s daty. Zde existují tyto čtyři možnosti:

- data i kód jsou uloženy nezávisle v systému souborů (klasické aplikace);

- data jsou spravována DBMS, kód je uložen a spravován vně DBMS (tradiční databázové systémy);
- data i kód jsou spravována DBMS (ortogonálně perzistentní jazyky);
- DBMS udržuje třídy které obsahují instanciované objekty a kód (varianta nejlépe splňující požadavky kladené na OODBMS).

3.5 Odstraňování dat

Vzhledem k limitované kapacitě perzistentního úložiště je dalším důležitým úkolem OODBMS efektivní odstraňování nepotřebných objektů.

Klasickým přístupem je technika explicitního zrušení objektu programátorem. Nespornou výhodou této metody je explicitní identifikace rušených objektů. Nevýhodou je ovšem přenesení zodpovědnosti na stranu programátora a snadná možnost ztráty konzistence. Konkrétně, je-li rušený objekt vázán určitými vztahy na jiné objekty, je třeba rozhodnout, zda se tyto vztahy zruší, mají-li se odstranit i další objekty vázáné těmito vztahy, nebo je odstranění daného objektu z hlediska sémantiky datového modelu nepřípustné (což například jazyk C++ při rušení objektu nepodporuje).

Druhou variantou odstraňování nepotřebných objektů je technika *garbage collectingu*. V tomto případě OODBMS sám automaticky identifikuje nepotřebné objekty a ruší je. Vyvstává zde však důležitý problém, jak identifikovat nepotřebné objekty:

- *Počítadlo referencí (reference counting).* Každý objekt si pamatuje počet odkazů na sebe sama, klesne-li počet na nulu, objekt se stává nedostupným a tudíž dále nepotřebným. Tato metoda je sice jednoduchá, ale selhává v případě, že se vytvoří izolované smyčky objektů, které jsou sice nedostupné, ale vzájemně na sebe odkazují a tudíž jejich počítadla referencí nikdy neklesnou na nulu. V praxi je tedy nutné tuto metodu doplnit o další techniky rozpoznávání izolovaných smyček.
- Technika *mark-and-sweep*. Funguje tak, že systém identifikuje objekty dosažitelné z perzistentních kořenů. Všechny ostatní objekty jsou pak prohlášeny za nedostupné a označené ke zrušení.

4 Standard ODMG

Podobně jako v případě relačních databází, postupný vývoj objektově orientovaných databází ukázal nutnost standardizace základních konceptů OODBMS. Jedině v případě respektování určitého společného standardu je možné splnit významné požadavky na univerzální databázový systém, jako je jednotný model datové struktury a rozhraní, snadná přenositelnost a interoperabilita napříč různými systémy i aplikacemi. Tato kapitola se bude blíže věnovat konceptům patrně nejvýznamnějšího standardu OODBMS, navrženém skupinou ODMG (*Object Database Management Group*).

4.1 ODMG

Standard ODMG je postaven na objektových standardech skupiny OMG (*Object Management Group*). Tato skupina zahrnuje více než 300 komerčních firem, společností vyvíjejících databázové systémy i jejich uživatelů. OMG standardizuje objektově orientované systémy na obecnější úrovni než jsou databáze, popisuje například obecný typový model, standardní popis rozhraní (*IDL – Interface Description Language*) apod. Produktem OMG jsou například architektury ORB a CORBA.

ODMG sice vychází z OMG, ovšem zaměřuje se speciálně na aspekty související s OODBMS. Skupina byla založena Rickem Cattellem ze Sun Microsystems a v současné době je tvořena odborníky z přibližně deseti hlavních společností vyvíjejících OODBMS systémy. Standard ODMG definuje následující základní aspekty OODBMS:

- obecnou architekturu OODBMS;
- logický datový model společný pro všechny OODBMS, umožňující stejnou úroveň interoperability jako u současných relačních DBMS;
- jazyk pro popis schématu dat, *ODL – Object Data Definition Language*;
- dotazovací jazyk, *OQL – Object Query Language*;
- návrh včlenění podpory ODMG do řady existujících objektově orientovaných programovacích jazyků. (*DBPL – Database Programming Languages*)

4.2 Datový model ODMG

Logický objektový model ODMG tvoří jádro celého standardu. Je postaven na obecných principech objektově orientovaných systémů, jak byly prezentovány v kapitole 2. Proto zde bude uveden pouze základní přehled a vyzdviženy případné rozdíly.

Datový model ODMG definuje:

- *Objekty a literály.* Jak objekty, tak i literály byly podrobně popsány v kapitole 2.
- *Atributy a vztahy (relationships).* Zatímco atributem se rozumí vlastnost či údaj příslušející k jednomu objektu, vztah má obousměrný charakter vazby mezi dvěma objekty. Příklad: mějme objekt třídy `Student`. Pak rodné číslo je atributem daného objektu, avšak zápis studenta do určitého kurzu je obousměrný vztah (student ví, který kurz má zapsaný, a naopak objekt reprezentující kurz má vazbu na všechny zapsané studenty). Vztahy jsou buďto typu 1:1, 1:N, N:1 nebo M:N.
- *Objektové typy.* Pod pojmem objektový typ definuje ODMG to, co se obvykle označuje termínem *třída*. Kromě základních vlastností třídy, jako jsou *jméno třídy*, množina *bázových tříd* (supertypů), *atributy* a *operace* (metody), doplňuje ODMG definici třídy o výše popsané *vztahy* (relationships), *klíčové položky* (keys) a tzv. *extent*. Extentem se nazývá automaticky spravovaná kolekce všech objektů dané třídy – logický ekvivalent tabulky záznamů z relačních databází. Klíče jsou pak datové položky, které jsou unikátní pro každý objekt dané třídy. Dalším rozšířením je podpora *výjimek*, což jsou speciální události, vyvolané při určité (typicky chybové) situaci.

Výjimka může nést data, blíže popisující vzniklou chybu; o zpracování výjimky se starají k tomu určené metody (exception handlers).

- *Předdefinované typy.* ODMG definuje řadu zabudovaných atomických i strukturovaných datových typů. Standard současně zavádí i bohatou množinu kolekcí polymorfního charakteru.

4.3 Jazyk ODL

Jazyk ODL slouží k definici logického schématu objektově orientované databázové aplikace.

4.3.1 Zabudované datové typy

OMDG/ODL definuje tyto atomické typy:

- celá čísla: `long`, `short`, `unsigned long`, `unsigned short` a `octet` (1 byte);
- reálná čísla: `float` a `double`;
- další typy: `boolean` a `char` (znak);
- speciální typ `any`: označuje datovou entitu libovolného typu.

Objektový model dále definuje strukturované datové typy množina (`Set`), multimnožina (`Bag`), pole (`Array`) a seznam (`List`). Od každého typu existují dvě varianty, jedna proměnlivá (mutable) a druhá neproměnlivá (immutable, literál). Kolekce (množina, multimnožina, pole a seznam) jsou navržené jako generické, tj. lze je používat podobně jako šablony v C++. Zvláštním případem seznamu jsou typy `String` a `Bit_String`.

4.3.2 Uživatelsky definované struktury

Struktury odpovídají klasickému datovému typu `struct` v jazyce C, tj. jedná se o uživatelsky definované strukturované typy obsahující předdefinovaný pevný počet pojmenovaných složek. Příklad definující typ adresy:

```
struct Address
{
       Unsigned Short House;
       String Street;
       String City;
       String Postcode;
};
```

4.3.3 Uživatelsky definované objektové typy (třídy)

Definice třídy má syntaxi

```
interface <jméno třídy> [: <seznam bázových tříd>]
        (<extenty a klíče>) [: <indikátor perzistence>]
        {<seznam veřejných položek>}
```

V seznamu extentů a klíčů se specifikuje extent a klíčové položky třídy. Definice extentu má tvar

```
extent <jméno extentu>
```

a definice klíčů

```
key <klíč>
```

nebo

```
keys <klíč_1>, ..., <klíč_n>
```

Jednoduchý klíč se zapisuje jako název klíčového atributu, složený klíč se zapisuje jako seznam atributů v kulatých závorkách. Indikátor perzistence je jedno z klíčových slov `persistent` nebo `transient`.

Existuje celkem šest druhů veřejných položek, které se mohou objevit v těle definice:

- *Deklarace typu.* Definuje typové synonymum (pojmenování typu) a má syntaxi

  ```
  typedef <typový výraz> <jméno typu>
  ```

- *Definice konstanty* má syntaxi

  ```
  const <jméno typu> <jméno konstanty> = <výraz>
  ```

- *Definice výjimky.* Syntaxe výjimky je

  ```
  exception <jméno výjimky> [ <seznam elementů> ]
  ```

  kde seznam elementů popisuje data, nesená výjimkou; každý element je tvaru

  ```
  <datový typ> <jméno>
  ```

- *Deklarace atributu* má syntaxi

  ```
  attribute <specifikace typu> <název atributu>
  ```

- *Deklarace vztahu* má syntaxi

```
relationship <typ objektu> <název vztahu>
[ inverse <inverzní typ objektu>::<název inverzního vztahu> ]
```

Jak již bylo řečeno, vztah má obousměrný charakter, proto část uvozená klíčovým slovem `inverse` definuje inverzní část vztahu mezi objekty. Typ vztahu (1:1, 1:N, N:1, nebo M:N) je určen typy objektů které tvoří vztah. Principu definice vztahu lze nejlépe porozumět na příkladu (částečně převzato z [2]):

```
interface Person
(extent People) persistent
{
       relationship Set<Adult> parents inverse Adult::children;
       relationship Company worksFor inverse Company::employees;
}
interface Company
(extent Companies) persistent
{
       relationship Set<Person> employees inverse Person::worksFor;
}
interface Adult : Person
{
       relationship List<Person> children inverse Person::parents;
}
```

Ve třídě `Person` jsou definovány dva vztahy. Vztah `parents` definuje rodiče osoby a jeho inverzní částí je vztah `children` ve třídě `Adult`. Protože na straně třídy `Person` je typem vztahu množina `Set<Adult>` a na straně třídy `Adult` seznam typu `List<Person>`, vztah je typu M:N. Vztah `worksFor` určuje objekt typu `Company`, tj. společnost pro kterou osoba pracuje. Inverzní částí je vztah `employees` ve třídě `Company`. Protože na straně třídy `Person` je vztah typu `Company`, tedy odkazuje na jediný objekt, a na straně třídy `Company` je typem vztahu množina `Set<Person>`, vztah je typu 1:N.

- *Rozhraní operace* je posledním druhem veřejné položky vyskytující se v definici třídy. Syntaxe operace je:

```
<typ návratové hodnoty> <název operace> ([<seznam parametrů>])
[ raises <seznam názvů výjimek> ]
```

Každý parametr v seznamu parametrů má pak tvar

```
<směr> <typ parametru> <název parametru>
```

Směr parametru určuje, zda je parametr pouze vstupní (`in`), pouze výstupní (`out`), nebo vstupně-výstupní (`inout`).

4.3.4 Moduly

Moduly slouží k zapouzdření logicky vzájemně souvisejících tříd a jiných definic. Syntaxe modulu je

```
module <název modulu> { <seznam definic> }
```

V těle modulu lze uvádět typové deklarace, definice konstant, definice výjimek a deklarace rozhraní tříd. Navíc, kromě těchto již dříve popsaných elementů jazyka ODL, může modul obsahovat i deklarace pojmenovaných perzistentních kořenů. Jejich syntaxe je

```
name <typ objektu> <jméno perzistentního kořene>
```

4.4 Dotazovací jazyk OQL

Nezbytnou součástí kvalitního OODBMS (a obecně jakéhokoliv DBMS) je podpora dotazovacích jazyků. Požadavky kladené na dotazovací jazyk lze shrnout do následujících bodů:

- snadná použitelnost
- popis dat na vysoké úrovni
- deklarativní jazyk
- efektivní provádění dotazů
- nezávislost na aplikaci, přenositelnost.

Je otázkou, zda má dotazovací jazyk současně plnit úlohu jazyka manipulačního, tedy umožňovat přidávání, úpravy a odstraňování údajů z databáze.

V oblasti relačních databází pokrývá výše uvedené požadavky nejrozšířenější dotazovací jazyk SQL. Ten v sobě zahrnuje jak jazyk dotazovací, tak i manipulační, protože obsahje příkazy pro vkládání, změny i rušení záznamů v databázi. Oproti relačním databázím, kde tabulkový model umožňuje snadný a velmi efektivní přístup k datům, však implementace dotazovacího jazyka v objektově orientovaných databázích naráží na několik problémů.

Prvním problémem, který už byl několikrát zmíněn v jiných souvislostech, je komplikovaná datová struktura objektových databází, založená na množství objektů svázaných řadou vztahů. Důsledkem je, ve srovnání s klasickými DBMS, vyšší složitost dotazovacích algoritmů a především jejich nižší efektivita.

Druhý typický problém vyplývá z filozofie objektově orientovaného modelování. Zatímco v relačních databázových systémech není přístup k datům v tabulkách nijak omezen, základním principem objektového návrhu je naopak maximální zapouzdření a ukrývání datových položek v objektech. Otázkou tedy je, jakým způsobem efektivně zpřístupnit data dotazovacímu procesoru. Řešením můžou být zvláštní přístupová práva pro dotazovací procesor. Pokud však procesor zároveň umožňuje manipulaci s daty, tento přístup potenciálně vede k vnesení nekonzistencí a narušení integrity dat. Pokud však dotazovací jazyk umožňuje pouze výběr dat a nikoliv jejich úpravy, nic nebrání tomu, aby měl dotazovací procesor volný přístup ke všem údajům v objektech.

Důsledkem výše uvedených úvah je závěr, že v OODBMS je vhodné dotazovací jazyk omezit pouze na vyhledávání a výběr dat.

V rámci standardu ODMG byl navržen dotazovací jazyk OQL. Jedná se o čistě dotazovací jazyk, tedy OQL kromě vyhledávání a výběru neumožňuje žádné manipulace s objekty v databázi. Centrálním příkazem jazyka je příkaz `select`, který je zobecněnou variantou `select`u z jazyka SQL. Jazyk podporuje kvantifikátory, agregační funkce, konverzní funkce, pojmenované dotazy, množinové operace a další. Součástí jazyka jsou i syntaktické konstrukce umožňující definice nových objektů a literálů, avšak tyto vznikají pouze jako tranzientní objekty podílející se na vzniku návratové hodnoty výběrového příkazu, a tedy nejsou vkládány do databáze. OQL je silně typovaný jazyk, výsledkem každé operace je objekt některé třídy, typicky kolekce objektů.

Následuje přehled základních jazykových konstrukcí OQL. Kompletní referenci lze najít například v [1].

4.4.1 Atomické literály, pojmenované objekty

Každý literál a pojmenovaný objekt je výrazem jazyka OQL. Příklady: `true` a `false` jsou literály typu `Boolean`, `27` je literál typu `Integer`, `nil` je literál třídy `Object`.

4.4.2 Konstrukce objektů

Příkazy

```
set(e1, e2, ..., en)
bag(e1, e2, ..., en)
list(e1, e2, ..., en)
array(e1, e2, ..., en)
```

vytváří množinu, multimnožinu, seznam nebo pole obsahující objekty e_1 až e_n. K vytvoření objektu libovolného typu slouží konstrukce

```
<třída objektu> (p1: e1, ..., pn: en)
```

která vytváří objekt požadované třídy a každému atributu p_i přiřadí hodnotu e_i.
Příklad:

```
Employee (name: "Peter", boss: Chairman)
```

definuje objekt třídy `Employee`.

Pokud se místo třídy objektu uvede klíčové slovo `struct`, konstrukce vytváří nepojmenovanou strukturu s danými atributy.

4.4.3 Výrazy nad atomickými typy

OQL podporuje tvorbu výrazů s logickými operátory `not`, `and`, a `or`, kde operandy jsou výrazy typu `boolean`. Dále podporuje aritmetické operátory `+`, `-`, `*`, `/`, `mod` (modulo), relační operátory `<`, `>`, `<=`, `>=`, `=`, `!=`, a operátor konkatenace řetězců `||`. Dalšími operátory jsou `in` (testuje přítomnost znaku v řetězci) a `like` (testuje zda řetězec odpovídá danému vzoru). Operace `s[i]` vrací i-tý znak řetězce `s`, operace `s[l:h]` vrací podřetězec od pozice l po pozici h.

4.4.4 Operace nad objekty

Operace `student.name` vrací hodnotu atributu `name` v objektu `student`. Operátor `->` umožňuje volat metodu objektu, výraz pak nabývá hodnoty vrácené volanou metodou nebo `nil`. Příklad:

```
student->register_course ("ZAP", 2003)
```

4.4.5 Kvantifikátory

Operátory `for all` a `exists` představují univerzální a existenční kvantifikátor. Jako výrazy nabývají hodnoty typu `boolean`.
Příklady:

```
for all x in Students: x.year > 2001
exists x in Students: x.name = "John"
```

4.4.6 Operace nad kolekcemi

Predikáty `exits(e)` a `unique(e)` zjišťují, zda je v kolekci `e` alespoň jeden respektive právě jeden prvek. Operátor `e in col` detekuje, zda se objekt `e` vyskytuje jako prvek kolekce `col`. Pro kolekce s definovaným poředím prvků vrací operace `col[i]` i-tý prvek kolekce, `col[i:j]` vrací podkolekci od i-tého po j-tý prvek, operace `first(col)` a last `last(col)` vrací první a poslední prvek kolekce, operátor `+` spojuje dvě kolekce. Operace `listtoset` konvertuje seznam na množinu. Operace `distinct` odstraňuje duplikáty v kolekci.

4.4.7 Množinové operace

Binární operátory `union`, `intersect` a `except` implementují množinové sjednocení, průnik a rozdíl nad kolekcemi. Operátory `<` a `<=` aplikované na kolekce detekují, zda je jedna kolekce vlastní podmnožinou respektive podmnožinou druhé kolekce.

4.4.8 Agregační funkce

OQL definuje agregační funkce `min(col)`, `max(col)`, `count(col)`, `sum(col)` a `avg(col)`. Tyto funkce vrací minimum, maximum, počet prvků, součet prvků a průměr prvků kolekce `col`.

4.4.9 Příkaz Select-From-Where

Obecná syntaxe výběrového příkazu select-from-where je

```
select [distinct] <projekce>
       from <seznam kolekcí s iterátory>
       [ where <filtrační výraz> ]
```

Seznam kolekcí s iterátory má tvar

```
col_1 [as] x_1, ..., col_n [as] x_n
```

kde `col`$_i$ jsou kolekce a `x`$_i$ jsou iterační proměnné probíhající prvky kolekcí. Nepovinná klauzule `where` obsahuje libovolný výraz nabývající hodnoty typu `boolean`.

Příkaz provede kartézský součin předložených kolekcí[1], na výsledek aplikuje filtrační výraz z klauzuje `where` a poté provede požadovanou projekci. Výsledkem příkazu je objekt typu `set` v případě `select distinct` nebo objekt typu `bag` v případě `select`.

Projekce v příkazu select-from-where je libovolný výraz konstruující objekt, obecně tvaru

```
<typ objektu> (p1: e1, ..., pn: en)
```

(viz odstavec Konstrukce objektů). Druhou možností je výraz tvaru

```
[p1:] e1, ..., [pn:] en
```

který konstruuje objekt třídy `struct`. Třetí možností je

```
*
```

V tomto případě vrací projekce objekty ve tvaru, jak byly obdrženy po provedení klauzule `where`. Výrazy `e`$_1$ až `e`$_n$ jsou libovolné výrazy, jejichž operandy mohou obsahovat iterační proměnné `x`$_i$ nebo názvy kolekcí vyskytující se v klauzuli `from`.

Příklad (převzato z [1]):

```
select student: Students.name, professor: teachers.name)
from Students,
     Students.courses as courses
     courses.taught_by teachers
where teachers.rank = "full professor"
```

Dotaz vrací multimnožinu s prvky typu `struct(name: string, professor: string)`.

4.4.10 Operátor Group-by

Operátor group-by má syntaxi

```
<select-from-where příkaz> group by <slučovací atributy>
        [having <predikát>]
```

Seznam slučovacích atributů má tvar

```
attr1: e1, ..., attrn: en
```

Operátor prochází kolekci po provedení klauzule `where` a seskupuje objekty, pro které všechny výrazy `e`$_i$ vyjmenované v seznamu slučovacích atributů nabývají stejných hodnot. Výsledek seskupování je typu

```
set <struct(attr1, ..., attrn, partition: bag <type_of(<seskupené zaznámy>)> >
```

[1] V případě, že kolekce nejsou stejného typu, tj. některé jsou množiny, jiné multimnožiny, pole, nebo seznamy, předchází kartézskému součinu transformace kolekcí na stejný typ (detaily viz [1]).

kde `partition` je multimnožina všech záznamů které byly zařazeny do této skupiny (přesná definice typu atributu `partition` viz např. [1]).

Po seskupení je na množinu skupin aplikován filtrační predikát z klauzule `having` a na závěr je na výsledek aplikována projekce z klauzule `select`. To znamená, že projekce se musí odkazovat na objekty, jejichž typ je daný seskupovací operací.
Příklad ([1]):

```
select *
from Employees e
group by
       low: e.salary < 1000,
       medium: e.salary >= 1000 and e.salary < 10000,
       high: e.salary >= 10000
```

Tento dotaz vrací množinu typu

```
set <struct(low: boolean, medium: boolean, high: boolean
partition: bag <struct(e:Employee)>)>
```

se třemi objekty pro tři různé úrovně výše platu, jejichž atributy `partition` obsahují zaměstnance spadající do dané kategorie.

4.4.11 Operátor Order-by

Operace order-by má syntaxi

<*výběrový příkaz*> `order by` e_1 `[asc|desc], ...,` e_n `[asc|desc]`

kde výběrový příkaz je buďto select-from-where nebo select-from-where-groupby a e_i jsou výrazy. Výsledkem operace je objekt typu `list` obsahující seřazené objekty.

4.4.12 Pojmenované dotazy

Příkaz `define` umožňuje definovat tzv. pojmenované dotazy. Příklad:

```
define Johns as select x from Students x
       where x.name = "John"
```

tvoří pojmenovaný dotaz `Johns`, který vrací všechny studenty se jménem John.

5 C++ jako databázový programovací jazyk

V relačních databázích se obvykle veškerá manipulace s databází provádí prostřednictvím příkazů jazyka SQL, a to i v případech, kdy je s daty manipulováno z jichých programovacích jazyků (Embedded SQL). Tento přístup ovšem není v případě objektových databází jednoduše realizovatelný, především z důvodů složitosti objektového modelu a nehomogenity dat v databázi (data roztýlena po objektech řady tříd, množství obousměrných vztahů atd.) Objektové databáze tedy volí jiný přístup, kterým je integrace databázové podpory do existujících objektově orientovaných programovacích jazyků, jako jsou C++

nebo Smalltalk. V této kapitole bude přiblíženo toto rozšíření jazyka C++ podle standardu navrženého skupinou ODMG.

Jazyk C++ je v současnosti nejrozšířenější objektově orientovaný programovací jazyk. Jeho charakteristickým rysem je velká flexibilita, umožňující programátorům nejrůznější přístupy k objektovému návrhu. Jazyk C++ podporuje vícenásobnou dědičnost, přetěžování metod i operátorů, „friend“ funkce a třídy, virtuální metody, virtuální bázové třídy, statické funkce a atributy, komplexní podporu pro generické programování (šablony), mechanismus výjimek a další. Jazyk C++ má ovšem pro použití v objektových databázích několik nevýhod. Těmi jsou především málo bezpečný typový systém, operace s ukazateli (*pointer arithmetic*), explicitní destrukce objektů (absence garbage collectoru) nebo možnost „friend“ tříd a operací. Uvedená řada nedostatků je většinou přímo či nepřímo důsledkem nízké úrovně jazyka C, nad kterým byl jazyk C++ postaven. V databázových aplikacích mají tyto nedostatky za následek menší bezpečnost a riziko ztráty konzistence.

5.1 Podpora OODBMS v C++

Aby mohlo C++ sloužit jako databázový programovací jazyk, je nutné jej doplnit o

- způsoby odkazování do databází a navázání objektů na perzistentní úložiště;
- indikaci perzistence;
- popis schémat databází;
- podporu transakcí a dotazování;
- podporu automaticky spravovaných obousměrných vztahů;
- podporu pojmenovaných objektů.

Rozšíření je možné provést v zásadě dvěma způsoby. Buďto se využije stávajících prostředků jazyka C++, tj. vytvoří se knihovny nových tříd a funkcí implementujících přístup k databázím, nebo se jazyk rozšíří o nové syntaktické konstrukce. První přístup je snadněji implementovatelný a lépe přenositelný, v druhém případě je výhodou přímočarý, jednoduchý a nenásilný popis specifických konstrukcí ve zdrojových kódech aplikace. Jak uvidíme dále, návrh ODMG pro jazyk C++ používá obě techniky.

5.2 Reprezentace databáze a schémat

Databázi je v C++ jednoznačně výhodnější implementovat s využitím stávajících prostředků, například jako objekt třídy `Database`.

Schémata databází by bylo možné reprezentovat pouhou definicí tříd na úrovni hlavičkových souborů. Z důvodů tradice udržování metadat přímo v databázi je ovšem vhodnější pro popis schémat implementovat knihovnu tříd. Standard ODMG v souvislosti se schématy v jazykce C++ nedává žádná doporučení.

5.3 Indikace perzistence

Pro indikaci perzistence jsou doporučovány čtyři možné přístupy:

- perzistentní forma operátoru `new`;
- instanciace od nějaké vyznačené bázové třídy;
- indikace v rámci popisu třídy;
- ortogonálně podle dosažitelnosti z perzistentních kořenů.

Perzistentní forma operátoru `new` se vytvoří jeho přetížením, kdy dalším parametrem je objekt databáze, v jejímž perzistentním úložišti má být objekt uložen. Druhým používaným přetížením je operátor `new` jehož parametr je odkaz na existující objekt. Vzniká tak perzistentní objekt, který je v rámci clusterování datového úložiště vložen do stejného clusteru jako daný objekt.

Zbývající tři techniky jsou opět snadno realizovatelné s původními prostředky jazyka C++.

5.4 Reference

Jazyk C++ umí pracovat s odkazy typu pointer (*) nebo reference (&). Tyto však nejsou použitelné v perzistentním prostředí, proto ODMG zavádí šablonu reference `Ref<T>`, která má chování klasické reference a současně splňuje požadavky na perzistenci. Kromě šablony pro reference na objekty dané třídy `T` existují ještě reference třídy `Ref_Any`, které můžou odkazovat na objekt libovolného typu.

5.5 Transakce a zámky

Přestože transakce a zámky by bylo možné realizovat rozšířením syntaxe jazyka, syntaktická úspora zde není nijak významná a proto ODMG doporučuje oboje implementovat jako objekty, např. tříd `Transaction` a `Lock`.

5.6 Dotazy

Doporučeným přístupem v případě dotazů je technika známá z relačních databází, tzv. *embedded query language*. Implementována může být buďto jako funkce třídy `Database` nebo jako nová třída. V obou případech je parametrem řetězec s dotazem v dotazovacím jazyce OQL, který je zpracováván dotazovacím procesorem v rámci systému řízení báze dat.

5.7 Obousměrné vztahy

Vztahy jsou prvním rozšířením, u kterého ODMG doporučuje syntaktické rozšíření jazyka. Vrátíme-li se k příkladu z ODL, tak vztah `worksFor` ve třídě `Person` se v C++ zapíše jako

```
class Person
{
        Ref<Company> worksFor inverse Company::employees;
}
```

5.8 Pojmenované objekty, extenty a klíče

Pojmenované objekty doporučuje ODMG implementovat s využitím původního C++, tedy v rámci třídy `Database`, zpřístupněné prostřednictvím patřičných metod. Extenty je možné do C++ doplnit buďto pomocí rozšíření syntaxe jazyka, explicitně v rozhraní třídy nebo automaticky transparentně u všech tříd. V případě klíčů ODMG nedává žádná doporučení.

6 O_2

O_2 reprezentuje významný objektově orientovaný databázový systém postavený na standardech ODMG. Jeho vývoj se datuje od roku 1987 (Altair Research Consortium), od verze 4.6 splňuje standard ODMG.

Architektura systému O_2 má hierarchickou strukturu. Na nejnižší úrovni je správce datového úložiště *O_2Store*. Nad úložištěm leží úroveň *O_2Engine*, která má dvě podúrovně. Nižší z nich je *Object Manager*, který vytváří logický objektový model dat nad úložištěm. Nad ním leží *Schema Manager*, spravující schémata a databáze. Nad jádrem O_2Engine existuje v systému O_2 řada dalších podpůrných prostředků, jako jsou knihovny, nástroje, grafická rozhraní, databázové programovací jazyky a další. Mezi nejdůležitější standardní knihovny a nástroje patří

- O_2Kit, obsahující standardní třídy a objekty uživatelského rozhraní;
- O_2Graph, editor grafických výstupů;
- O_2Version, software pro správu verzí;
- O_2Meta_schema, knihovna pro manipulaci s metadaty;
- O_2Look, nástroj pro automatické generování formulářů pro manipulaci s objekty;
- O_2Tools, uživatelské rozhraní na bázi formulářů, nadstavba O_2Look;
- O_2Web, aplikace pro podporu WWW;
- O_2DBAccess, rozhraní pro přístup do relačních databází.

Podporovanými programovacími jazyky jsou C++, Smalltalk a O_2C (O_2 nadstavba jazyka C), jako dotazovací jazyk se používá OQL.

6.1 O_2Store

O_2Store implementuje správu datového úložiště v O_2. Data v úložišti nemají objektový charakter, jsou strukturována jako množiny záznamů. Důvodem je vyšší efektivita. Manipulace s daty probíhá na úrovni stránek, za výběr stránek je zodpovědná vyšší úroveň. Celé úložiště je tedy navrženo tak, že je nezávislé na datovém modelu nad ním.

Základní jednotkou úložiště jsou *svazky* (*volumes*). Svazkem může být disková partition, jeden unixový soubor, skupina souborů apod. Svazky obsahují:

- sekvenční soubory s proměnnou délkou pro numerická a textová data;
- B-stromy a další typy indexů pro rychlý přístup k datům;
- speciální struktury pro objemná data až do 4 GB (obrázky, multimedia, apod.);
- speciální, tzv. *scan* struktury používané pro správu přístupu k datům.

Úložiště podporuje dvoufázové zamykání, propracovaný systém cachování a write-ahread žurnálový systém zotavení.

6.2 Object Manager

Primární úlohou Object Manageru je mapování logického objektového modelu dat na systém stránek a záznamů, používaný úložištěm O_2Store. Object Manager spravuje objekty, reference, vztahy a indexy. Model perzistence poskytovaný Object Managerem je založený na perzistentních kořenech. Součástí je i inkrementální garbage collector paměti, spouštěný explicitně.

6.3 Schema Manager

Schema Manager spravuje schémata databází, tj. třídy, datové typy, operace, funkce, aplikace, pojmenované objekty a další. Podporuje inkrementální změny schématu (kdykoliv za chodu). Při změně schématu se uplatňuje tzv. *lazy* přístup ke změnám v objektech; po změně schématu zůstávají objekty v původním tvaru a transformace jednotlivých objektů na nové schéma proběhne až v okamžiku použití objektu. Další vlastností je možnost použití dosud nedefinovaných entit (nedefinovaných tříd, typů apod.). Spuštění kódu nad nekompletním schématem je odmítnuto a jsou zobrazeny entity, které je potřeba dodefinovat.

6.4 Databázová struktura, aplikace

Nejvyšší logickou jednotkou v O_2 je *schéma*. Schéma obsahuje definice tříd, literálů, funkcí a pojmenovaných objektů. Každá databáze je *instancí některého schématu*. Současně se schémata používají i pro zapouzdření jednotlivých knihoven systému, z kterých je pak možné importovat funkce a data.

Další logickou jednotkou je *aplikace*, zapouzdřující programy a transakce. Součástí aplikace je několik předdefinovaných programů, spouštěných při startu aplikace, restartu aplikace, ukončení aplikace a po ukončení každého programu.

6.5 Podpora verzí

O_2 umožňuje udržovat systém různých verzí tříd a objektů. Ty se odvozují od třídy *O2_Version*. Systém eviduje graf odvozování verzí, pojmenovávání verzí, transformace mezi verzemi, slučování a další.

7 Závěr

Přestože relační databázové systémy jsou v současné době nejpoužívanějším prostředkem pro správu a manipulaci s rozsáhlými daty, jejich koncepce postavená na jednoduchém tabulkovém modelu dat se ukázala nepříliš vhodnou pro řadu aplikací, pracujících se složitě strukturovanými daty, jako jsou například geografické systémy nebo aplikace z oblasti průmyslového designu. Důsledkem byl vznik nové třídy databázových systémů postavených na objektovém modelu, schopném lépe reprezentovat popisované entity z reálného světa. Objektově orientované databázové systémy (OODBMS) kombinují objektový model dat s technikami perzistence, transakčního přístupu, možnostmi vyhledávání na bázi dotazovacích jazyků a řadou dalších vlastností. Výsledkem je univerzální databázový systém, spravující bázi dat ve formě vzájemně provázaných objektů řady různých tříd.

Podobně jako v případě relačních databází, i v oblasti objektově orientovaných databází vznikla nutnost standardizovaného logického modelu dat a rozhraní. Skupina *Object Database Management Group* (ODMG) vytvořila standard pro obecnou strukturu OODBMS, zahrnující základní objektový datový model, obecný popis schémat databází prostřednictvím jazyka ODL, dotazovací jazyk OQL a doporučená rozšíření objektově orientovaných programovacích jazyků o podporu objektových databází. Tento standard umožňuje sjednotit různé varianty objektově orientovaného přístupu v oblasti databází a tím zajistit jednoduchou přenositelnost a interoperabilitu mezi OODBMS, podobně jako je tomu u relačních databází.

Ačkoliv objektově orientované databáze přináší řadu prvků umožňujících mnohem intuitivnější návrh datových struktur aplikací, nelze očekávat, že v nejbližší době zcela nahradí relační databázové systémy. V široké oblasti aplikací je použití komplexního objektového modelu dat zbytečné a tyto aplikace naopak těží z jednoduchosti a vysoké efektivity relačních databází. Mimo tuto oblast však existuje množství aplikací, v kterých je zavedení OODBMS velkým přínosem. Je proto pravděpodobné, že s dalším vývojem OODBMS se bude oblast použití objektově orientovaných databází dále rozšiřovat.

Reference

[1] ODMG OQL User Manual, O_2 Documentation Set, 1998.

[2] R. Cooper, *Object Databases: An ODMG Approach*, Intl. Thomson Computer Press, 1997.