POET 3.0
Systém pro práci s persistentními objekty

Jana Kuklová, Luboš Popelínský
Fakulta informatiky
Masarykova Universita Brno
{jana, popel}@fi.muni.cz

Abstrakt

Ve zprávě jsou popsány základní charakteristiky objektově orientovaného databázového systému POET.

Systém byl vyvinut organizací POET Software, která je členem skupiny ODMG (Object Database Management Group), zabývající se vývojem standardu v oblasti objektově orientovaných databázových systémů.

Obsah

1. Úvod **3**

2. Datový model **3**

3. Programování databázových aplikací v jazyce C++ **5**

3.1. Funkce preprocesoru PTXX *5*

3.2. AllSet *6*

3.3. Query *6*

3.4. Množiny *7*

3.5. Inicializace, otevření a uzavření databáze *8*

3.6. Metoda STORE() *8*

3.7. Metoda DELETE() *9*

3.8. Verze tříd *9*

3.9. *Indexy* *10*

3.10. Transakční zpracování *10*

3.11. Zamykání objektů *10*

3.12. Funkce vyvolané událostmi *11*

3.13. Přístupová práva *11*

4. Dotazovací jazyk OQL **11**

5. Závěr **12**

6. Literatura **12**

1. Úvod

V technologii objektově orientovaných databází [1, 2, 6] lze sledovat tři hlavní proudy. Prvním z nich je rozšiřování vlastností relačních databází o objektově orientované rysy, druhým rozšíření objektově orientovaného programovacího jazyka o možnosti persistentního ukládání objektů [2, 5]. Třetím směrem je vytvoření nového čistě objektově orientovaného databázového systému. Systém POET [7] vznikl jako persistentní rozšíření jazyka C++.

Objektově orientovaný databázový systém POET lze provozovat na různých operačních systémech - MS Windows 3.1/NT, SCO Unix, Solaris, SGI IRIX, IBM AIX, Linux, Mac a NeXTSTEP. Pro operační systém Windows 3.1 je potřeba předem instalovat 32-bitové rozšíření Win32s. Pro vývoj aplikací je navíc nutno mít k dispoozici C++ překladač a to Borland C++ verze 4.5 nebo MS Visual C++ verze 2.0. Pro práci s vývojovým prostředím je minimální požadavek 8 MB RAM, pro spuštění hotové aplikace 4 MB RAM. Pro vlastní instalaci DB systému POET je potřeba 7 MB HD.
Databázový systém lze provozovat v jednouživatelském režimu, to znamená každá databáze je přístupná nejvýše jednomu uživateli. Ve víceuživatelském režimu jednotlivý uživatelé sdílejí databázi prostřednictvím DB serveru.
Součástí instalace je grafické vývojové prostředí, grafické prostředí pro správu databáze, preprocesor do jazyka C++ a další knihovny nutné pro provoz aplikací.

2. Datový model

POET plně podporuje objekty podle syntaxe a sémantiky jazyka C++. Znamená to, že je podporováno zapouzdření, dědičnost, polymorfismus, uživatelsky definované typy, identita objektů. Navíc jsou definovány následující vlastnosti objektů.

1. Třída objektů může být deklarována jako persistentní (klíčové slovo *persistent*). Tato deklarace umožňuje permanentní uložení všech instancí této třídy do databáze (metoda *Store()*).
Mějme třídu *Person* definovanou následovně:

```
persistent class Person {
 private:
        char        name[30];
        short       age;
        Address     address;
public:
        ...
};
```

2. Na instanci třídy lze vytvářet více odkazů, odkazy, které jsou součástí persistentní třídy jsou uloženy v databázi. Objekt, který je vícenásobně referencován je uložen v databázi pouze jednou. Při čtení objektu z databáze je do paměti přečten požadovaný objekt a všechny objekty, na které se odkazuje.

```
Adam.father = 0;
Cain.father = &Adam;
Abel.father = &Adam;
Cain.Store();                    /* Je uložen také Adam!
Abel.Store();                    /* Adam již byl uložen.
```

3. Odkaz typu *OnDemand* se nenačítá do paměti implicitně, je nutno použít metodu *Get()* pro explicitní čtení referencovaného objektu.

```
persistent class Person
{
public:
        Person*     Spouse;
        ondemand<Person>          Child;
};

Person * pChild;
Father->Child.Get(pChild);
```

4. Odkaz typu *Depend* určuje závislost referencovaného objektu. Referencovaný objekt existuje v databázi jen pokud existuje objekt, který se na něj odkazuje.

Příklad:

```
persistent  class  Person {

   depend  Person  *alter_ego

}

pPerson->Assign(pObjBase);
```

V tomto příkladě *alter_ego* z třídy *person* nebude více existovat, bude-li smazán *person*. Je-li Superman zabit, POET dává jistotu, že zmizí také i Clark Kent.

5. Persistentní třída může obsahovat atributy, které nejsou persistentní (uvozeny klíčovým slovem *transient*) a neukládají se proto do databáze.
Máte-li např. systém oken a používáte okno k zobrazení objektu, pak asi nebudete chtít ukládat strukturu okna do vaší databáze. Samozřejmým řešením tohoto problému je deklarování členu třídy jako *transient*, což pro něj znamená, že nebude uložen , je-li ukládán objekt.

```
persistent  class  Person {
private:
        char          name[30];
        Address       address;

        transient WINDOW *          viewer;
};
```

Je-li tento objekt načítán z databáze, atribut *viewer* je nedefinován. Protože by tohle mohlo způsobit nepředpokládané výsledky, je nutno inicializovat *transient* - členy použitím metody *PtObject::Activate()*.
Tato metoda je POETem dána každé třídě jako virtuální a je volána, kdykoliv je objekt čten z databáze do paměti.

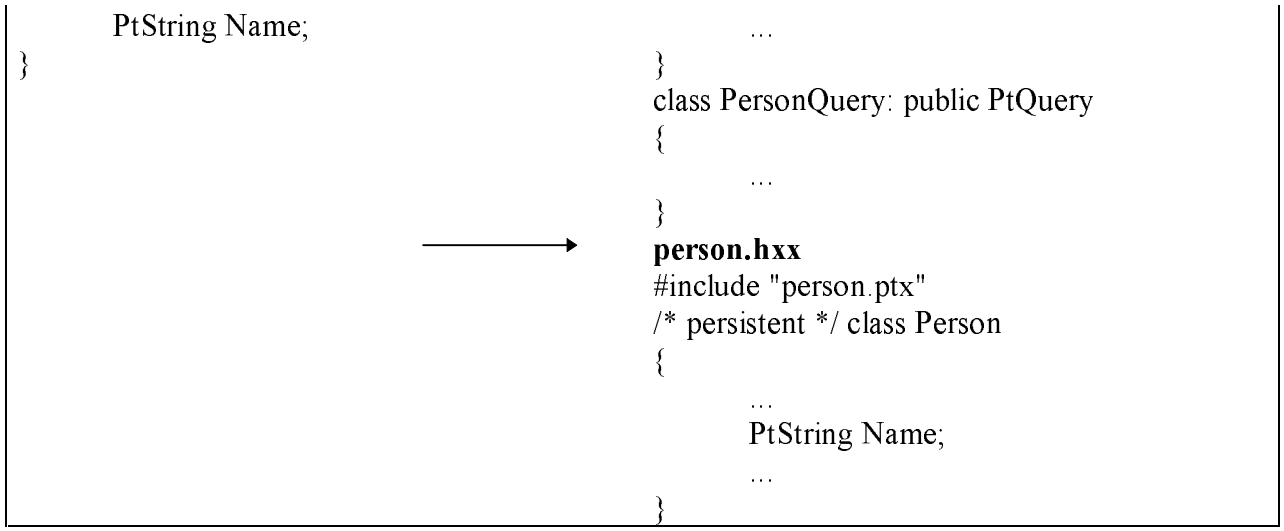
```
persistent  class  Person {
 private:
        PString       name;
         transient  WINDOW*          viewer;
 public:
        virtual  void                Activate() {viewer = new  WINDOW}
};
```

3. Programování databázových aplikací v jazyce C++

3.1. Funkce preprocesoru PTXX

Preprocesor PTXX pro jazyk C++ zpracuje deklarace tříd uložené v souboru .HCD vytvoří *Class Dictionary* databáze, hlavičkové soubory pro programy v jazyce C++ a zdrojový text v C++.
Class dictionary je generovaná preprocesorem PTXX standardně do stejného adresáře jako vlastní databázový soubor. Class dictionary obsahuje popis schématu jedné nebo více databází. Jsou zde popsány všechny (persistentní i nepersistentní) třídy databáze, atributy těchto tříd a definované indexy. V současné verzi POETu není zahrnut popis funkcí tříd a informace o módu public, protected nebo private. POET nabízí programátorům metody pro přímý vstup do class dictionary.
Dalším generovaným souborem je *Class Factory* (.CXX). Tento soubor obsahuje informace o způsobu práce s objekty v databázi, konstruktory tříd, verze tříd a musí být kompilován a linkován s vlastní aplikací. Pokud aplikace požívá více databází, musí být sestavována s .CXX soubory všech těchto databází.
Soubor typu .PTX obsahuje deklarace tříd *AllSet*, *Query* a dalších. Je automaticky připojen jako hlavičkový soubor pro soubor obsahující deklarace tříd (.HXX).
Soubor typu .HXX je nutno použít jako hlavičkový soubor pro aplikaci.
Soubor typu .OCD prekompilací vytvořen ze zdrojového textu (.HCD) a je využíván pro vytvoření databázového schématu.

person.hcd		**person.ptx**
persistent class Person	⟶	class PersonAllSet: public PtAllSet
{		{



Ve zdrojovém textu aplikace v C++, který bude pracovat s persistentními objekty, musí být uvedeny tyto hlavičky:

```
#include <poet.hxx>
#include classes.hxx

int main()
{
...
}
```

3.2. AllSet

Ke každé třídě objektů je automaticky generována třída *AllSet*. V této třídě lze sekvenčně přistupovat (metoda *Get()*) ke všem instancím základní třídy a všech tříd z ní odvozených.

```
PersonAllSet  allPersons(pObjBase);
Person *pPerson;
long  i;

for(i = 0; allPersons.Get(pPerson,i,PtSTART) == 0; i++) {
        pPerson->DoSomething();
}
allPersons.Unget(pPerson);          /* mazání objektu */
```

3.3. Query

Ke každé třídě objektů je automaticky generována třída *Query*, která svými metodami umoňuje vytvářet dotazy nad danou třídou nebo nad výsledkem dotazu.
Mějme například persistentní třídu *Person*:

```
persistent class Person {
        PtString Name;              /* PtString je datový typ POETu pro řetězce proměnné
    délky */
};
```

Pak prekompilátor vygeneruje následující třídu:

```
class PersonQuery: public PtQuery {
 public:
        SetName(PtString& param, PtCmpOp co = PtEQ);
};
```

Nyní můžeme použít tuto dotazovací třídu pro vyhledání všech lidí, jejichž jméno začíná písmenem "A". Potřebujeme *PersonAllSet*, protože potřebujeme prohlédnout všechny lidi a také proto, že potřebujeme množinu pro obsažení výsledků.

```
typedef lset<Person*> PersonSet;

PersonAllSet* all = new PersonAllSet(pb);
PersonSet* HusbandsOfEsthers(pb);
PersonQuery husband, wife;

wife.SetName(PtString('Esther') , PtEQ);      /* manželky se jménem Esther */
husband.SetSpouse(&wife);                /* manželé jejichž manželky se jmenují Esther */
all->Query(&husband,HusbandsOfEsthers);
```

3.4. Množiny

Předpokládejme, že chceme, aby každá instance třídy *Person* obsahovala odkazy na své děti. Protože *Person* může mít i více dětí, jeden odkaz nestačí. Pole také není řešením, protože předem nemůžeme určit maximální počet dětí. POET nabízí řešení v podobě nového datového typu zvaného *set*. Deklarace :

```
cset<Person*> people;
cset<int> integers;
```

Množin je několik typů ,*cset* je compact set - jsme omezeni velikostí segmetu (na PC), mimo compact set existují i další - large a huge.

Použijme nyní cset v naší třídě person :

```
persistent class Person {
public:
        cset<Person*> children;
}
```

Nyní můžeme děti do množiny přidávat i z ní číst:

```
Adam.children.Append(&Cain);     /* Cain je dítě 0 */
Adam.children.Append(&Abel);     /* Abel je dítě 1 */

Person* second_child;
Adam.children.Get(second_child, 1L, PtSTART);
```

3.5. Inicializace, otevření a uzavření databáze

Předpokládejme, že chceme napsat aplikaci v POETu. Prvním krokem je inicializace POETu pomocí funkce *InitPoet()*, která vytváří globální objekt *PtRoot*. Tato funkce požaduje jeden volitelný parametr - jméno klienta - které může být zobrazováno ostatním procesům v síti, pracujete-li v prostředí client/server. Jakmile jste zavolali *InitPoet()*, můžete se připojit na server a otevřít databázi použitím *GetBase()*.

```
InitPoet("Client name");
PtBase* pObjBase;
int err = PtBase::POET()->GetBase("LOCAL", "base", pObjBase);
if (err < 0) {
        PtBase::POET()->UnGetBase(pObjBase);
        ErrorExit("Could not open database", err);
}
```

K ukončení se pak používá následující sekvence :

```
PtBase::POET()->UngetBase(pObjBase);
DeinitPOET();
```

3.6. Metoda STORE()

Každý persistentní objekt ví, jak sám sebe uložit, protože je odvozen z *PtObject*. Nejprve je však nutno mu sdělit, do jaké databáze se má uložit. To mu sdělí metoda *Assign()*. Pak lze teprve volat metodu *Store()*.

```
                    ;
            *       Oj;
                    ;

    Oj=             ;

    Oj  m   = '     '
    Oj>     g   h   > OjB      ;

  f       = OjS
            M   g S          ,"   ;

}
```

3.7. Metoda DELETE()

Persistentní objekty také vědí, jak se vymazat z databáze používajíce Oj D . Vymazání objektu jej však neodstraní z RAM, k tomu slouží metoda Oj F g .

```
                D
        *   Oj = 0;

  f  Oj=F
  f     u    D       h    b    ? Y"  =Y
  OjD          ; *  ým z  z       lá  *

  Oj>F  g      ; *       ě í  z  RM, j     ž        ě     x   ji ž é     š   k z   *
}
}
```

3.8. Verze tříd

C , který obsahuje popis schématu databáze umožňuje ukládát několik verzí pro popis každé třídy. Takovým způsobem je zajištěna kompatibilita se staršími verzemi aplikací a není nutné tyto aplikace rekompilovat při každé změně schématu. Aplikace, které používají novější verzi tříd mají přístup i k instancím tříd starší verze. Starší verze schématu jsou nutná pro přístup k datům, která byla uložena s použitím těchto schématů. Konverze databáze do poslední verze schématu je možná pomocí funkce V .

3.9. Indexy

Používání indexů pro danou třídu se povolí při deklaraci třídy použitím atributu *useindex* se jménem příslušného indexu. Deklarace indexu je uvozena klíčovým slovem *indexdef*. Index lze vytvářet pro jeden atribut, pro skupinu atributů, u každého textového atributu lze zadat kolik znaků zleva bude platných pro vytváření indexu.

```
                    {
    h              m [30];
    Address    address;

    useindex   PersonIndex;
};

    x f        I  x
        m  [[10]];                  * I   x bu     ž       a       í h D z   ka *
}
```

Uspořádání u složených indexů probíhá odshora dolů (nejprve je tedy indexováno podle první položky indexu, pak podle druhé, atd. až podle poslední položky). Následuje deklarace pro index založený na jménu a PSČ. Tento index deklarujeme také jako *u q* , což způsobí, že metoda *S* skončí chybou při ukládání duplicity.

```
u q      x f      I  x              {
    name[[10]];
    address.psc;
}
```

Je možno vytvářet indexy i pro atributy referencovaných objektů. Pro každou třídu lze definovat libovolný počet indexů

3.10. Transakční zpracování

POET používá implicitní transakce na úrovni systému při všech operacích zápisu. Změny provedené v databázi jsou zaznamenávány do žurnálu, který je použit v případě zotavení z chyb. Použití žurnálu musí být explicitně povoleno nastavením *R F = 1* v konfiguračním souboru PTSERVER.CFG.

Uživatelsky definované transakce jsou uvozeny příkazy *Bg T* , *Cm T* , *b T* , *T L* . Uživatelsky definované transakce lze do sebe vnořovat, potvrzení veškerých změn v databázi se uskuteční až v okamžiku, kdy je potvrzena transakce definovaná na nejvyšší úrovni. Uživatelsky definované transakce jsou realizovány různými verzemi objektů, po neúspěšném ukončení transakce lze obnovit poslední platnou verzi objektu příkazem *Q* *Rf h*.

3.11. Zamykání objektů

Zamykání objektů v databázi je umožněno prostřednictvím třídy *L k* . Uzamykat lze celý objekt, objekt včetně všech závislých objektů, celý objekt včetně všech referencovaných objektů,

které nejsou definovány *ODMG*, celý objekt včetně všech referencovaných objektů. Definovaný zámek je použit jako parametr příkazu Get() pro čtení hodnoty objektu, zámek je nutno explicitně zrušit použitím zámku jako parametru příkazu *Ug*. Uváznutí při zamykání objektů musí předcházet programátor.
Při použití příkazů *S* a *D* uvnitř uživatelem definované transakce je nastaven implicitní zámek na daný objekt, který je systémem uvolněn až po ukončení transakce.
Zámek je možno definovat i na úrovních *S*, *Op S*, *S* a *ODMG S* metodami *Lock* nebo *Upgrade*, nebo může být použit i jako parametr příkazů *G* a *Q* na příslušnou množinu objektů.

Příklad:

```
Lock(S, Lock::WRITE, WAIT, 10);
MyClass *c = (MyClass *) c->Get("1", 0, START_READ_ONLY);
...
c->Change(...);
...
c->Upgrade(Lock::WRITE);
```

3.12. Funkce vyvolané událostmi

Funkce (*Callback Functions*), které reagují na události při práci s daty jsou součástí modulu nazvaného exception manager. Každá taková funkce může na zahájení databázové operace, její ukončení a může být opakovaně používána při zpracování dat.
Model *Watch&Alert* umožňuje reagovat na situace, kdy jsou data v databázi zpracovávána jiným uživatelem. Lze reagovat na ukládání, aktualizaci a mazání dat a na zamykání dat. Jako reakce na tyto události je vyvolán apříslušná funkce *f*. K volání funkce *f* dojde pouze pokud není instalována odpovídající *Callback* funkce.

3.13. Přístupová práva

Po vytvoření je databáze všeobecně přístupná, používání přístupových práv je třeba nejdříve povolit příkazem *bouhaz*. Administrátorem databáze se pak stane uživatel POETADM, který má oprávnění přidávat a odebírat uživatele nebo skupiny uživatelů a přidělovat jim jednotlivá přístupová práva na úrovni tříd objektů nebo hodnot atributů instancí objektů. S přístupovými právy lze pracovat v grafickém prostředí správy databáze, nebo přímo z aplikace.

4. Dotazovací jazyk OQL

Deklarativní dotazovací jazyk OQL lze používat ve vývojovém prostředí POETu nebo v hostitelském prostředí C++. POET 3.0 implementuje část OQL popsaného ve standardu ODMG-93 [3].

Některá omezení:
1. ve where části dotazu není povoleno testování identity objektu
2. výsledkem dotazu je *<X>*; *bag<X>* není možno používat
3. není implementováno *group by*, *bag*

4. nejsou implementovány podmínky , *x* - ,*f*
5. není možno dynamicky vytvářet objekty, struktury a kolekce
6. není možno volat metody

Pro vazbu OQL a C++ používá své vlastní rozhraní. Každý dotaz je definován jako instance třídy *OQLQu* . Tato instance se pak stane parametrem funkce *q_x u* , která vyvolá dotaz na databázi a vrátí výsledky dotazu.

Následující příklad vybere z databáze všechny osoby, které se jmenují Krbec, jsou starší než 70 roků a mají plat vyšší než 5000:

```
f      Kb
          xf mx                h      x   m   = "K b   ";
f      Kb
              f m        Kb
                  h      . g   > 70
        zf mz        Kb
       h     z          > 5000
```

5. Závěr

POET 3.0 je sým pojetím zaměřen převážně na možnost persistentního ukládání objektů z prostředí C++ do objektově orientované databáze. Vlastní databáze není přístupna z jiných aplikací. Databázové vlastnosti, které POET nabízí jej řadí do kategorie desktopových databází.
V současné době byla vyvinuta nová verze tohoto databázového systému. POET 4.0 se převážně zaměřil na rozšíření týkají se otevřenosti systému. Je plně podporován jazyk OQL dle standardu ODMG-93, dále je možné přistupovat k datům přes standardní rozhraní ODBC a je zakomponována podpora pro OLE technologii.

6. Literatura

[1]Atkinson M. et al.: The Object-Oriented Database System Manifesto.
In: 1st International Conference on Deductive & Object-Oriented Databases, Kyoto, 1989.

[2]Biliris A., Gehani N. H., Jagadish H. V., Roome W. D.: ODE Object Database and Environment.
AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[3]Cattel R.G.G.(ed.): The Object Database Standard: ODMG-93. Morgan Kaufmann Publishers, 1994.

[4]Kroha P.: Objects and Databases. McGraw Hill 1993

[5]Kroha P.: Management of Persistent Objects. DATASEM'95.

[6]Kuklová J. Popelinsky L.: Our Experiences with Public-Domain Object-Oriented Databases. In: Sborník konference SOFSEM'93 Hrdoňov 1993.

[7]Dokumentace k systému POET 3.0.