Objektorientierte Datenbanken

Vorlesung 8 vom 03.06.2004

Dr. Sebastian Iwanowski
FH Wedel

JDO

- **Persistenzkonzept**

  Persistenzfähige Klassen, objektspezifische Persistenz, Persistenz durch Erreichbarkeit

- **Transaktionskonzept**

  mehrere Transaktionsmanagementstrategien

- **Anfragesprache (JDOQL)**

  mit objektorientiertem Fokus, Java-Syntax

- **Konzept zum Management von Datenveränderungen**

  über so genannte Lebenszykluszustände von Daten
  definiert Mechanismen für den Lebenszyklusübergang

- **Datenidentitätskonzepte**

  berücksichtigt unterschiedliche Anforderungen von Datenbank und Programm

Inhalt heute:

Die JDO-Anfragesprache JDOQL, 2. Teil

Zusammenfassung: Funktionalität von JDOQL

Noch fehlende Details der Funktionalität

Vergleich von JDOQL mit weiteren OQL-Funktionalitäten

Erweiterungen von JDOQL durch FastObjects

Zum Vergleich: Vorlesungsskript von Dr. Schmidhauser

Zusammenfassung der Funktionalität von JDOQL

Zusammenfassung der Funktionalität

- **JDOQL liefert Filter für Boolesche Auswertungen**
- **Andere als Booleschen Operationen sind im Filter nicht möglich**
- **Die Eingabe in den Filter ist eine Menge von Elementen derselben Klasse (Collection oder Extent)**
- **Die Ausgabe vom Filter ist eine Collection von Elementen der Eingabeklasse**

Beim vorigen Mal behandelte Funktionalität von JDOQL

- **Filtern beliebiger Eigenschaften von Attributen der Elemente**
- **Arbeiten mit Parametern**
- **Filtern von Existenzeigenschaften von Collection-wertigen Attributen (über `contains` mit Variablen)**
- **Sortieren der Antwort**

Noch fehlende Details der Funktionalität

Funktionalität von JDOQL-Queries

Fragen mit mehr als 3 Parametern:

Interface Query

public void declareParameters (String declarationOfParams);

public Object executeWithMap (Map mapOfParams);

public Object executeWithArray (Object[] arrayOfParams);

- **declarationOfParams und Verwendung im Filter wie zuvor**

Vor dem Aufruf mit `execute` muss die aktuelle Parameterliste als `Map` oder `Array` erzeugt werden:

- **Maps** — **key: Parametername (als `String`)**
  value: eingesetzter Wert vom Typ wie in declarationOfParams angegeben
- **Arrays** — **Elemente: eingesetzte Werte in derselben Reihenfolge wie in declarationOfParams**

- **Typen und Anzahl der Parameter müssen zueinander passen**
  (mit derselben Ausnahme wie eben)

Funktionalität von JDOQL-Queries

Weitere Funktionen zur Effizienzsteigerung:

Interface Query

public void close (Object result);

public void compile ();

public void setIgnoreCache (boolean transactionChangesAreNotConsidered);

Zusammenspiel mit anderen JDO-Funktionen:

- **Einbettung einer JDOQLQuery in eine Transaktion nicht zwingend erforderlich, aber empfohlen**

Vergleich von JDOQL mit weiteren OQL-Funktionalitäten

Funktionen in where-Klauseln

☹ geht in ODMG-FastObjects gar nicht (außer count) ☹

- Finde alle Professoren, die lange Vorlesungen halten:

```
select p
from p in AlleProfessoren
where max(select v.SWS from v in p.liest) >= 4
```

JDOQL-Lösung ?

Existenzquantoren

Beispiel für Existenzquantor:

- Finde die Vorlesungen, in denen weibliche Studenten sitzen und die von Sokrates gehalten werden:

select v

from v **in** AlleVorlesungen

where (v.gelesenVon.Name = „Sokrates“) and (exists s in v.Hörer: s.female)

JDOQL-Lösung ?

2. Variante:

- 1. Frage: Finde die Vorlesungen, in denen weibliche Studenten sitzen.
- 2. Frage: Finde alle Professoren, die solche Vorlesungen halten

Allquantoren

Beispiel für Allquantor:

- Finde die Vorlesungen, in denen nur weibliche Studenten sitzen und die von Sokrates gehalten werden:

select v

from v **in** AlleVorlesungen

where (v.gelesenVon.Name = „Sokrates“) and (**for all** s in v.Hörer: s.female)

JDOQL-Lösung ?

OQL-Funktionalitäten, die kein Analogon in JDOQL haben

Zählfunktion mit `count`:

- Ordne die Vorlesungen von Sokrates nach der Zahl der Hörer:

select v.Titel **from** v **in** AlleVorlesungen **where** (v.GelesenVon.Name = „Sokrates“)

order by count (**select** s **from** s **in** v.Hörer) DESC

Partitionierung: ☹ geht in FastObjects-OQL auch nicht ☹

- Teile Vorlesungen in kurze, mittlere und lange ein:

select * **from** v **in** AlleVorlesungen

groupBy kurz: v.SWS <= 2, mittel: v.SWS = 3, lang v.SWS >= 4

Integration von Objekt-Methoden: ☹ geht in FastObjects-OQL auch nicht ☹

- Finde alle Professoren, deren Gehalt größer als 50000 € ist :

select p **from** p **in** AlleProfessoren **where** p.Gehalt**()** > 50000

Erweiterungen von JDOQL durch FastObjects

Erweiterungen von FastObjects

JDOQL-Standardschnittstelle für Erweiterungen:

Interface PersistenceManager

public Query newQuery (String queryLanguage, Object newQueryInTheOtherLanguage);

FastObjects-Erweiterung für OQL-Anfragen:

```
pm.newQuery ("org.odmg.OQL", queryString);
```

- **queryString ist normale OQL-Anfrage als String**
- **OQL-Anfragen funktionieren grundsätzlich nur innerhalb von Transaktionen**
- **Falls Variable gebraucht werden (z.B. `xt` für `Extent`), kann Definitionsbefehl vorgeschaltet werden:**

```
queryString = "define extent xt for Professoren; SELECT ...";
```

Das Einführungsbeispiel als OQL-Frage in JDO

```
Properties pmfProps = new java.util.Properties();
pmfProps.put("javax.jdo.PersistenceManagerFactoryClass",
             "com.poet.jdo.PersistenceManagerFactories" );
pmfProps.put("javax.jdo.option.ConnectionURL", "fastobjects://LOCAL/MyBase" );
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory( pmfProps );
PersistenceManager pm = pmf.getPersistenceManager();
Transaction txn = pm.currentTransaction();
txn.begin();
Extent AlleProfessoren = pm.getExtent (Professoren.class, true);
```

```
// Frage formulieren:
String queryString = " select p.Name from p in AlleProfessoren " +
                            " where p.Rang = \"C4\" ";
// Frageobjekt erzeugen:
Query query = pm.newQuery("org.odmg.OQL", queryString );
// Frage stellen:
Collection result = (Collection) query.execute();
// Ergebnis auswerten (nicht nötig: Antwort ist bereits Namenmenge !)

txn.commit();
```

Erweiterungen von FastObjects

Weitere Erweiterungen:

Class Queries

public void setOptimizedResults (Query query, boolean resultsWillBeOptimized);

- **liefert als Antwort auf die `query` immer eine `List`, sehr effizient abgespeichert (Details siehe JDOProgrammer`s Guide, S. 110)**

Zum Vergleich:
Vorlesungsskript von Dr. Schmidhauser

Java Data Objects

© Dr. Arno Schmidhauser
Letzte Revision: April 2004
Email: arno.schmidhauser@sws.bfh.ch
Webseite: http://www.sws.bfh.ch/~schmd/db

Dieses Skript stützt sich wesentlich auf die Spezifikation
Java Data Objects, Version 1.0.1, 31. Mai 2003

Inhalt

Queries

1. Queries über Extent
2. Queries über beliebige Collections
3. Parameter (von der Applikation in die Abfrage)
4. Variablen(innerhalb einer Abfrage)
5. Sortierung

6. Kapselung (protected, private) wird nicht beachtet
7. Keine Funktionsaufrufe (wie bei SQL-J oder OQL)
8. Kompilierung (Prepare) möglich

17

Zu 1: Queries über einen Extent werden serverseitig ausgeführt. Ob latente Objekte im Cache berücksichtigt werden ist durch `Query.setIgnoreCache()` `einstellbar.`

Zu 2: Queries über Collections werde clientseitig ausgeführt. Die Collection muss Objekte einer bestimmten Klasse enthalten. Die Klasse ist bei PersistenceManager.newQuery() oder mit Query.setClass() anzugeben. Queries dürfen gemäss JDO Spezifikation über Collections mit *transienten* Objekten abgesetzt werden, wenn die Implementation dies zulässt.

Zu 8: Queries können kompiliert werden (Methode `Query.compile()`). Das kommt insbesondere den relationalen Systemen entgegen, welche sehr oft die Möglichkeit besitzen, SQL-Abfragen vorzubereiten (kompilieren, Rechte prüfen, Ausführungsplan erstellen) und dann wesentlich schneller auszuführen.

Queries sind rein für Abfragen vorgesehen. Modifizierende Queries (analog zu den SQL-Befehlen update, delete, insert) existieren nicht.

Query, Beispiel 1

```
...
Extent ext = pm.getExtent( PMQ.class, false );
Query query = pm.newQuery();
query.setCandidates( ext );
query.setFilter( "qname == p_qname" );
query.declareParameters( "String p_qname" );
String param1 = args[1];
Collection result = (Collection) query.execute( param1 );
Iterator it = result.iterator();
if ( it.hasNext() ) { pmq = (PMQ) it.next(); }
// ... use pmq to add or get messages ...
```

→ **FatMessageConsumer.java: 36-51, FatMessageSelector: 85-98**

18

Dieses Query soll eine Message Queue mit dem Namen in args[1] auffinden. `pm` ist eine Instanz von PersistenceManager.

Die Methode `setCandidates()` legt fest, auf welche Objekte das Query wirken soll. Hier ist es der Extent der Klasse PMQ.

Die Methode `setClass()` legt fest, von welcher Klasse die abgefragten Objekte sind.

Die Methode `setFilter()` beinhaltet das eigentliche Query und resultiert für jedes Candidate-Object in einem boolschen Ausdruck. Der Filter darf Parameternamen enthalten. Der Filter kann die Attribute der Klasse direkt benutzen und auch weiter dereferenzieren. String- und Datums-Attribute und natürlich Zahlenwerte können direkt mit ==, !=, <, >, <=, >= verglichen werden. Boolsche Verknüpfungsoperatoren sind &&, || und ! . Zahlen-Ausdrücke können innerhalb von Filtern mit +, -, *, / gebildet werden. Strings können auch mit + konkateniert werden. Die einzig unterstützten Methodenaufrufe in Filtern sind `String.startsWith( String s )`, `String.endsWith( String s )`, `Collection.isEmpty()` und `Collection.contains( Object o )`.

Die Methode `declareParameters()` legt Name und Typ der Parameter fest. Mehrere Parameter können durch Komma getrennt angegeben werden, beispielsweise `query.setParameters( "String p_qname, int p_priority" )`. Werte von Basistypen können als Basistypen (int, double) oder mit den entsprechenden Wrapper-Klassen (Integer, Double) deklariert sein.

Für die Methode execute() gibt es Varianten mit einem, zwei oder drei Parametern. Ausserdem gibt es die Methoden `executeWithArray( Object[] params )` und `executeWithMap( Map params )`.

Query, Beispiel 2

```
Query query = pm.newQuery();
query.setCandidates( pmq.getAll() );
query.setClass( Message.class );
query.setFilter("priority <= p_prio && sender == p_sender" );
query.declareParameters( "int p_prio, String p_sender" );
Integer param1 = new Integer( args[2] );
String param2 = args[3];
Collection result = (Collection) query.execute(param1, param2);
```

→ **FatMessageSelector: 69-82**

19

Dieses Query stellt ein Collection Query dar. Es sucht nach allen Meldungen einer bestimmten Message Queue, welche eine Priorität < args[2] und den Absender args[3] haben. Im Gegensatz zum Extent Query werden nicht alle Instanzen von Message durchsucht, sondern nur diejenigen in der Vorauswahl von `setCandidates()`.

Query, Beispiel 3

```
Extent ext = pm.getExtent( PMQ.class, false );
Query query = pm.newQuery();
query.setCandidates( ext );
query.setFilter("messages.contains(m)
                && m.priority <= p_prio" );
query.declareVariables( "Message m" );
query.declareParameters( "int p_prio" );
Integer param1 = new Integer( 3 );
query.setOrdering( "qname" );
Collection result = (Collection) query.execute( param1 );
```

20

Dieses Query sucht nach PMQ Instanzen, welche mindestens eine Meldung mit einer Priorität kleiner gleich 3 besitzen. Es benutzt die Variable m und die Operation contains(). Die resultierenden Instanzen werden nach Name sortiert.

Der Unterschied zwischen einem Parameter und einer Variablen ist, dass der Wert eines Parameters vom umgebenden Programm in das Query hineingegeben wird. Eine Variable wird nur durch ihren Namen und ihren Typ definiert, nicht durch ihren Wert. Der Wert einer Variable ergibt sich aus den Objekten von `setCandidates()`.

Variablen-Deklaration werden mit Strichpunkt abgetrennt. Parameter-Deklaration werden mit Komma abgetrennt.

Beim nächsten Mal:
Management von Datenveränderungen