Klein aber fein

Das ODBMS-System db4objects

Stefan Edlich

In die etwas vernachlässigte Welt der Objektdatenbanken (ODBMS) ist wieder etwas Bewegung gekommen. Die embedded Java-Datenbank-Engine db4o hat dabei im letzten Jahr durch einige gewaltige Versionssprünge, Native Queries und interessante Lizenzbedingungen auf sich aufmerksam gemacht. Grund genug, sich neue ODBMS-Entwicklungen anhand eines konkreten Industrieproduktes anzuschauen. Es wird gleichzeitig veranschaulicht, was embedded Objektdatenbanken heutzutage zu leisten imstande sind und wie einfach Java-Persistenz sein kann.

Um in der Welt der Objektdatenbanken erfolgreich zu agieren, ist ein Spagat besonderer Art nötig. Es gilt sowohl eine Nische zu finden, in der Mitbewerber bisher wenig zu bieten haben, als auch ein breites Band an Anwendungsfeldern zu bedienen. Seit 2004 spielt das ODBMS-System db4objects [db4o] in dieser Liga mit. Eine der primären Kernzielrichtungen dieses Produktes ist der Markt der embedded und mobilen Endgeräte.

Gleichzeitig sollte es dennoch möglich sein, auch große Web-Anwendungen unter J2EE oder .NET mit ihren speziellen Anforderungen – wie einem Session- oder Transaktionskonzept – in gewissem Rahmen zu unterstützen. Dies scheint auf den ersten Blick kaum möglich, wenn man Performance, Größe der Datenbank-Engine oder ähnliche Parameter betrachtet.

Zielgruppe und DB-Formate

Mit der Festlegung des Anwendungsbereiches auf eher kleinere embedded oder mobile Systeme ist eine Grenze gesetzt, in der sich eine Objektdatenbank wie db4o von der renommierten relationalen Welt unterscheidet. Bei relationalen Datenbanken handelt es sich anwendungstechnisch oft um eine Data-Warehouse-Architektur, bei der mit verschiedenen Objektsichten auf ein Datenbankschema zugegriffen wird. Meist werden derartige Datenbanken zudem von Administratoren verwaltet.

Im Gegensatz dazu liegt bei Objektdatenbanken und hier auch bei db4o meistens genau ein Java-Objektschema in einer Anwendung vor. Dies ist eine gewisse Einschränkung, die allerdings auch viele Vorteile mit sich bringt. Eine natürliche Folgerung aus dem ODBMS-Ansatz ist, dass keine Energie in das *(impedance mismatch)* Mapping investiert werden muss, was ja bei komplexen Beziehungen und Datenmodellen oft eher unbequem ist. Unter embedded wird hier daher eine Datenbank verstanden, die – wie z. B. auch hsqldb oder H2 – komplett in Anwendungen oder Clients integriert werden kann. Eine weitere Folgerung dieser engen Beziehung zwischen Anwendung und (Objekt-)Daten ist aber auch, dass letztere nicht in einem (SQL) standardisiert zugreifbaren relationalen Format vorliegen müssen.

Die Aufgabe des Data-Warehouse-Gedankens – der ja für Großunternehmen sinnvoll und lebenswichtig ist – bringt aber noch viele weitere Vorteile mit sich: Die Objektdatenbank db4o kann alle Java-Objekte nun in nur einer (verschlüsselten) Datei speichern, die der Entwickler selbst benennt (z. B. `kunden.yap`), kopieren und archivieren kann. Weiterhin kann der Entwickler dann beliebig viele (n) Dateien für bestimmte Objektgruppen öffnen und so bei db4o quasi mit einer beliebigen Datenbankgröße (n*254 GByte) arbeiten.

Proprietäre Speicherformate sind zwar etwas anrüchig, haben aber den Vorteil, dass sie u. a. auf Objekte und einen performanten Zugriff optimiert werden können. Es empfiehlt sich daher für eine moderne Objektdatenbank unter Java, die Ein-/Ausgabe- und Reflection-Schnittstellen offen zu legen, was bei db4o der Fall ist. Dies ermöglicht bspw. einen XML-Export der kompletten Datenbank mit Hilfe eines Plug-In-Reflectors oder die Entwicklung eines Datenbankbrowsers.

Einfaches Java-Objekthandling

Um zu beurteilen, ob ein System einfach zu handhaben ist, werden oft zwei Größen herangezogen: einerseits die Zeit, die man benötigt, um das System erfolgreich aufzusetzen, und andererseits die Lernkurve.

Um db4o herunter zu laden und ein beliebiges Java-Objekt mit `set` zu speichern, benötigt man nur wenige Minuten. Dies dürfte mit keinem anderen Datenbanksystem schneller gehen. Zum einen liegt dies daran, dass die Engine selbst auch als nur eine ca. 400 KByte große Jar-Datei vorliegt. Diese muss dann in der Entwicklungsumgebung nur noch zum *Classpath* des Projektes hinzugefügt werden. Zum anderen benötigt es ganze drei Codezeilen, um die Engine zu importieren, eine Instanz von ihr und des Geschäftsobjektes zu erzeugen und dieses mit `set()` zu speichern. Mit einem `commit()`-Befehl wären es vier Zeilen. Um ein Datenbankschema braucht sich der Anwender nicht zu kümmern. Selbst ein Weiterarbeiten mit geänderten Klassen (z. B. das Umbenennen eines Feldes) ist einfach so möglich, d. h. viele Schemaänderungen machen einem lauffähigen db4o-System nichts aus. Gerade hier besteht ein wichtiger Unterschied zu relationalen Mappern und Datenbanken mit Objektfeatures. Selbst wenn diese grafische Tools anbieten, ist eine Änderung des Schemas meistens mit ein wenig Ärger für die Anpassung der Mappings verbunden.

Ein extrem wichtiger Aspekt für eine transparente Integration in (auch bestehenden) Java- oder C#-Code ist aber, dass weder ein Postcompiler noch ein Ableiten von speziellen Klassen nötig ist. Einige Produkte mit ähnlicher Zielrichtung erfordern dies, was den Entwicklungsprozess aber schwer stören kann.

Der Code in Listing 1 zeigt ein komplettes Java-Beispiel. Es werden drei Geschäftsobjekte gespeichert und gesucht, eines wird modifiziert und danach gelöscht. Man sieht, dass das Suchen, Modifizieren und Löschen von Objekten nur wenige Codezeilen in Anspruch nimmt.

```
ObjectContainer db=Db4o.openFile("C:/tmp/t03.yap");
try { // Speichert drei Objekte
  db.set(new Customer("Max Fisch","TOP INC",25));
  db.set(new Customer("Tim Rice","NEW Ltd.",50));
  db.set(new Customer("Jane Doe","Alk AG",35));

  ObjectSet result=db.get(new Customer());
  while(result.hasNext()) { // Zeige alle Objekte
    System.out.println(result.next());
  }
  result.reset();
  result=db.get(new Customer("Jane Doe"));
  Customer found = (Customer) result.next();
  found.setAge(36); // Hat Geburtstag
  db.set(found); // Objekt wieder speichern (update)
  db.delete(found); // Feier endete mit Unfall ...
  db.commit()
}
finally { db.close(); }
```

Listing 1: Safe, Search, Modify, Delete von Objekten

Einfache Anfragen

Wie Listing 1 zeigt, enthält db4o unter anderem einen extrem einfachen Abfragemechanismus. Dieser wird *Query-by-Example* (QBE) genannt und verwendet *Search-by-Pattern*. Um ein Java-Objekt oder mehrere Objekte zu finden, wird ein „leeres" Objekt mit einem Java-Konstruktor erzeugt. Nach dem optionalen Setzen von Feldern der Klasse, die die Suchwerte bestimmen, kann der Objektcontainer schnell alle passenden Objekte finden und in einer Collection zurückliefern.

Dieser QBE-Mechanismus erlaubt zwar keine komplexen Abfragen, ist aber sehr einfach anzuwenden. Dennoch kommt kein modernes ODBMS heutzutage ohne komplexe Abfragesprachen aus. Wie db4o diesen innovativen und forschungsintensiven Bereich abdeckt, wird gegen Ende untersucht.

Objektmanagement

Ein für die Performance wichtiges Feature ist die *Activation* und *Update Depth* bei „tiefen" Objekten. Unter tiefen Objekten werden hier solche Objekte verstanden, die eine lange Kette von Objekten in Objekten etc. enthalten. Bei komplexen Objekten der Industriepraxis sind Tiefen von 10 – 20 keine Seltenheit, da oft auch triviale Objekte wie Strings oder Bibliotheken dazu beitragen, die Objekttiefe stark zu erhöhen.

Beim performanten Speichern, Laden oder Updaten von Objekten ist die Kontrolle der tatsächlich ausgeführten Objekttiefe von entscheidender Bedeutung, da sie sich extrem auf die Performance der Anwendung auswirkt. Die Standard-Tiefe bei db4o ist 1. Sie kann aber jederzeit beliebig angepasst werden.

Ähnliche Probleme treten beim Löschen von Objekten auf (*recursive deletions*), die mit Einstellungen am Container gelöst werden können. Der folgende Codeabschnitt zeigt die Einstellungen für ein komplettes *delete* der enthaltenen Objekte:

```
Db4o.configure().ojectClass ("com.db4o.f1.chapter2.Car")
.cascadeOnDelete(true);
```

Dabei kann die Datenbank keine Lösch-Befehle auf Objekte ausschließen, auf die Java oder C# noch Referenzen hält. Daher ist ein *deep-delete* mit einiger Vorsicht zu verwenden.

Aber natürlich kann es für Anwender genauso wichtig sein, dass alle Objekte (bspw. schiefe Bäume) transparent, also automatisch, aktiviert werden. Dies wird demnächst unterstützt.

Die Kraft von Objektdatenbanken zeigt sich teilweise erst dadurch, inwieweit das komplette Objekt ohne Ausnahmen – wie private Felder oder komplexe Datentypen – korrekt gespeichert wird. Und zwar ohne Präcompilierung oder erzwungenes Ableiten einer ODBMS-Klasse. Neben allen Grunddatentypen und Objekten jeder Sichtbarkeit werden auch Collections und Arrays so behandelt, wie man es bei der bereits bekannten API mit „normalen" Objekten erwartet. Das heißt, dass bspw. auch ein QBE auf ein Array das gewünschte Ergebnis liefert.

Interessant ist ebenfalls das Handling von Interfaces, abstrakten Klassen und Vererbungen, die ja weniger beim Speichern als beim Laden der Objekte problematisch sein können: Zunächst muss der Entwickler die Typen der Objekte kennen, die wiederum aus der Datenbank geholt wurden. Ganz normal erwartet man entsprechend auch, dass *Example Queries* auf einer Unterklasse das erwartete Ergebnis zurückliefern. Also `A a = new B()`, wobei `B` eine Unterklasse von `A` ist, und dann ein `get(a)` ausgeführt wird. Für QBE-Abfragen auf Superklassen wird eine Abfrage mit `Klasse.class` formuliert.

Es zeigt sich, dass db4o dem Java-Entwickler keine Einschränkungen irgendwelcher Art auferlegt und keine Annahmen über die zugrunde liegenden Objekte trifft.

Replikation

Replikation ist ein Thema, welches gerade in embedded ODBM-Systemen lange vernachlässigt wurde, aber dort sehr wichtig ist. Bereits die Version 4.5 von db4o stellt einen einfachen Mechanismus zur Verfügung, um Java-Objekte in mehreren Datenbanken konsistent zu speichern. Der Replikationsprozess ist standardmäßig bidirektional, kann aber auf aktualisierende Abonnenten *(Updating Subscribers)* eingestellt werden.

Der Mechanismus des Objektabgleichs geht über UUIDs vonstatten, die in allen Datenbanken vorliegen. Diese können automatisch vom Objektcontainer generiert oder aus Ressourcengründen auch für Objekte individuell vergeben werden. Der Aufruf für ersteres Beispiel lautet dann wie folgt:

```
Db4o.configure().generateUUIDs(Integer.MAX_VALUE);
Db4o.configure().generateVersionNubers(Integer.MAX_VALUE);
```

Bei dem Update zweier gleicher Objekte auf verschiedenen Replikaten können immer Inkonsistenten auftreten. Diese werden in der synchron symmetrischen Einstellung der db4o-Umgebung mit einer `ConflictHandler`-Methode behandelt, die der Entwickler selbst implementieren muss. Sie bekommt die inkonsistenten Objekte und liefert das richtige Objekt einfach als Rückgabewert zurück. Um keine der Varianten zu akzeptieren, wird null zurückgeliefert:

```
ReplicationProcess replication =
  desktop.ext().replicationBegin(db2,
    new ReplicationConflictHandler() {
      public Object resolveConflict(
        ReplicationProcess replicationProcess,
        Object a, Object b) {
          return a;
        }
    }
  );
replication.setDirection(db1, db2);
```

In der Java-Version 5.1 von db4o ist zudem ein Replikationsmechanismus (dRS) integriert, mit dem Objekte mit Hilfe von Hibernate einfach auf relationale Datenbanken repliziert werden können. Dies ermöglicht z. B. das Erfassen von Daten auf

mobilen Endgeräten in Objektform und später das Replizieren dieser Daten in die relationale Datenbank des Unternehmens. Eine andere Anwendung ist der Transfer von Daten zwischen relationalen Datenbanken. Dieser kann jetzt sehr einfach mit dem dRS von db4o über Java-Objekte in einer Transferdatenbank geschehen. Dies alles sind Paradebeispiele für den Einsatz von embedded Datenbanken.

Neue Abfragesprachen

Wie bereits erwähnt, ist die vorgestellte QBE-Abfrage für triviale Abfragen ideal, aber keineswegs mächtig. Daher bietet db4o zwei weitere Abfragesprachen an. S.O.D.A. (Simple Object Database Access, [SODA]) und *Native Queries* (ab db4o-Version 5).

Mit der auf Sourceforge gehosteten SODA API kann der Anwender eigene komplexere Suchbäume in Form von Objektgrafen aufbauen. Das nachfolgende Java-Beispiel (`q2` ist vom Typ `Query` und kann mit `execute` ausgeführt werden) gibt einen Eindruck dieser API:

```
q2.descend("alter").constrain(50).smaller(); // Findet alle < 50 Jahre
```

Viel interessanter – auch unter dem Gesichtspunkt der Abfrageoptimierung – ist der sehr junge und vielversprechende Ansatz der *Native Queries* [COOK06]. Diese Art der Abfragen ist ab db4o 5.0 implementiert und geht auf Forschungsarbeiten von William Cook aus dem Jahr 2005 zurück.

Die Ausgangsidee ist, dass stringbasierte Abfragesprachen wie SQL zwar etabliert sind, aber manchmal nicht zur OO-Welt passen und fehleranfällig sein können. Selbst JDO mischt Strings mit Objekten, um Abfragen zu formulieren. Daher ergibt sich zwangsläufig die Frage, ob nicht Java oder C# selbst die besten Abfragesprachen für Queries sind. Ein Beispiel:

```
List<Cat> cats = db.query(new Precidate<Cat>(){
  public boolean match(Cat cat) {
    return cat.name.startsWith("Tom")
          && cat.weight >= 3 && cat.weight <= 5;}});
```

Interessant ist dabei das Folgende:

- Die Abfrage ist Teil der Klasse und in Java oder C# formuliert, also der natürlichen „Lieblingssprache“ des Entwicklers (also auch mit &&, | |, <, >, =)!
- Parameter können bei Bedarf von außen als final in die Abfrage einfließen.
- Die Abfrage ist refactoring-safe! Das Umbenennen eines Klassenmembers mit den üblichen Refactoring-Tools bezieht die Abfrage mit ein.
- Die Abfrage kann Java 1.5 Generics beinhalten.
- Die komplette Abfrage wird vom Compiler der Entwicklungsumgebung sofort beim Editieren auf syntaktische Korrektheit geprüft!
- Das ODBMS kann die Abfrage optimieren.

Das herausragende an *Native Queries* ist auch, dass die Abfragen im `match()` beliebige Java-Methoden enthalten können. Dies macht *Native Queries* zu einer extrem mächtigen Abfragesprache.

Für manche mag es ungewohnt wirken, dass Abfragen dadurch in ihrer Lokalität immer Teil von Klassen, also Methoden sind. Die oben genannten Vorteile dieses Ansatzes sind nach Ansicht des Autors so immens, dass er sich sicher ist, dass *Native Queries* in Zukunft ein fester Bestandteil vieler Datenbanken sein wird. Dies hat z. B. auch schon Microsoft erkannt und wird einen ähnlichen Ansatz (LINQ) in C# 3.0 integrieren.

Fazit

Als ODBMS-Vertreter ist db4o eine sehr kleine, sehr schnelle und recht leistungsfähige Java-Objektdatenbank-Engine mit grafischem Browser (s. Abb.1), die fast vergessen lässt, eine Persistenzschicht zu haben. Bei db40 sorgen wenige Codezeilen – und erst recht das fehlende Tabellenmapping – für eine sehr schnelle Implementierung, auch für wenig Fehler im Code. Und natürlich ist die Datenbank extrem performant [POLE], da auf das Objektmapping verzichtet werden kann.

Beeindruckt hat den Autor neben der Einfachheit (QBE und S.O.D.A.) und der Mächtigkeit (Native Queries) der Abfragesprachen auch die Offenheit des Systems, was bspw. Reflection und Datei-Ein-/Ausgabe angeht.

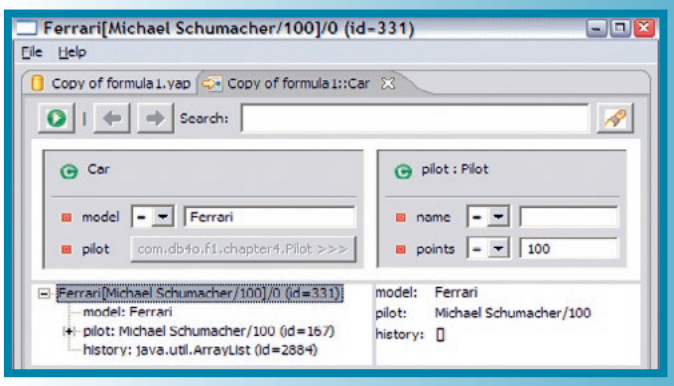

Abb. 1: db4o-Objektbrowser

Für die Industrie dürfte es – neben dem embedded Einsatz – ebenfalls sehr interessant sein, ODBM-Systeme wie db4o in einer gemischten Java-, .NET- oder Mono-Umgebung laufen zu lassen. Dies kann sich anbieten, da einerseits die Dateiformate kompatibel zueinander sind und es andererseits db4o-Engines für beide Welten (.Net / C# / VB und Java) gibt.

Es gibt eine GPL-Version der Datenbank-Engine und eine *Commercial License*, die notwendig wird, wenn Produkte mit db4o vertrieben werden müssen.

db4o als gelungenes Beispiel einer Java-ODBMS versucht auch größere transaktionale Systeme zu bedienen, hat aber mit der Kernausrichtung auf die embedded oder mobile Welt einfach andere Vorteile zu bieten.

Literatur und Links

[COOK06] W. Cook, C. Rosenberger, Native Queries for Persistent Objects, in: Dr. Dobb's, 02/06

[db4o] ODBMS-System db4objects, `http://www.db4objects.com`

[ODBMS] Portal zu Objektdatenbanken, `http://www.odbms.org`

[POLE] Benchmarks für Opensource-Datenbanken, `http://www.polepos.org`

[SODA] S.O.D.A. – Simple Object Database Access, `http://sourceforge.net/projects/sodaquery/`

Dr. Stefan Edlich ist Professor für Softwaretechnik an der TFH-Berlin. E-Mail: edlich@gmail.com.