JavaSet – eine Spracherweiterung von Java um persistente Mengen

Markus Schordan und Harald Kosch

Institut für Informationstechnologie, Universität Klagenfurt,
A-9020 Klagenfurt, Österreich
markuss(harald)@ifi.uni-klu.ac.at

Zusammenfassung Wir stellen in diesem Beitrag eine Spracherweiterung von Java vor, die es ermöglicht elegant persistente und transiente Mengen von Objekten zu manipulieren und deklarative Mengenabfragen zu formulieren. Durch diese Spracherweiterung ist der Zugriff auf den persistenten Datenspeicher für den Programmierer transparent. Die Übersetzung von JavaSet beinhaltet eine effiziente Objektalgebra Optimierung von Mengenoperationen und deren Abbildung auf einen Ausführungsplan.

1 Einleitung

Java[1] ist sicher die Programmiersprache, die seit Ihrer Einführung am meisten von sich hat reden lassen. In Verbindung mit Web-Browsern, mit Internet Anwendungen und durch das Konzept des Network Computers hat Java eine hohe Popularität erreicht [1]. Java ist nicht nur eine Programmiersprache für das Internet, sondern auch eine typsichere und eine konsequent objektorientierte Sprache. Eine Vielzahl von Paketen und Schnittstellen werden dem Programmierer zur Verfügung gestellt. Allerdings fehlen saubere Abstraktionen von hohen Niveau um persistente und transiente Datenmengen in Java zu definieren und zu verarbeiten, die sich in Zeit und in Größe verändern können.

In diesem Zusammenhang stellen wir JavaSet, eine Spracherweiterung von Java um Mengen vor. Diese Erweiterung ermöglicht es, elegant persistente und transiente Mengen von Objekten zu deklarieren, zu manipulieren und auf diesen komplexe Operationen zu formulieren. Auf einzelne Elemente wird assoziativ (via ihrem Inhalt) zugegriffen, was vor allem für größere Mengen von Vorteil ist. Die zur Verfügung gestellten Operationen (Vereinigung, Durchschnitt, Existenz- und Allquantor, um nur einige zu nennen) sind einfach zu beherrschende und vor allem bekannte Operationen, deren Semantik sauber definiert ist. Darüberhinaus ermöglicht die Formulierung von deklarativen Anfragen mit Hilfe eines speziellen Ausdrucks (Select-Ausdruck) den Einsatz effizienter Optimierungstechniken.

Praktische Erfahrungen aus früheren Arbeiten zu einer Spracherweiterung von Modula-3 um Mengen [2] konnten bei dem Design von JavaSet gewinnbringend eingesetzt werden, wobei die M3Set Funktionalität hinsichtlich Persistenz und Optimierung wesentlich erweitert wurde.

[1] Java^{TM} ist ein eingetragenes Warenzeichen von Sun Microsystems Inc.

Dieser Beitrag stellt die JavaSet Sprachspezifikation vor (Kapitel 3). Besonderes Augenmerk wird auf den JavaSet Übersetzer (Kapitel 4) mit seinem integrierten Optimierer für komplexe Mengenabfragen gelegt. Darüberhinaus wird die Anbindung an einen persistenten Speicher beschrieben. Das folgende Kapitel 2 analysiert bisherige Arbeiten.

2 Bisherige Arbeiten

Persistenz in Programmiersprachen hat in den letzten Jahren immer mehr an Bedeutung gewonnen (für Persistenz in Java gibt es eine spezielle Workshopserie [3]). Oft wird Persistenz dabei als Datei-Persistenz verstanden, d.h. Daten werden mit Hilfe von Serialisierung in Dateien [4] geschrieben. Diese Art der Persistenz ermöglicht offensichtlich nur sehr eingeschränkte Datenoperationen und ist für die Verknüpfung größerer persistenter Datenmengen ungeeignet. Ein schnellerer Zugriff auf die persistenten Daten kann durch kleinere Serialisierungseinheiten ermöglicht werden. Erweiterte Varianten verändern die Objektsprache oder den Interpreter JVM [5], um Persistenz in Java zu gewährleisten. Das Ziel ist eine flexiblerer Zugriff auf die Daten.

Neben der Datei-Persistenz gibt es noch eine Reihe von kombinierten objektorientierten Datenbank- und Programmiersprachen, wie z.B. der OQL C++ Standard [6] verwirklicht im O_2 C++ binding [7]. Diese Kombination ermöglicht volle Datenbankfunktionalität und erleichtert die Verwaltung von großen Datenmengen in der Programmiersprache. Die Kombination von Datenbank- und Programmiersprachen ist in der Regel sehr umfangreich [4] und der Programmierer braucht ein fundiertes Datenbankwissen, um effizient arbeiten zu können. Erste Ansätze einer solchen Kombination mit Java werden von der ODMG (Object Data Management Group) erarbeitet [8]. Weiters ist eine Datenbankanbindung über ein JDBC/ODBC API [9] möglich. Die eingebettete Datenbanksprache unterscheidet sich hier grundsätzlich von der Programmiersprache und macht die Programmierung persistenter Daten im allgemeinen schwieriger.

Einige Autoren haben in den oben erwähnten Zusammenhängen erkannt, daß persistente Mengen ein hohes Abstraktionsniveau bieten um persistente Daten zu verarbeiten, ohne den Sprachumfang unübersichtlich zu erhöhen.

Mehrere Ansätze, Mengen in eine Programmiersprache zu integrieren sind bekannt. Erste Konzepte sind die Datenbanksprachen-Orientierten FAD und SVP [10]. Diese Sprachen benützen nur beschränkte objektorientierte Konzepte. Andere Ansätze, wie ParSet implementiert auf dem SHORE C++ Object Store [11], bieten keine direkte Sprachunterstützung.

Daneben gibt es auch mengenbasierte Sprachen, wie etwa SetL [12]. Die Mengenoperatoren sind denen von JavaSet ähnlich, allerdings nur auf transiente Mengen anwendbar. Im Gegensatz zu SetL unterstützt StarSet [13] auch Persistenz. Zum Laden und Speichern von Mengen und Klassen muß der Anwender spezielle Funktionen verwenden. Der Zugriff auf Mengen ist daher nicht transparent.

3 Sprachdefinition

Die Aufnahme von Mengen in das JDK1.2 zeigt, daß Mengen vielfach verwendete Datenstrukturen sind. Durch JavaSet stellen wir ein weit umfassenderes Mengenkonzept vor, das Mengenoperationen und einen mächtigen Select-Ausdruck beinhaltet, der auf Elemente von Mengen ähnlich wie in Datenbankabfragesprachen zugreifen läßt, wobei die elegante mathematische Notation für Mengen verwendet wird. Weiters wird der Zugriff und die Manipulation von persistenten Mengen ermöglicht, wobei das Lesen und Schreiben als auch der Verbindsaufbau zum persistenten Speicher transparent ist. Der Zugriff auf persistente Mengen wird durch die Optimierung von Select-Ausdrücken wesentlich beschleunigt. Dadurch wird die effiziente Verwendung persistenter Daten in einer objektorientierten Programmiersprache wie Java wesentlich erleichtert.

Mengentyp. Alle Elemente einer Menge sind vom gleichen Typ, der Elementtyp der Menge genannt wird. Ist der Elementtyp einer Menge T, dann wird der Typ der Menge selbst als $T\{\}$ geschrieben. Wir nennen eine Variable vom Typ $T\{\}$ eine Mengenvariable. Die Größe einer Menge ist nicht Bestandteil des Typs. Neben den Methoden der Klasse Object sind, wie auch bei der Implementation von transienten Mengen ab JDK1.2, die Methoden `size()` und `isEmpty()` in Mengen vorhanden. JavaSet stellt diese Funktionalität auch für persistente Mengen zur Verfügung. Der Elementtyp einer Menge kann ausschließlich vom Typ Referenztyp [14] sein. Mengen sind dynamisch erzeugte Objekte und können Variablen vom Typ Object zugewiesen werden. Alle Methoden der Klasse Object können für eine Menge aufgerufen werden.

Mengenvariablen. Eine Variable eines Mengentyps enthält eine Referenz auf ein Mengenobjekt. Die Deklaration einer Variable erzeugt kein Mengenobjekt. Es wird nur die Variable selbst erzeugt.

Erzeugung von Mengen. Eine Menge wird durch eine Mengeninitialisierung, einen Mengenerzeugungssausdruck oder einen Select-Ausdruck (sh. Kapitel 3) erzeugt.

Die Mengeninitialisierung ist syntaktisch äquivalent der von Feldern [14] §10.6. Zusätzlich kann angegeben werden, ob die Menge an einen persistenten Speicher gebunden werden soll oder nicht.

(1) { $Variableninitialisierer_{opt}$,$_{opt}$ } @ $PSBezeichner$
(2) { $Variableninitialisierer_{opt}$,$_{opt}$ }

Durch Angabe von $PSBezeichner$ (1) wird die Menge an einen persistenten Speicher gebunden. Diese Information, an welche persistente Menge eine Variable gebunden ist, wird zur Übersetzungszeit bei der Optimierung von Mengenausdrücken verwendet. Durch die Erzeugung des Objektes wird auch die Verbindung zum persistenten Speicher hergestellt, falls diese noch nicht existiert hat. Existiert die Menge noch nicht im persistenten Speicher, so wird diese angelegt und mit den angegebenen Elementen initialisiert. Enthält die Menge bereits Objekte, so kann auf diese über das Mengenobjekt zugegriffen werden. Wird $PSBezeichner$ nicht angegeben (2), so wird eine transiente Menge initialisiert.

```
Student var=new Student("Gunther");
Person{} personen={new Person("Karin"), var} @ "db1:pers1";
```

In obigem Beispiel wird die Variable `personen` an eine Menge im persistenten Speicher `db1:pers1` gebunden. Ist die Menge noch nicht vorhanden, so wird eine polymorphe Menge erzeugt, die zwei Elemente enthält. Für alle Elemente der Menge gilt, daß der Typ zuweisungskompatibel zum Elementtyp der Menge auf der linken Seite der Zuweisung ist (Klassendefinition sh. Abb. 1).

Zuweisung. Wird einer Mengenvariable eine Menge zugewiesen, so wird die Referenz auf die Menge, für die die Mengenvariable der linken Seite steht, überschrieben. Die Information für die Persistenz wird jedoch nicht überschrieben. Die Referenzsemantik bleibt also für Mengen, nicht aber für die Persistenz erhalten. Die Information, an welche persistente Menge eine Variable gebunden ist, wird nicht überschrieben. Dies macht es einfach, neue Mengen zu generieren und persistent zu machen. Auch ist die Veränderung von persistenten Mengen dadurch leicht möglich.

```
Student{} stud3={} @ "db1:studenten1";
Student{} stud4;
stud4=stud3;
```

Es wird ein Mengenobjekt mit Elementtyp `Student` erzeugt und die Verbindung zum persistenten Speicher `db1` hergestellt. Dieses Objekt wird der Variable `stud3` zugewiesen. Wird der Variable `stud3` eine andere Menge zugewiesen, wird diese Menge persistent und überschreibt im persistenten Speicher die an die Variable gebundene Menge. Wird die Anweisung `stud4=stud3` ausgeführt, so kann über die Variable `stud4` auf alle Objekte zugegriffen werden, auf die auch durch `stud3` zugegriffen werden kann. Durch eine Zuweisung können also Mengen transient gemacht werden und umgekehrt.

Mengenoperationen und erweiterter Mengenerzeugungsausdruck. Zusätzlich zum Mengenerzeugungsausdruck mittels `new` stellen wir auch einen erweiterten Mengenerzeugungsausdruck zur Verfügung, durch dessen Verwendung einfach Elemente in eine Menge aufgenommen oder von dieser entfernt werden können. Der Ausdruck ist syntaktisch ähnlich dem Initialisierungsausdruck, muß jedoch mindestens ein Element enthalten. Der Typ des Ausdrucks ist der allgemeinste Typ der angegebenen Elemente. Die Operationen Vereinigung, Durchschnitt und Differenz werden durch binäre Operatoren, die ausschließlich auf Mengen angewendet werden können, ausgedrückt.

```
personen=personen+{var}+stud3;
personen+={var}+stud3;
```

Der Operator '+' wird für die Vereinigung von Mengen verwendet. Es wird auf der rechten Seite eine einelementige Menge erzeugt, die dann mit der Men-

ge `personen` und `stud3` vereinigt wird. Für die Operationen Durchschnitt und Differenz werden die Operatoren `'*'` bzw. `'-'` verwendet.

Der Ergebnistyp einer Vereinigung ist der allgemeinere Typ der beiden Operanden. Wird der Durchschnitt zweier Mengen gebildet, so ist der speziellere Typ der Ergebnistyp. Bei der Differenz ist der Ergebnistyp der Typ des ersten Operanden.

```
class Vorlesung {                    class Student extends Person {
   public String titel; ...             public Vorlesung vorl;
}                                       Student(String name){...} ...
class Person {                       }
   protected String name;            class Paar {
   protected Adresse adresse;           private Person p1,p2;
   Person(String name){...}             Paar(Person p1, Person p2) {
   public Adresse adr() {...}...         ...} ...
}                                    }
Person{} personen = {} @ "db1:personen1";
Student{} stud1 = Student{} @ "db1:studenten1";
Student{} stud2 = Student{} @ "db1:studenten2";
```

Abbildung1. Klassendefintionionen und Programmfragment.

Select-Ausdruck. Mittels Select-Ausdrücken kann in JavaSet auf Objekte in Mengen, ähnlich wie einer objektorientierten Datenbankabfrage (z.B. OQL [15]), zugegriffen werden. Der Select-Ausdruck erzeugt aus verschiedenen Mengen durch die Angabe eines boolschen Ausdrucks eine neue Menge. Diese kann auf zwei Arten erzeugt werden: objekterhaltend oder objekterzeugend. Mengen werden ähnlich der mathematischen Schreibweise angegeben.

$$\{Selector \mid B_1 \texttt{ in } Menge_1, \ldots, B_n \texttt{ in } Menge_n \texttt{ :: } BoolAusdruck \}$$

Jeder Bezeichner B_i $(1 \leq i \leq n)$ kann nur einmal in einem Select-Ausdruck auftreten. Wird als Selector ein Bezeichner B_i angegeben, so wird eine objekterhaltende Operation durchgeführt, sodaß die neue Menge nur Objekte enthält, die schon vor der Auswertung des Ausdrucks existiert haben. Ist der Selector ein Konstruktor einer Klasse, so werden Objekte erzeugt. Als Parameter können beliebige Pfadausdrücke von Objekten verwendet werden. Weiters kann auch ein Pfadausdruck als Selector angegeben werden.

```
(1) {p|p in personen :: p.adr()==null};
(2) {st1| st1 in stud1, st2 in stud2 ::
                                  st1.adr().equals(st2.adr())};
(3) {Paar(st1,st2)| st1 in stud1, st2 in stud2 ::
                                  st1.adr().equals(st2.adr())};
```

In Beispiel 1 wird die Menge aller Personen erzeugt, für die keine Adresse eingetragen ist. Beispiel 2 zeigt, wie mittels eines Select-Ausdrucks ein semi-join ausgedrückt werden kann. Es werden genau jene Objekte der Menge `stud1` in die Ergebnismenge aufgenommen, deren Adressen gleich denen eines Objekts der Menge `stud2` sind. Im dritten Beispiel wird der Konstruktor der Klasse `Paar` als Selector angegeben. Es wird eine Menge von Objekten der Klasse `Paar` erzeugt, wobei als Parameter des Objektkonstruktors die Namen von je zwei Elementen der Mengen `stud1` und `stud2` übergeben werden. Der Elementtyp der Ergebnismenge ist daher `Paar`. Die Ausdrücke $Menge_i$ $(1 \leq i \leq n)$, sind beliebige Mengenausdrücke. Select-Ausdrücke können somit beliebig geschachtelt werden.

foreach. Wir erweitern Java um eine zusätzliche Anweisung, die es erlaubt, Operationen auf Elementen von Mengen zu definieren.

`foreach (` *Bezeichner* `in` *Mengenausdruck* `)` *Anweisung*

Die einzelnen Elemente der Ergebnismenge von *Mengenausdruck* werden an *Bezeichner* gebunden. Die Reihenfolge, in der diese Bindungen erfolgen, ist nicht festgelegt, da Mengen ungeordnet sind. Es ist sichergestellt, daß alle Elemente der Menge ausgewählt werden und *Anweisung*[2] für jedes Element genau einmal ausgeführt wird. Ist die Menge leer, so wird *Anweisung* nicht ausgeführt. Der Bezeichner ist lokal und existiert nur innerhalb des Blocks.

Sonstige Ausdrücke. Weiters stellen wir einen Allquantor, einen Existenzquantor und einen Ausdruck zur Verfügung, der testet, ob ein Element in einer Menge enthalten ist; dieser ist aus Effizienzgründen als ein Ausdruck[3] in der Sprache enthalten. Der Ergebnistyp aller drei Ausdrücke ist vom Typ boolean.

(1) `all` *Bezeichner* `in` *Mengenausdruck* `::` *BoolAusdruck*
(2) `any` *Bezeichner* `in` *Mengenausdruck* `::` *BoolAusdruck*
(3) *Ausdruck* `in` *Mengenausdruck* `::` *BoolAusdruck*

4 Übersetzung

JavaSet ist eine Obermenge von Java. Bei der Übersetzung (sh. Abbildung 2) eines JavaSet Programms werden reine Java-Konstrukte, Mengenoperationen und Select-Ausdrücke separat übersetzt. Die Mengenoperationen werden durch Optimierungen auf der Normalform von Mengenausdrücken optimiert. Die Select-Ausdrücke werden durch spezielle Strategien unter Zuhilfenahme einer Objektalgebra optimiert (sh. 4.1).

Die optimierten Mengenoperationen und Select-Ausdrücke gehen in die Erstellung eines Ausführungsplans ein. Der Ausführungsplan legt die Verarbeitungsreihenfolge der Mengenoperationen fest und ist auf die Größenverhältnisse

[2] Eine Anweisung kann auch ein Block sein; analog der Definition für `for` in Java.
[3] Dieser Ausdruck kann auch durch `!(`{*Ausdruck*} $*$ *Mengenausdruck*`).empty()` ausgedrückt werden.

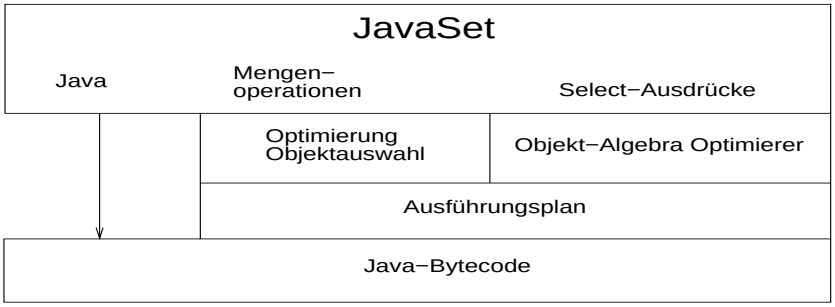

Abbildung2. Übersetzung eines JavaSet Programms.

der zu verarbeitenden Mengen optimiert. Da sich jedoch die Größenverhältnisse während der Laufzeit (insbesondere über einen längeren Zeitraum) so verändern können, daß die vorgenommene Optimierung nicht mehr optimal ist, ist es notwendig Anpassungen zur Laufzeit vorzunehmen (sh. 4.2).

4.1 Optimierung

Die Optimierung beruht auf dem Ansatz der algebraischen Anfrageoptimierung [16]. Dazu wird der Select-Ausdruck in einen objektorientierten algebraischen Ausdruck übersetzt, der auf einer Objekt-Algebra aufbaut, deren Operatoren jeweils eine atomare Ausführungsfunktion auf den Mengen modellieren. Die wichtigsten Operatoren sind : *Join* und *Selection* (ähnlich wie in einer relationalen Algebra), dazu der implizite Objekt-Join *OJoin*, der auf den eindeutigen OIDs operiert, weiters der *Flatten*, der eine Menge von Mengen einebnet und der *Mat*, der eine Komponente eines Pfadausruckes materialisiert [17].

Der Select-Ausdruck wird von dem Übersetzer in einen Verarbeitungsbaum (engl. Processing Tree) übersetzt. Dieser Baum ist eine ideale Abstraktion um die Verarbeitungsreihenfolge der Objektalgebra-Operatoren darzustellen. Die inneren Knoten eines Verarbeitungsbaumes stellen die Mengenoperatoren dar und die Blätter bezeichnen die zu verarbeitenden Mengen. Die gerichteten Kanten zwischen den Knoten beziehen sich auf den Datenfluß (z.B. ein Join verarbeitet zwei Eingangsmengen zu einer Resultatsmenge). Für einen gegebenen Ausdruck gibt es eine Vielzahl alternativer Verarbreitungsreihenfolgen. Eine genaue Analyse dieser Vielfalt würde den Rahmen dieses Beitrags sprengen und der Leser muß z.B. auf [18] verwiesen werden.

Das Ziel der Suchstrategie innerhalb der Optimierung ist es, den Suchraum möglicher Verarbeitungsstrategien aufzuspannen und mit Hilfe einer Kostenfunktion (sh. z.B. [19]) einen kosten-minimalen Baum zu finden. Wir bedienen uns hier eines traditionellen 'dynamic programming' Ansatzes.

Betrachten wir zwei mögliche Verarbeitungsbäume für den folgenden Select-Ausdruck der die Menge aller Studenten beschreibt, die sowohl in den Studentenmengen `stud1` und `stud2` enthalten sind und die eine Vorlesung mit dem Titel "Hardware" besuchen (Klassendefinitionen sh. Abbildung 1):

```
{st1 | st1 in stud1, st2 in stud2, v in st2.vorl ::
st1.adr().equals(st2.adr()) && v.titel.equals("Hardware") }
```

Zwei mögliche Verarbeitungsreihenfolgen sind als Verarbeitungsbäume in Abbildung 3 linker und rechter Hand dargestellt. Linker Hand wird ein OJoin benötigt, um das Ergebnis des Flatten (ebnet den Pfadausdruck `st2.vorl()` ein) und Join operators miteinander zu verknüpfen. Rechter Hand stellt eine effizientere Verarbeitungsreihenfolge dar, die ohne OJoin auskommt.

Im allgemeinen entscheidet der Optimierer mit Hilfe einer Kostenfunktion, welcher der möglichen Verarbeitungsbäume ausgewählt wird. Die Kosten eines Baumes werden im wesentlichen durch die Selektivität der einzelnen Operatoren bestimmt (d.h. wie groß ist die Zwischenmenge, die ein Operator erzeugt).

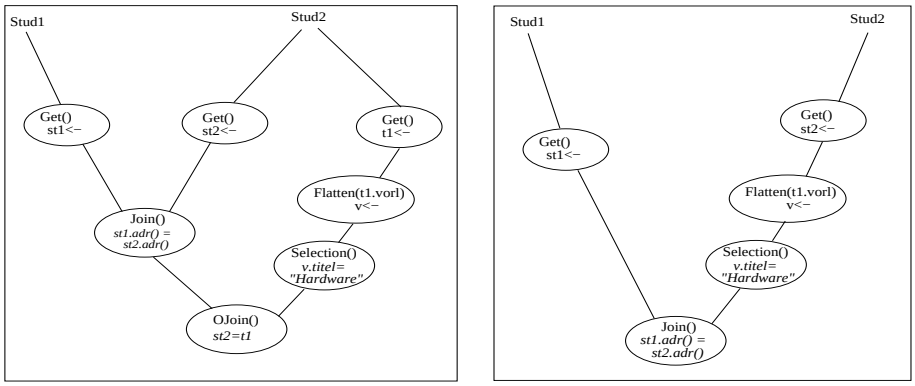

Abbildung3. Zwei mögliche Verarbeitungsbäume für den Select-Ausdruck.

Der Verarbeitungsbaum wird zusammen mit den optimierten Mengenausdrücken (sh. Abbildung 2) zu einem Ausführungsplan zusammengefaßt. Der Ausführungsplan ist eine Datenstruktur, die alle zur Laufzeit relevanten Informationen bezüglich der Mengen enthalten muß (z.B. die geschätzte Größe der Zwischenergebnisse, da diese zur Laufzeit überprüft werden).

4.2 Anbindung an den persistenten Speicher

Die Abarbeitung des Ausführungsplans stützt sich auf eine physikalische Mengenschnittstelle bestehend aus den physikalischen Mengenoperationen. Die konkrete Realisierung dieser Operationen wird von der physikalischen Mengenschnittstelle verborgen und ermöglicht so eine größtmögliche Transparenz. Die Schnittstelle beinhaltet Methoden zur Verwaltung der Gesamtheit der persistenten Mengen und Methoden zu deren Manipulation und Abfrage.

Bei der Abarbeitung des Ausführungsplans zur Laufzeit wird die Gültigkeit des Verarbeitungsbaums überprüft. Diese Überprüfung vergleicht die Größenverhältnisse der im Programm verwendeten Mengen zur Übersetzungszeit

gegenüber der Laufzeit. Nur eine signifikante Änderung der Größenverhältnisse (nicht nur eine Veränderung der Mengengrößen) führt zu einer Umformung des Verarbeitungsbaums. Diese Umformung wird nun nicht mit dem zur Übersetzungszeit verwendeten 'dynamic programming' Ansatz, sondern mit einer wesentlich schnelleren regelbasierten Optimierung durchgeführt. Dies ist möglich, da man davon ausgehen kann, daß der zur Übersetzungszeit erstellte Verarbeitungsbaum in der Regel noch eine relativ gute Verarbeitungsreihenfolge darstellt. Anschließend wird ein neuer Ausführungsplan erstellt.

Der so gewonnene Ausführungsplan, der eine Verarbeitungsreihenfolge festlegt, wird dann abgearbeitet und die Methodenaufrufe der physikalischen Mengenschnittstelle ausgeführt. Im typischen Fall enthält er Aufrufe zum Öffnen der Datenbank, zum Einfügen von Mengen in diese und Methoden zur Manipulation der einzelnen Mengen.

Bis jetzt stellen wir physikalische Mengenimplementierungen für zwei konkrete persistente Objektspeicher zur Verfügung. Erstens, der in einer Kooperation von der Universität Glasgow und SUN entstandene PJama [5] persistente Objektspeicher und zweitens, der PSE [20] von Object Store. Darüberhinaus wird auch noch eine einfache Implementierung mit der JDK Serialisierung bereitgestellt. Für die Zukunft ist auch die Anbindung an eine objektorientierte Datenbank mit Hilfe des in Entwicklung befindlichen ODMG-Java Standard [8] vorgesehen. Dieser Standard wird eine direkte Unterstützung des Java-Daten Modells realisieren: Klassenhierarchien, Objekte und Referenzen.

5 Zusammenfassung und Ausblick

Dieser Beitrag zeigt, daß die Erweiterung einer objektorientierten Sprache wie Java um Mengen es ermöglicht, den Zugriff auf persistente Objektspeicher transparent zu machen. Durch das Erzeugen von Datenstrukturen zur Darstellung von Ausführungsplänen und deren Anpassung zur Laufzeit werden die Vorteile von Optimierungstechniken aus der Datenbanktechnologie genutzt und ein effizienter Zugriff auf persistente Daten ermöglicht.

Die Anbindung an unterschiedliche persistente Speicher innerhalb eines Programms ist Gegenstand derzeitiger Untersuchungen. Wir wollen erreichen, daß der Zugriff ausschließlich durch *PSBezeichner* festgelegt wird, und der Code auch bei Anbindung an verschiedene, insbesonders auf verteilte persistente Speicher transparent gemacht wird. Mengen bilden für eine deklarative Formulierung derartiger Anforderungen ein geeignetes Mittel für die notwendige Abstraktion. Eine Partitionierung einzelner Mengen kann durch Verwendung des Select-Ausdrucks und der Mengenoperationen einfach beschrieben werden. Ein transparenter Zugriff auf verteilte Mengen wird somit ermöglicht. Das Anlegen von Duplikaten von Mengen, d.h. duplizieren der einzelnen Objekte, sehen wir als eine notwendige Voraussetzung an, um Operationen auf verteilten Mengen effizient umsetzten zu können.

Literatur

1. Jon Bosak (Sun MicroSystems). XML, Java and the future of the Web. Internet : http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlappls.html, März 1997.
2. L. Boeszoermenyi and K.-H. Eder. M3set – a language for handlincg of distributed and persistent sets of objects. *Parallel Computing*, 22(1):1913–1925, Januar 1997.
3. M. Jordan and M. Atkinson. First International Workshop on Persistence and Java. Technical Report TR 96-58, Sun Microsystems Laboratories, September 1996.
4. N. Paton, R. Cooper, H. Williams, and P. Trinder. *Database Programming Languages*. Prentice Hall, London, GB, 1996.
5. M.P Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *Sigmod Records*, 25(4):68–75, Dezember 1996.
6. David Jordan. *C++ Object Databases : Programming with the ODMG Standard*. Addison Wesley, 1997.
7. F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System, the Story of O_2*. Morgan Kaufmann, 1992.
8. F. Debatim. Java und Datenbanken - der ODMG-Standard zur objektorientierten Datenbank-Anbindung. *Java Spektrum*, 2, April 1997.
9. P. Watzlaw. JDBC - Datenbanken mit Java. *Java Spektrum*, 5, Dezember 1996.
10. D.S. Parker E. Simon and P. Valduriez. SVP - a Model Capturing Sets, Streams, and Parallelism. In *Proceedings of the International Conference on Very Large Data Bases*, Vancouver, British Columbia, Canada, August 1992.
11. D. DeWitt J Naughton J Shafer and Sh. Venkataram. Parallelizing OODBMS traversals : A performance evaluation. *Very Large Databases Journal*, 5(1):3–18, 1996.
12. J. Schwartz, R.B.K Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets - An Introduction to SetL*. Springer, 1986.
13. M. Gilula. *The Set Model for Database and Information Systems*. Addison-Wesley, 1994.
14. J. Gosling, B. Joy, and G. Steele. *Java - Die Sprachspezifikation*. Addison-Wesley, 1997.
15. R. G. G. Cattell. *The Object Database Standard : ODMG-93*. Morgan Kaufmann, 1993.
16. L. Brunie, H. Kosch, and W. Wohner. From the modeling of parallel relational query processing to query optimization and simulation. *Parallel Processing Letters*, 8(1):2–14, März 1998.
17. L. Fegaras. An experimental optimizer for OQL. Technical report, University of Texas at Arlington, Department of Computer Science and Engineering, 1997.
18. D.D. Straube and M.T. Ozsu. Query optimization and execution plan generation in object-oriented database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):210–227, April 1995.
19. L. Brunie and H. Kosch. Optimizing complex decision support queries for parallel execution. In *International Conference of PDPTA 97*, Las Vegas, USA, July 1997. CSREA Press.
20. G. Landis, C. Lamb, T. Blackman, S. Haradhvala, M. Noyes, and D. Weinreb. Objectstore PSE: A Persistent Storage Engine for Java. In *Proceedings of the 1st International Workshop on Persistence for Java*, Glasgow, Scotland, September 1996. Sun Microsystems, TR 96-58.